Optimization of source code for safety-critical systems

Wendlasida Ouedraogo

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2023IPPAX027

Thèse de doctorat

Source code optimization for safety critical software

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à École polytechnique

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Palaiseau, le 15/09/2023, par

M. WENDLASIDA OUÉDRAOGO

Composition du Jury :

Sylvain Baro Ingénieur Système, SNCF	Examinateur
Tristan Crolard Professeur, CNAM	Rapporteur
Pierre-Evariste Dagand Chargé de Recherche, CNRS & IRIF, Université Paris Cité	Rapporteur
Catherine Dubois Professeure, ENSIIE	Présidente du jury
Danko Ilik Ingénieur, CNES	Co-directeur de thèse
Jean-Marie Madiot Chargé de Recherche, Inria Paris	Examinateur
Claude Marché Directeur de recherche, Inria Saclay & LMF, Université Paris-Saclay	Examinateur
Lutz Straßburger Directeur de recherche, Inria Saclay & LIX, Ecole Polytechnique	Directeur de thèse

DOCTORAL SCHOOL

Résumé substantiel en français:

Un système informatique critique de sécurité désigne un ensemble de composants logiciels et matériels pouvant causer de graves dommages matériels, environnementaux ou humains. Dû à leur dangerosité, ce type de système nécessite l'implémentation de mesures de sécurité aussi bien au niveau logiciel que matériel pendant leur spécification, leur développement, leur compilation et leur exécution, les rendant ainsi complexes. Ces mesures de sécurité qui varient d'un système à un autre conduisent très souvent à des dégradations de performances, en particulier l'augmentation des temps d'exécution des systèmes concernés.

Le travail de recherche décrit dans ce document se situe dans le contexte du système CBTC (Communication-Based Train Control) de Siemens Mobility France, un système de contrôle de train certifié EN-50128 et SIL-4 (le plus haut niveau de sûreté) et utilisé dans le développement de nombreux systèmes de pilote automatique à travers le monde, en particulier sur les lignes 1, 4 et 14 du métro parisien. La chaîne de développement de ce type de système s'appuie sur deux technologies essentielles:

- La méthode B, qui permet de rédiger des spécifications formelles de programmes et d'obtenir, dans notre contexte, du code source VCP Ada (un sous-ensemble d'Ada) qui respecte ces spécifications,

- La technique du processeur codé dont l'implémentation logicielle et matérielle permet de détecter les erreurs de calculs et les incohérences dans le flot d'exécution entre le code source VCP Ada et le programme exécutable.

Si cette chaîne de développement fournit des garanties depuis la spécification à l'exécution, le mécanisme de détection d'erreur cause des ralentissements que les compilateurs optimisants habituels ne peuvent outrepasser.

Dans ce contexte, le but de cette thèse est de trouver des solutions pour réduire le temps d'exécution des systèmes CBTC tout en conservant les garanties de sécurité déjà acquises.

La réponse apportée par cette thèse est une optimisation formellement vérifiée du code source. Une première contribution est un compilateur VCP Ada vers VCP Ada, qui optimise le code source tout en préservant la sémantique des programmes. Ce compilateur a été implémenté avec l'assistant de preuve Coq et fournit des preuves en Coq qui garantissent l'équivalence en termes d'exécution entre le programme original et le programme optimisé. Ce compilateur tient aussi compte des complexités liées aux mesures de sécurité matérielle qui sont potentiellement incompatibles avec l'utilisation des compilateurs formellement vérifiés existants. Par ailleurs, le choix de l'application des optimisations sur le code source présente des avantages méthodologiques par rapport aux optimisations utilisant de nombreux langages intermédiaires, car ils permettent de simplifier et de réduire l'effort de preuve nécessaire.

Si CompCert et d'autres travaux de recherche fournissent des techniques qui nous ont servi de base dans le développement de ce compilateur, il faut reconnaître que la vérification formelle des analyseurs lexicaux, un des premiers composants logiciels dans le processus de compilation, n'a jusqu'à présent, pas reçu beaucoup l'attention de la part des chercheurs. Afin de ne pas créer un maillon faible dans le compilateur que nous avons développé durant cette thèse, notre deuxième contribution a consisté au développement de CoqLex, le premier générateur d'analyseur lexical formellement vérifié en Coq, basé sur une modification d'une implémentation existante en Coq des expressions régulières via les dérivés de Brzozowski. Cette contribution apporte aussi un générateur d'analyseur lexical qui produit le code Coq d'un analyseur lexical à partir d'une description dont la syntaxe est proche de celle de Ocamllex, un générateur d'analyseur lexical non vérifié pour Ocaml.

La théorie et les outils développés ont été utilisés pour optimiser les programmes VCP Ada des systèmes CBTC, composés de milliers de fichiers sources, avec des résultats prometteurs.

Institut Polytechnique de Paris
91120 Palaiseau, France

Contents

Chapter 1

Introduction

A safety-critical system is a system whose failure could result in dramatic consequences such as loss of life, significant property damage, or damage to the environment [Knight, 2002]. Usually found in application areas such as medical devices, weapons, nuclear systems, and transportation systems, those kinds of systems are subject to national/international regulations and standards [Boulanger, 2015, Brosgol and Comar, 2010, Potii et al., 2015]. Those standards require safety guarantees like the absence of bugs, leading system providers to find solutions to implement trustworthy software and prove their correctness. Building software that meet the specification required by such standards requires the use of particular techniques in their specification, architecture, verification, and process.

However, on the one hand, those techniques turn out to slow down the execution speed of critical software, and on the other hand, most standards impose restrictions making those kinds of software non-trivial to optimize. For instance, because they demand traceability between source code and binary code, they forbid performing non-local code transformation and hence forbid the use of common optimization techniques at the compiler level.

Techniques to overcome the antagonism between safety guarantees and execution time performance vary from one system to another, because there is no standard way to develop such a system –the architecture of each critical system is unique. This research aims to present new optimization techniques that would meet the applicable standards, as identified and applied to a real and functioning critical system, the autopilot for driverless trains of Siemens Mobility France. The research also led us to contribute to the strengthening of a weak chain in current formally proven compilation toolchains, the lexical analysis.

1-I Background and context

The European standard EN 50128 [Myklebust et al., 2015] "Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems" specifies procedures and technical requirements for the development of programmable electronic systems which are used in railway system development for the control and protection applications. It requires that all systems with safety implications that contain software should be assigned a Software Integrity Level (SIL), ranging from a value of 0 (the lowest accepted safety level) to 4 (the highest one) [Jansen and Schäbe, 2004].

Siemens Mobility France is the World's foremost supplier of computer systems (hardware and software) that serve for automatic control and operation of metro trains and in particular driver-less trains. The systems it develops are SIL4 certified –the probability of failure of such systems is lower than 10^{-8} per hour. Building a SIL4 software requires providing the guarantee this software executes as expected. To achieve this goal, Siemens researchers and engineers used a set of techniques and processes: the B-method [Lano, 2012] to build source code equipped with formal proofs of its correctness, the Vital Coded Processor technique [Chapront, 1992] that ensures that the compiled code runs safely on the

computer platform without any safety requirement on the CPU, and the DigiSafe® XME architecture [Lejeune et al., 2004] that protects from compiler and processor errors.

Developed by J.R. Abrial, the B-method refers to a development process and associated tools that make it possible to implement and prove the correctness of safety-critical programs. In this process, developers must first specify the behaviour of the safety-critical program to be implemented using the B specification language. Then, using the B implementation language and a semi-automatic theorem prover, the developers must write an implementation of this specification and prove its correctness. These proofs are checked by the B proof assistant and, if accepted, the implementation is translated into Ada source code that can be used for functional testing.

At Siemens, this method was first used to write and prove the absence of bug in the source code of the auto pilot of Paris Line 14. However, this method does not protect from compiler and processor errors and hence lead to the Vital Coded Processor technique and the associated DigiSafe® XME architecture.

The Vital Coded Processor technique is a technique allowing to detect incoherences between the intended behaviour of the source code and the execution behaviour of binaries produced from the code source. It uses a probabilistic approach, associated with a compilation and execution architecture allowing to ensure the coherence between the executed program and the input source code. This technique is proven to have a failure rate (probability of not detecting incoherences) of less than 3.15×10^{-15} per hour. Associated with the B-method and the DigiSafe® architecture, the Vital Coded Processor technique allows to ensure coherence between program specifications and execution but turns out to slow down the execution time of the software implemented using them. The aim of this research is to find and implement techniques to overcome the antagonism between safety guarantees and execution time performance of safety-critical software generated using this method, such that are generic enough to be transposable in other contexts as well.

1-II The research problem and objectives

To achieve the requirements of the EN 50128 certification, Siemens Mobility developed a development process that ensures coherence between the formal specification and its execution. However, this development process leads to programs whose worst-case execution time can easily reach an upper bound of, say, 200 milliseconds. This value can be interpreted as the reaction time of the autopilot software in a driverless train system and has a great impact on the quality of service of the whole system. The higher it is, the longer it takes for the autopilot program to make a decision.

Optimizing the execution time of programs is important for software providers and their clients, particularly in railway systems. Numerous studies have investigated program optimization techniques, from those applicable at the source code level [Grove and Torczon, 1993] to those applicable at the level of compilation [Jones, 2005]. Using programming language processing tools and methods, those studies provide solutions and techniques that reduce the execution time of compiled programs. But, most of those studies focus on performance and do not provide safety or correctness guarantees. This makes those techniques not usable in safety-critical software without further work on correctness, if at all possible.

Exceptions like the CompCert project [Leroy et al., 2016], a formally verified optimizing compiler, overcome the antagonism between safety guarantees for programs and optimizing their execution time. Thanks to their formal proofs, these compilers are guaranteed not to introduce any difference in behaviour (except for speed-up) between optimized binaries and non-optimized ones. However, it is not clear whether one could readily use the CompCert compiler for an existing industrial control system like those implemented using the DigiSafe® development method.

In particular, the research would have to address the following questions:

1. Can the source code obtained from the B-method be transformed into a code compilable by CompCert? In fact, while the current method extracts VCP Ada, a subset of Ada [Taft, 1997] code from the B-method, CompCert processes languages such as C [Ritchie et al., 1988] and Cminor.

2. Can the CompCert compilers emit code that can execute on the DigiSafe® platform? For instance, the DigiSafe® platform requires a binary to be equipped with a special trace to be executed on a co-processor, at the same time as the main program executing on the main processor.
3. Can the use of the CompCert compiler reduce the execution time in spite of the limitations on performance posed by the DigiSafe® platform?
4. Is it possible to weave CompCert into the DigiSafe® XME architecture and to produce a qualification argument for the whole compilation chain?

There are studies that present solutions to make CompCert compile and optimize code from other languages. Projects such as MLCompCert [Dargaye, 2009], a verified compiler for MiniML, and Vélus [Bourke, 2021], a verified compiler for Lustre [Bergerand, 1986] lead to formally verified compilers that are obtained by writing a bridge between MiniML or Lustre and Cminor.

Except solutions that suggest the use of CompCert, are there other approaches? How do those approaches ensure their correctness? How efficient are they? How much will they affect the current development process?

In *Source code optimization techniques for data flow dominated embedded software* [Falk and Marwedel, 2004], Heiko Falk and Peter Marwedel have shown that source code optimizations have an equivalent impact on execution time as compiler optimizations. They also proposed efficient optimizations and techniques to implement them. However, they do not provide proof of their correctness.

The aim of this research is to reduce the execution time of the software implemented using the DigiSafe® development process, while, in parallel preserving their safety guarantees. To do that, we will evaluate 2 possibilities to modify this process:

1. Replace the current compiler with a formally verified optimizing one such as CompCert.
2. Implement a formally verified tool to perform optimizations on source code.

This study will contribute to the body of knowledge on formal methods, compilation, and source optimization techniques. It will also tackle some of the challenges of current industrial systems, much wider than only the railway sector.

1-III Structure of this document

Chapter 2 of this document will present the background and the state of the art for this research. It will present the B-method, Vital Coded Processor, DigiSafe® development process and its implementation. It will also give an overview on compiler architecture, certified compilers such as CompCert, and on the Coq proof assistant that is used to prove the correctness of CompCert.

Chapter 3 presents the Siemens development process, the process used to generate the programs this thesis aims to optimize. Regarding that process, Chapter 4 will explore the optimization possibilities, explain why VCP Ada (a subset of Ada) source code optimization appears to be the best optimization option and how we will implement and formally verify a VCP Ada to VCP Ada compiler that will perform those optimizations. This implementation led us to identify a gap in the compilation toolchain of certified compilers such as CompCert, at the level of lexer generation. Consequently, we implemented CoqLex [Ouedraogo et al., 2021], a formally proven meta-tool used to generate lexers, which will be presented in Chapter 5, the Chapter that is in charge of producing a front-end (the first component of the compiler) equipped with formal proofs of its correctness.

Chapter 6 presents the formal semantics of VCP Ada and the semantic preservation definition that is used to prove the correctness of the middle-end described in Chapter 7. We describe the back-end of the VCP Ada to VCP Ada compiler we implement in Chapter 8, evaluate the performances of its optimizations and discuss future works in Chapter 9.

Chapter 2

Background and state of the art

This chapter introduces the basic knowledge of published technologies and concepts used in this research. The first section introduces the Coq proof assistant, a platform for the formalization of mathematics and the development of programs. The second one presents the B method, a formal method used to implement and reason about programs. The third one details the theoretical concepts of the Vital Coded Processor (VCP) technique, a system to protect software execution against hardware and compilation errors, and the last part presents compiler design concepts.

2-I Coq proof assistant

Proof assistants are software that provide a formal language to write mathematical definitions, prove theorems, and sometimes, as in the case of Coq, write executable algorithms. Throughout the years, they have shown their ability to prove the correctness of important and complex pieces of software like the complete toolchain of the DeepSpec project [Weng, 2016]. During this thesis, we implemented and used tools developed using this platform, and used Coq to prove our own optimizations correct. We will present here the features of the Coq syntax that are necessary for being able to read the Coq source code extracts given in the thesis.

2-I.A Basic terms

In Coq, objects have names and types. The Coq documentation defines 4 categories of types:

Set intends to be the type of small sets. It includes data types such as booleans (true, false) and naturals (0,1...), but also products, subsets, and function types over these data types.

Prop intends to be the type of logical propositions. If M is a logical proposition then it denotes the class of terms representing proofs of M. An object m belonging to M witnesses the fact that M is provable. An object of type Prop is called a proposition.

SProp (strict propositions) is an experimental construction that is similar to Prop but whose propositions are known to have irrelevant proofs [Gilbert et al., 2019].

Type is the type of all types, in particular the types constructible from the above-mentioned types.

For our work, we only used Prop and Set. Let us illustrate those concepts using examples: Symbol `1` defines a constant value of type `nat`, a predefined type for natural numbers whose type is `Set`. The binary function `leb` (less-or-equal comparison) has type `nat -> nat -> bool` (or `nat -> (nat -> bool)`).

When leb is applied to a nat term t (using the notation leb t), the result type is a function of type nat -> bool. This is called **partial application**.

Let us present the Coq syntax for logical propositions:

Math syntax	$\top$	$\bot$	$\neg$ P	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$	$x = y$	$x \neq y$	$\forall x, P$	$\exists x, P$
Coq syntax	`True`	`False`	`~P`	`P /\ Q`	`P \/ Q`	`P -> Q`	`P <-> Q`	`x = y`	`x <> y`	`forall x, P`	`exists x, P`

Example. We can then state the following propositions:

- $\forall x, 0 \leq x$: `forall x, leb 0 x = true`.
- $\forall x, x \leq 0 \Rightarrow x = 0$: `forall x, leb x 0 = true -> x = 0`.

To prove properties, Coq suggests the use of **tactics** to do backward reasoning. At the beginning of the proof, the proposition to prove is the current goal. Then, the developer has to enter tactics that will transform the current goal into a set of sub-goals to be proven. The proof is finished when there are no remaining sub-goals. Propositions can be declared using the following syntax:

`Theorem` *name* : *prop*.

where *name* is the name of the property (object) and prop is the statement of the property. Theorem can be replaced by `Lemma`, `Goal`, `Fact`, `Remark`, `Corollary`, `Proposition`, or `Property`.
Objects can be given a name by using the following syntax:

`Definition` *name args* : *type* := *term*.

This command checks that the type of $term$ is $type$ and then binds $name$ to the object $term$. When $term$ is omitted, $type$ is required and Coq enters proof mode. This can be used to define a term incrementally. $type$ can be omitted when $term$ is set. $args$ parameter is used to set parameters. In that case, this command defines a function.

Examples:.

```
Definition zero := 0. (*binding the name zero to value 0*)
Definition id (n: nat) : nat := n. (*definition of identity function for natural numbers*)
```

The keyword `Definition` can be replaced by `Example`.

Coq provides a command allowing to define records as in many programming languages. The syntax for this command can be described as follows:

`Record` *name* : `Set` := *constructor_name* { *field_name*$_1$: *type*$_1$; ... }.

This command defines a record type whose name is $name$, a constructor named *constructor_name* for this record type whose fields are *field_name*$_1$... Fields can be any term: propositions, functions or simpler type such as nat. `Record` can be replaced by `Structure`. The definition of a record can be done using the following syntax:

{| *field_name*$_1$:= *value*$_1$; ... |}

or

constructor_name *value*$_1$...

The projection can be done using the following syntax:

variable_name. (*field_name*)

Example: This example defines the type of rational numbers and defines the rational 1/2.

```
Record Rat : Set := mkRat
 { sign : bool
 ; top : nat
 ; bottom : nat
 ; Rat_bottom_cond : 0 <> bottom
 ; Rat_irred_cond :
     forall x y z:nat, (x * y) = top /\ (x * z) = bottom -> x = 1
 }.

 Theorem two_is_not_zero : 0 <> 2.
Admitted.

 Theorem one_two_irred : forall x y z:nat, x * y = 1 /\ x * z = 2 -> x = 1.
Admitted.

Definition half := mkRat true 1 2 one_is_not_zero one_two_irred.
```

2-I.B Inductive declarations

Inductive definitions are one of the main ingredients of the Coq language. They allow describing different notions such as data types, logical connectives, primitive relations, and to perform proofs on them using the principle of mathematical induction.

2-I.B.1 Inductive data types

Coq allows us to declare a data type by specifying a set of constructors. A constructor can be seen as a function that when applied to all its arguments returns an object of the type we are defining. After reading the definition, Coq makes sure that the definition is well-founded. The syntax to declare an inductively defined type is:

$$\texttt{Inductive}\ \ name : sort := c_1 : C_1 | ... | c_n : C_n.$$

where *name* is the name of the type to be defined; *sort* is one of Set or Type (or even Prop); c_i are the names of the constructors and C_i is the type of the constructor c_i. To make recursive constructors, C_i can describe a function that takes arguments of type *name*. After validating an inductive definition, Coq system generates primitive objects and theorems expressing that *name* is the smallest set containing the terms built only with the constructors given in its definition.

Example: The data type of natural numbers and natural number list can be defined inductively as follows:

```
Inductive nat : Set :=
  O : nat
| S : nat -> nat.


Inductive natList : Set :=
  LEmpty : natList
| LCat : nat -> natList.
```

Coq also allows to implement parametric (or polymorphic) inductive data types.
Example: The polymorphic list data type is defined inductively as follows:

```
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A -> list A -> list A.
```

2-I.B.2 Definitions by pattern-matching

The *Pattern-Matching operator* allows building a case analysis over the various constructors of a term that belongs to some inductive type. To do that, we can use the following Coq syntax:

match *term* **with** $|c_1\ args_1 => result_1 ... |c_n\ args_n => result_n$ **end**.

Coq allows to define a new term using other terms or by case analysis.
Example: This example uses the case analysis on nat to return true if and only if the input nat is equal to zero

```
Definition iszero n :=
        match n with
        | O => true
        | S x => false
        end.
```

2-I.B.3 Fixpoint definitions

To define recursive functions, Coq allows to do structural recursion by fixpoint definition. Structural recursion means that each recursive call is performed on a structurally smaller argument. The syntax for fixpoints is:

Fixpoint *name* $(x_1 : type_1) ... (x_p : type_p)$ {`struct` x_i} : *type* := *term*.

The variable x_i following the struct keyword is the recursive argument. Its type $type_i$ must be an instance of an inductive type. The type of *name* is `forall` $(x_1 : type_1) ... (x_p : type_p)$, *type*.

Examples: Let us see a fixpoint definition for leb

```
Fixpoint leb (n: nat) (m: nat) := match n with
| O => true
| S x =>
        (match m with
        | O => false
        | S y => leb x y
        end)
end.
```

This example defines the operator ≤ by case analysis on n and on m.

2-I.B.4 Inductive relations

Inductive definitions can be used to define relations. This definition is specified by a set of properties similar to inference rules. The syntax of such a definition is:

Inductive *name* : *arity* := $c_1 : C_1 | ... | c_n : C_n$.

where *name* is the name of the relation to be defined, *arity* its type (for instance `nat->nat->Prop` for a binary relation over natural numbers) and, as for data types, c_i and C_i are the names and types of constructors respectively.

Example: Let us define the order relation over natural numbers verifying the following rules:

$$\forall n : \texttt{nat}, 0 \leq n \qquad \forall n\ m : \texttt{nat}, n \leq m \Rightarrow (\texttt{S}\ n) \leq (\texttt{S}\ m)$$

This definition can be represented as the following set of inference rules:

$$\frac{}{0 \leq n} \qquad \frac{n \leq m}{(\texttt{S}\ n) \leq (\texttt{S}\ m)}$$

In Coq, this can be defined as follows:

```
Inductive LE : nat -> nat -> Prop :=
| LE_0 : forall n : nat , LE 0 n
| LE_S : forall n m : nat , LE n m -> LE ( S n ) ( S m ).
```

This declaration introduces identifiers LE, LE_0, and LE_S, each having the type specified in the declaration and can be used as axioms for other proofs.

2-I.C Program specification and implementation with Coq

In Coq, an object of type A -> B is a total function, that is, for all given values of type A, the evaluation must always terminate and give a value of type B. To implement partial functions, there are other possibilities.

2-I.C.1 Use of option type as a validation result for preconditions

The option type is defined as follows:

```
Inductive option (A:Type) : Type :=
  | Some : A -> option A
  | None : option A.
```

They are used as containers to represent the presence (Some) or the absence (None) of value. The main drawback of this method is that the type of the function becomes A -> option B. So, the use of this function always implies a case analysis.

2-I.C.2 Use of logic constructions to define preconditions

Coq allows to mix freely types and properties so, the users can define preconditions using properties. For instance, let us take a function A -> B to which the developers want to add preconditions. To do so, they can define the function domain, dom, using an inductive relation (for example). They can then redefine their function to add this domain. The type of that function becomes:

$$\forall x : A, dom\ x \rightarrow B$$

Each call to $f\ a$ requires a proof p of $dom\ a$ and will be represented as: $f\ a\ p$. We can partially hide the proof in a subset type: $f : \{x : A | dom\ x\} \rightarrow B$.

2-I.C.3 Define functions as relations

In that case, a function f of type A -> B is rewritten by a relation F that has the type A -> B -> Prop. Then we have to prove the functionality of F: $\forall a\ b\ c,\ F\ a\ b \wedge F\ a\ c \Rightarrow b = c$. This relation means that for all input, there is at most one output. Each time we want to mention $f(x)$, we will have to introduce a variable y and a hypothesis $F\ x\ y$.

2-I.D Extraction and programs

Extraction is an important Coq feature. It enables developers to write a functional program inside Coq and translate it into an OCaml, Haskell, or Scheme source code that can be compiled and executed. The main motivation for this extraction mechanism is to produce certified programs: each property proved in Coq will still be valid after extraction. Using this feature, the developers can write their program differently: Usually, the developers detect errors during testing and may miss some of them. With Coq, the developers can use case analysis to have a global view of all the possible executions of the functions they write and prove the correctness of those functions in all of those cases.

2-II B-method

Developed by Jean-Raymond Abrial, B is a formal method for specifying, designing, and coding software systems. It uses concepts of first-order logic, set theory, and integer arithmetic and has earned the trust of its users with its commercially available tools support for specification, design, proof, and code generation. The term B-method refers to the software development method based on B whose theoretical basis is the use of Hoare triples [Hoare, 1969]. Programs are described using the B notation, which is suitable for specification and classic imperative constructs for software programming. In this document, we are presenting Classical B, the first version of the B-method as described in the B-Book [Abrial, 1996]. It is important to mention that there exists an event-based version of the B-method called Event-B [Abrial, 2010], used for event driven reactive systems. In this section, we will present the basic concepts of the B method and its impact on industry.

2-II.A Main concepts

The B-method structuring mechanisms are based on 3 main concepts: machine, refinement, and implementation.

Machine refers to an abstract state machine whose state is represented by a set of variables and transitions are represented by a set of operations with preconditions. Its state constraints are described by invariants – conditions that must always be verified.

A refinement is a module that refines a machine or another refinement. In refinements, the developer can add preconditions to specify implementation choices. Each refinement must also prove its coherence with the machine or refinement it is refining. The proof obligations are generated by a validation system and the proof can be done using an automatic proof assistant.

Implementation is the most concrete level of specification. It is written using a subset of the B notation called B0, which consists of more concrete constructs that are similar to those available in most programming languages, making the implementation suitable for a translation to a programming language.

Those concepts and associated proofs allow to:

- provide clear structured technical and system specifications
- provide guarantees that those specifications are coherent and unambiguous
- provide guarantees that the final software is fault-free

Let us see an example. In this example, we want to implement a simple drink dispenser. This drink dispenser has 2 operations:

- **give**: to serve a drink and make sure to update the availability of drinks.
- **refill**: to add drinks to the dispenser when it is empty.

The machine can be described as follows:

```
MACHINE
    DrinkDispenser
ABSTRACT_VARIABLES
    empty
INVARIANT
    empty : BOOL
INITIALISATION
    empty := FALSE
OPERATIONS
    give =
```

```
        PRE
            empty = FALSE
        THEN
            CHOICE
                skip
            OR
                empty := TRUE
            END
        END ;

    refill =
        PRE
            empty = TRUE
        THEN
            empty := FALSE
        END
END
```

In this machine, the developer made the choice to use one boolean variable. Skip refers to the absence of action. So we can see that when the dispenser gives a drink, the value TRUE can be assigned to empty (when the dispenser gives its last drink). To be refilled the dispenser has to be empty – the precondition of this function is that empty must be equal to true. We can also notice that the invariant is the type of empty remains bool.

The next step is to propose a refinement of the dispenser. In this refinement, we suggest setting a capacity for the dispenser and a variable that holds the number of drinks remaining in the dispenser.

```
REFINEMENT
    DrinkDispenser_r
REFINES
    DrinkDispenser
VARIABLES
    supply
INVARIANT
    supply : 0..10 &
    ((empty = TRUE) <=> (supply = 0))
INITIALISATION
    ANY
        nn
    WHERE
        nn : 1..10
    THEN
        supply := nn
    END
OPERATIONS
    give =
        PRE
            supply > 0
        THEN
            supply := supply - 1
        END ;

    refill =
        PRE
            supply = 0
        THEN
            supply := 10
        END
END
```

This listing shows us that the capacity of the dispenser is 10. Variable supply stores the number of drinks available in the dispenser. So, its value has to be between 0 and 10. There is a relation between empty and the supply. Those two conditions are the invariant of this refinement. We can then see that preconditions are adapted using the supply variable. Now we can provide an implementation of this drink dispenser.

```
IMPLEMENTATION
    DrinkDispenser_i
REFINES
    DrinkDispenser_r
CONCRETE_VARIABLES
    conc_supply
INVARIANT
    conc_supply : 0..10 &
    conc_supply = supply
INITIALISATION
    conc_supply := 10
OPERATIONS
    give =
        BEGIN
            conc_supply := conc_supply - 1
        END ;

    refill =
        BEGIN
            conc_supply := 10
        END
END
```

2-II.B Consistency and refinement proofs

The correctness of this method relies on the verification of the following properties. For every machine, consistency proof consists in proving that:

- there exist function parameters that meet the preconditions on the operations of the abstract machine.
- there exist values that meet the invariant of the state machine.
- each operation called under its precondition preserves the invariant.

A refinement is a way to rewrite abstract machines using more concrete (opposite of abstract) data structures. For every refinement, the coherence proof consists in proving that for any operation:

- the preconditions of the refined model are compatible with those described by the model it refines.
- the changes made by the operation are compatible with the invariants and the operation description (post-conditions)

2-II.C B-method as a software development process

The B-method allows to implement software by starting with an abstract machine specification, followed by incremental refinements until it reaches an implementation. At the implementation level, only imperative-like constructs may be used. Such an implementation is then translated into source code in a common programming language using code generation tools. The B development process can be summarized by the following graphic.

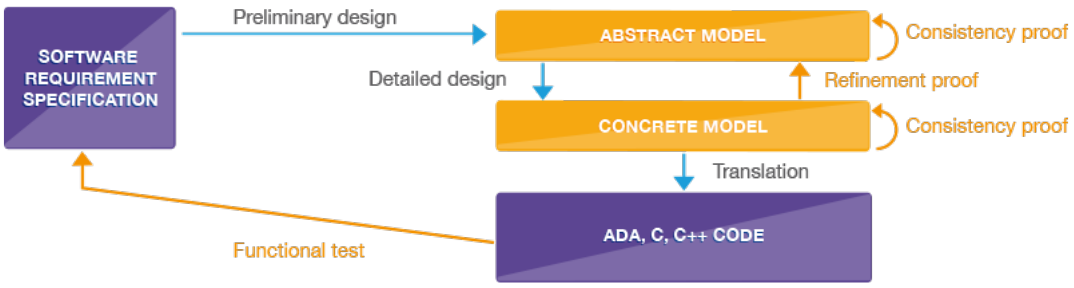


Figure 2.1 *– B-method software design*
Source: https://www.atelierb.eu/en/presentation-of-the-b-method

2-II.D Tool Support

The B-method is supported by tools that contribute to making the specification and verification of the models easier. The most popular ones are Atelier B [Clearsy, 2003], ProB [Leuschel and Butler, 2008]

Atelier B Implemented by ClearSy [Clearsy, 2003], Atelier B is a commercially available tool that allows an industrial use of the B-method. It has many helpful components such as:

- an editor for the specification of modules using the B notation
- automatic generation of proof obligations
- automatic provers for the proof obligations
- an interactive prover for proofs that cannot be performed automatically
- an assistant to help during the refinement process

The tool also assists developers during project management, distributed development, and documentation. AtelierB has an educational free version for Windows, Linux, and OS X. Also, some of its components are open source.

ProB refers to an animator and model checker for the B-Method. The tool allows to check models automatically for inconsistencies such as invariant violations, deadlocks, and others. It also provides graphic visualizations for the models, has constraint-solving capabilities, and also has testing capabilities. It is free, open-source, and has versions for Windows, Linux, and OS X.

2-III Vital Coded Processor

The B-method allows to ensure that pieces of source code conform to their specification. But a correct source code can execute with errors. In fact, compilers and processors are subject to faults [Leroy, 2011, Ayatolahi et al., 2013]. For this reason, Siemens Mobility France created and developed the Vital Coded Processor (VCP) [Dollé, 2006] technique. This error detection system uses a probabilistic approach to detect certain inconsistencies between software source code and its execution. To do that, this technique suggests to associate variables, operations, and the memory state to pre-determinable values (values that can be computed at compile-time) that are called signatures. During the execution of such programs, those signatures are then re-computed and compared to the values computed at compile-time. If the values computed during the execution are different from those computed at compile-time, then it means that something went wrong. This section presents the main concepts of this technique.

2-III.A Variable coding

Variable values of variables are divided into 2 parts:

the functional part that forms the usual part of the data.

the control part that is made up of the arithmetic sum of three terms:

- $r_k(x)$: Let A be a large prime number chosen randomly called the code key, x, the functional value of the variable, k, an arbitrary number such as $2^k > A$, we define $r_k(x)$ as $2^k x \ mod \ A$.
- B_x: Also called the static signature, this value depends on the operands and the operator used during the last assignment of the variable x.
- D: a dynamic signature, common to all data, which guarantees the freshness of the data in cyclic programs. This value is incremented by a constant value on every application cycle as well as on each iteration in the internal loops. It ensures that the previous value of a variable cannot be used after an update.

Variable values can be represented in form of a pair $(x, \ -r_k(x) + B_x + D)$ where x is the functional part and $-r_k(x) + B_x + D$ is the control part.

Notes:

- in the following sections, X_F and X_C will refer respectively to the functional part and the control part of variable X.
- the term *vital variables* will be used to designate variables that are associated with signatures.

2-III.B Operation Coding

Let X and Y vital variables and op a binary operation, for example, addition or multiplication. The result of $X \ op \ Y$ can be given by the following formulas:

$$(X_F \ op \ Y_F, G_{op}(X, Y))$$

$$G_{op}(X, Y) = (-r_k(X_F \ op \ Y_F) + F_{op}(B_X, B_Y) + D) \ modulo \ A$$

The static signature of the result is then equal to $F_{op}(B_X, B_Y)$.
For every operation op, F_{op} is implemented to allow the detection of operands and operation errors.

Example: Let us suppose that one wants to compute '$X \ op \ Y$'. The expected result is $(X_F \ op \ Y_F, -r_k(X_F \ op \ Y_F) + F_{op}(B_X, B_Y) + D)$.

Detecting operand errors: If the operand T is used instead of Y, the computed result would be $(X_F \ op \ T_F, -r_k(X_F \ op \ T_F) + F_{op}(B_X, B_T) + D)$, which is different from the expected result because static signatures are unique for each variable ($B_T \neq B_Y$).

Detecting operator errors: If the operation op' is used instead of op, the result would be $(X_F \ op' \ Y_F, -r_k(X_F \ op' \ Y_F) + F_{op'}(B_X, B_Y) + D)$, which is different from the expected result. An error will be detected if $F_{op}(B_X, B_Y) \neq F_{op'}(B_X, B_Y)$.

In both of the above error cases, the resulting signature is incorrect as the expected static signature would be different from the one that is expected.

2-III.C Securing the execution flow

Execution errors are detected by an internal coded data called a "tracer" and noted as S. It is dependent on all of the operations performed since the application was initialized. It does not have a functional part. Starting with a randomly chosen value S^0, its evolution can be expressed by the following formula:

$$S^{n+1} = \alpha(B_z + S^n)$$

where α is a constant and B_z is the static signature of the result of the last executed operation. As B_z is pre-determinable (t), S is also pre-determinable. During the execution, the tracer is computed and compared to the pre-computed one. If the results diverge, it means that an error is detected. In that case, the tracer is said to be ”out-of-code”. This variable helps to make sure that the sequence is respected and that no permutation is done.

2-III.C.1 Branching

When a conditional branching function is used, the tracer signature must ensure that the taken branch is the right one. But, this signature must, on one hand, be pre-determinable and on the other hand depend on the functional value of the condition as the tracer signatures and the updated variables change differently, depending on which branch is followed (as the operations are branch-dependent). Let us consider an if-then-else branch. To guarantee these two points, the following processing is performed:

- For any branch, the evolution of static signatures is precomputed.
- The value of the tracer is different for each branch. For example, in an if-else branch, its value is calculated taking into account a value that is equal to 1 (when the branch condition is evaluated to true) in the then-branch and equal to 0 in the else-branch: If X is the branch condition, the tracer value becomes $S^{n+1} = \alpha(r_k(X) + B_X + S^n)$ where X is 1 (true) or 0 (false).
- At the end of the branch, the signature of the tracer and all the modified variables are incremented by constants depending on the branch they come from – This operation is called compensations and is executed by convergence instruction. Those pre-computed constants, stored in a data structure called *compensation table*, are unique for every converge instruction and are necessary to make sure that the signatures and the tracer will remain pre-determinable after the branch.

Example: Let us consider the following code

```
1 ...
2 X := E op F;
3 if C then
4         X := X op G
5 else
6         X := X op H
7 end if;
8 converge;
```

The signature of X changes in the following way:

line number	signature of X when then branch is taken	signature of X when else branch is taken
1	Bx_1	Bx_1
2	Bx_2	Bx_2
4	Bx_3	
6		Bx'_3
8	$Bx_4 = Bx_3 + comp_{true}$	$Bx_4 = Bx'_3 + comp_{false}$

After the `converge` instruction, the signature of X is Bx_4, regardless of the taken branch, that is pre-determinable.

2-III.C.2 Loops

For repetitive structures (loops), the number of iterations performed must be correct. Furthermore, the tracer signatures and the signatures of the variables updated in the loop must be independent of the number of iterations performed. If signatures were dependent on the number of iterations, signatures would not be pre-determinable and the system would not be able to validate the results. To ensure that, a technique, similar to the one used for if-else instruction is applied: compensation. Before entering in a while instruction, the signature of the tracer and the signature of each variable that can be modified in the loop is modified. At the end of each iteration, the signature of each of those variables is modified so that those signatures remain pre-determinable regardless of the number of iterations.

2-III.D Error detection

The VCP technique does not of course allow pre-determining calculation results, but it pre-determines a coding that detects when an error occurs. Error detection consists in comparing the static signatures and the value of the tracer to their pre-computed values at each operation. This allows to detect operand, operator, and sequence errors. When correctly implemented, the probability of failing to detect errors is lower than 1/A (where A is the large prime chosen randomly that we mentioned in the *variable coding section*). This result comes from the fact that the signatures of variables are considered to be randomly distributed and their values are between 0 and $A-1$. Failing to detect an error means that the signature of the variable implicated in that error is correct (the probability of this event is $1/A$). The details of the Siemens VCP implementation can be seen in Chapter 3.

2-IV Compilers

Compilers are computer programs that translate a program description written in one programming language (the source language) into a description written in a (different or not) programming language (target language). Those descriptions are notations used to give instructions to a computer. In this section, we simplify and present the basic notions found in Compiler design [Wilhelm and Maurer, 1995].

2-IV.A Compiler architecture

Compilers are generally divided into 3 parts. The first one is the *front-end*. It is in charge of reading the source code (from a file) and transforming it into a computer object usually called parse tree or abstract syntax tree (AST). The second one is the *middle-end*, which modifies and/or transforms the AST into an Intermediary Representation (IR) suitable for the *back-end*, the third part of compiler architecture that is in charge of translating the input IR into the target file, written in the target language.

2-IV.A.1 Compiler front-end

The front-end is the first stage of the compilation. It is traditionally made of 2 tools:

A lexer: It is in charge of the lexical analysis which is the first phase of a compiler. This task consists in the transformation of the sequences of characters of a source code into a sequence of tokens (strings associated with meaning). It also removes any extra space or comment written in the source code. It can also detect invalid or forbidden sequences of characters. The implementation of lexers is usually generated by lexer generators like lex/flex [Levine et al., 1992] and ocamllex [Smith, 2007]. This generation is done from a specification that can be described as regular expressions (to recognize a sequence of characters) associated with semantic actions (to return the token that would correspond to the recognized sequence of characters). We give more details about those notions in Chapter 4.

A parser: It is in charge of syntax analysis which is the second phase of a compiler. Its role is to check the validity of the order of tokens sequence and so determine if a source code respects the syntax of a language. Parsers are usually generated using parser generators like yacc [Johnson et al., 1975], ocamlyacc [Smith, 2007] and menhir [Régis-Gianas, 2016]. The description of the generation is done in form of context-free grammar [Cremers and Ginsburg, 1975] whose productions are associated with object constructors. This organization allows to build an object representing the whole code by using the successive constructors from the derivation tree which is usually called abstract syntax tree (AST).

In the front-end, one can also perform semantic analysis such as type checking, reserved identifier miss-use, multiple declarations or non-declaration of a variable, by writing and calling functions that analyse the AST.

2-IV.A.2 Compiler middle-end

Also known as optimizer, the middle-end refers to all the steps performed between the front-end and the code generation. It is the stage where program analysis and optimizations are performed. Those analyses and optimizations rely on the semantics of the input programming language that allow for understanding and reasoning about how programs behave (more details about formal semantics can be found in the formal semantics section). Analyses are usually functions that scan an AST object to detect errors and collect information (Example: data-flow analysis [Barth, 1978]). Optimizations can be defined using functions that take an AST as input and return an AST (Example: loop transformations [Lebras, 2019]). At the end of the middle-end phase, the program is represented in a form called intermediary representation (IR). That representation serves to simplify the manipulation of programs during their optimization.

2-IV.A.3 Back-end design

The back-end is in charge of the target code generation. It is usually implemented by functions that take an IR as input. This phase can include target code optimizations. Those optimizations are called machine-dependent optimizations. A prominent example is peephole optimizations [McKeeman, 1965], which rewrites short sequences of assembler instructions into more efficient instructions.

2-IV.B Describing the behavior of programs by formal semantics

The purpose of formal semantics is to provide a description of program behaviour. In this section, we will give an overview of the types of formal semantics used for imperative programming languages, as presented in "Sémantiques formelles" [Blazy, 2008].

2-IV.B.1 Operational semantics

Operational semantics is based on a formalized abstract machine. It describes the meaning of a programming language by specifying the effect of its execution on this abstract machine. This description is done by formalizing the transition rules between the abstract machine states. Depending on the level of detail in which transitions are described, there are 2 kinds of operational semantics:

- big-step or natural semantics
- small-step or reduction semantics

Let us illustrate big-step semantics using inference rules.

Expression evaluation:

$$\frac{\sigma \vdash e_1 \mapsto v_1 \qquad \sigma \vdash e_2 \mapsto v_2 \qquad \text{eval\_bin}\ (op, v_1, v_2) = v}{\sigma \vdash e_1\ op\ e_2 \mapsto v}$$

$$\frac{\sigma \vdash e_1 \mapsto v_1 \qquad \text{eval\_un}\ (op, v_1) = v}{\sigma \vdash op\ e_1 \mapsto v}$$

Instruction evaluation:

$$\frac{}{\sigma \vdash \texttt{skip} \Rightarrow \sigma} \qquad \frac{\sigma \vdash e \mapsto v}{\sigma \vdash a := e \Rightarrow \sigma[a \mapsto v]}$$

$$\frac{\sigma \vdash i_1 \Rightarrow \sigma_1 \qquad \sigma_1 \vdash i_2 \Rightarrow \sigma_2}{\sigma \vdash i_1; i_2 \Rightarrow \sigma_2}$$

$$\frac{\sigma \vdash cond \mapsto \texttt{true} \qquad \sigma \vdash i_1 \Rightarrow \sigma'}{\sigma \vdash \texttt{if}\ (cond)\ \texttt{then}\ i_1\ \texttt{else}\ i_2 \Rightarrow \sigma'} \qquad \frac{\sigma \vdash cond \mapsto \texttt{false} \qquad \sigma \vdash i_2 \Rightarrow \sigma'}{\sigma \vdash \texttt{if}\ (cond)\ \texttt{then}\ i_1\ \texttt{else}\ i_2 \Rightarrow \sigma'}$$

$$\frac{\sigma \vdash cond \mapsto \texttt{true} \qquad \sigma \vdash i \Rightarrow \sigma' \qquad \sigma' \vdash \texttt{while}\ (cond)\ i \Rightarrow \sigma''}{\sigma \vdash \texttt{while}\ (cond)\ i \Rightarrow \sigma''} \qquad \frac{\sigma \vdash cond \mapsto \texttt{false}}{\sigma \vdash \texttt{while}\ (cond)\ i \Rightarrow \sigma}$$

Listing 1 *– Big-step semantics inference rules. In this notation, $\sigma \vdash e \mapsto v$ means that the value of the expression e in machine state σ is v. $\sigma \vdash i \Rightarrow \sigma'$ means that the execution of the instruction i in machine state σ transforms it into σ'. $\sigma[a \mapsto v]$ refers to a machine state similar to σ except that the evaluation of a give v. eval_bin and eval_un are functions that compute the result of binary and unary operations*

Let us illustrate small-step semantics using inference rules.

Expression evaluation:

$$\frac{\langle e_1, \sigma\rangle \mapsto_1 \langle e_1', \sigma\rangle}{\langle e_1\ op\ e_2, \sigma\rangle \mapsto_1 \langle e_1'\ op\ e_2, \sigma\rangle} \qquad \frac{\langle e_2, \sigma\rangle \mapsto_1 \langle e_2', \sigma\rangle}{\langle v_1\ op\ e_2, \sigma\rangle \mapsto_1 \langle v_1\ op\ e_2', \sigma\rangle}$$

$$\frac{\text{eval\_bin}\ (op, v_1, v_2) = v}{\langle v_1\ op\ v_2, \sigma\rangle \mapsto_1 \langle v, \sigma\rangle} \qquad \frac{\text{eval\_un}\ (op, v_1) = v}{\langle op\ v_1, \sigma\rangle \mapsto_1 \langle v, \sigma\rangle}$$

Instruction evaluation:

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \Rightarrow_1 \langle \texttt{skip}, \sigma\rangle} \qquad \frac{\langle e, \sigma\rangle \mapsto_* \langle v, \sigma\rangle}{\langle a := e, \sigma\rangle \Rightarrow_1 \langle \texttt{skip}, \sigma[a \mapsto v]\rangle} \qquad \frac{}{\langle \texttt{skip}; i, \sigma\rangle \Rightarrow_1 \langle i, \sigma\rangle} \qquad \frac{\langle i, \sigma\rangle \Rightarrow_1 \langle i', \sigma'\rangle}{\langle i; i_1, \sigma\rangle \Rightarrow_1 \langle i'; i_1, \sigma'\rangle}$$

$$\frac{\langle cond, \sigma\rangle \mapsto_* \langle e, \sigma\rangle}{\langle \texttt{if}\ (cond)\ \texttt{then}\ i_1\ \texttt{else}\ i_2, \sigma\rangle \Rightarrow_1 \langle \texttt{if}\ (e)\ \texttt{then}\ i_1\ \texttt{else}\ i_2, \sigma\rangle} \qquad \frac{\langle cond, \sigma\rangle \mapsto_* \langle e, \sigma\rangle}{\langle \texttt{while}\ (cond)\ i, \sigma\rangle \Rightarrow_1 \langle \texttt{while}\ (e)\ i, \sigma\rangle}$$

$$\frac{}{\langle \texttt{if}\ (\texttt{true})\ \texttt{then}\ i_1\ \texttt{else}\ i_2, \sigma\rangle \Rightarrow_1 \langle i_1, \sigma\rangle} \qquad \frac{}{\langle \texttt{if}\ (\texttt{false})\ \texttt{then}\ i_1\ \texttt{else}\ i_2, \sigma\rangle \Rightarrow_1 \langle i_2, \sigma\rangle}$$

$$\frac{}{\langle \texttt{while}\ (\texttt{true})\ i, \sigma\rangle \Rightarrow_1 \langle i; \texttt{while}\ (\texttt{cond})\ i, \sigma\rangle} \qquad \frac{}{\langle \texttt{while}\ (\texttt{false})\ i, \sigma\rangle \Rightarrow_1 \langle \texttt{skip}, \sigma\rangle}$$

Listing 2 *– Small-step semantics inference rules. In this notation, $\langle e, \sigma\rangle \mapsto_n \langle v, \sigma'\rangle$ means that the simplification (or reduction) of the expression e in machine state σ leads to expression or value v and machine state σ' in n steps. $\langle i, \sigma\rangle \Rightarrow_n \langle i', \sigma'\rangle$ means that the evaluation of the instruction i in machine state σ leads to the instruction i' and machine σ' in n steps. $\Rightarrow_*$ and $\mapsto_*$ means a finite number of repetitions of an operation.*

The main difference between those 2 types is the level of detail in the description. Big step semantics is simpler to understand. There is a proven equivalence between big-step and small-step semantics (for programs that terminates).

2-IV.B.2 Axiomatic semantics

This type of semantics uses sets of logic assertions to describe the behaviour of programs. The evaluation of instructions affects this set by adding or removing properties. It is an approach suited for writing proofs on programs. This model does not focus on a precise view of the memory state but only on properties that can be used for program verification. Let us have an example.

$$\frac{}{\{P\}skip\{P\}} \quad \frac{}{\{P\}a := e\{P \wedge a = e\}}$$

$$\frac{\{P\}i_1\{R\} \quad \{R\}i_2\{Q\}}{\{P\}i_1; i_2\{Q\}}$$

$$\frac{P \Rightarrow P' \quad \{P'\}i\{Q'\} \quad Q' \Rightarrow Q}{\{P\}i\{Q\}}$$

$$\frac{P \Rightarrow cond = true \quad \{P\}i_1\{Q\}}{\{P\}if\ (cond)\ then\ i_1\ else\ i_2\{Q\}} \quad \frac{P \Rightarrow cond = false \quad \{P\}i_2\{Q\}}{\{P\}if\ (cond)\ then\ i_1\ else\ i_2\{Q\}}$$

$$\frac{P \Rightarrow cond = true \quad \{P\}i\{R\} \quad \{R\}while\ (cond)\ i\{Q\}}{\{P\}while\ (cond)\ i\{Q\}} \quad \frac{P \Rightarrow cond = true \quad \{P\}i\{P\}}{\{P\}while\ (cond)\ i\{false\}} \quad \frac{P \Rightarrow cond = false}{\{P\}while\ (cond)\ i\{P\}}$$

Listing 3 *– Axiomatic semantics inference rules. In this notation $\{P\}$ denotes a conjunction of assertions. $\{P\}i\{Q\}$ denotes that if P is valid (conjunction is evaluated to true) set of properties, the execution of the instruction i will lead to a set of properties Q that is valid. $\{false\}$ means inconsistency – for non terminating programs.*

2-IV.B.3 Denotational semantics

Denotational semantics is a technique for defining the meaning of programming languages provided with a mathematical foundation by Dana Scott [Scott, 1982]. At one time called "mathematical semantics," it uses the more abstract mathematical concepts of complete partial orders, continuous functions and least fixed points. Let us give an example. For that we need to consider 2 functions:
A **Expression → Environment → Value** function called ξ (and noted $\xi[\![e]\!]\sigma$) and a **Instruction → Environment → Environment** function called Λ (and noted $\Lambda[\![i]\!]\sigma$). Let us show some rules for this semantics.

Expression evaluation:

$$\text{for every literal value } v \text{ and environment } \sigma, \xi[\![v]\!]\sigma = v$$

$$\forall \sigma\ op\ e,\ \xi[\![e]\!]\sigma = eval\_un(op, \xi[\![e]\!]\sigma)$$

$$\forall \sigma\ op\ e_1\ e_2,\ \xi[\![e_1\ op\ e_2]\!]\sigma = eval\_bin(op, \xi[\![e_1]\!]\sigma, \xi[\![e_2]\!]\sigma)$$

Instruction evaluation:

$$\Lambda[\![\texttt{skip}]\!] = id = \sigma \mapsto \sigma$$

$$\Lambda[\![i_1; i_2]\!] = \Lambda[\![i_1]\!] \circ \Lambda[\![i_2]\!]$$

$$\Lambda[\![\texttt{if (}c\texttt{) then } i_1 \texttt{ else } i_2]\!] = \sigma \mapsto \begin{cases} \Lambda[\![i_1]\!]\sigma & \text{if } \xi[\![c]\!]\sigma \text{ is equal to } true \\ \Lambda[\![i_2]\!]\sigma & \text{if } \xi[\![c]\!]\sigma \text{ is equal to } false \end{cases}$$

$$\Lambda[\![\texttt{while (}c\texttt{) do } i]\!] = fix\ h$$

where

$$h\ t = \texttt{if } c \texttt{ then (}i\texttt{; } t\texttt{) else skip} \quad \text{and} \quad fix\ h = h\ (fix\ h)$$

Listing 4 *– Denotational semantics rules. In this notation, ξ denotes the evaluation of an expression and Λ refers to the evaluation of an instruction. fix is the smallest solution of the fixed point equation [Brown, 1988].*

2-IV.C Formally verified compilers

As software, compilers are computer programs that translate computer code written in one programming language (the source language) into another language such as binary code (ex: gcc [Rothwell and Youngman, 2007]) or other higher level languages (ex: JPT [Coco et al., 2018]). This translation can create bugs. This reason led researchers to work on the formal verification of compilers.

Compcert [Leroy et al., 2016] is a formally verified optimizing compiler for a large subset of the ISO C 1999 language, called CompCert C. Formally verified means that this software provides a formal proof of its correctness by using a tool such as a proof assistant. The backend of Compcert targets the PowerPC [Song et al., 1994] architecture, a common chip for embedded software, ARM [Ryzhyk, 2006], RISC-V [Kane, 1988] and x86 [Owens et al., 2009] (32 and 64 bits) architectures. Its intended use is the compilation of life-critical and mission-critical software written in C and meeting high levels of assurance. Compcert is proven to be exempt from miscompilation issues, that is, the executable code it produces is proved to behave exactly as specified using formal semantics to describe the behaviour of all the 9 intermediary languages it manipulates. The architecture of CompCert can be described as follows:

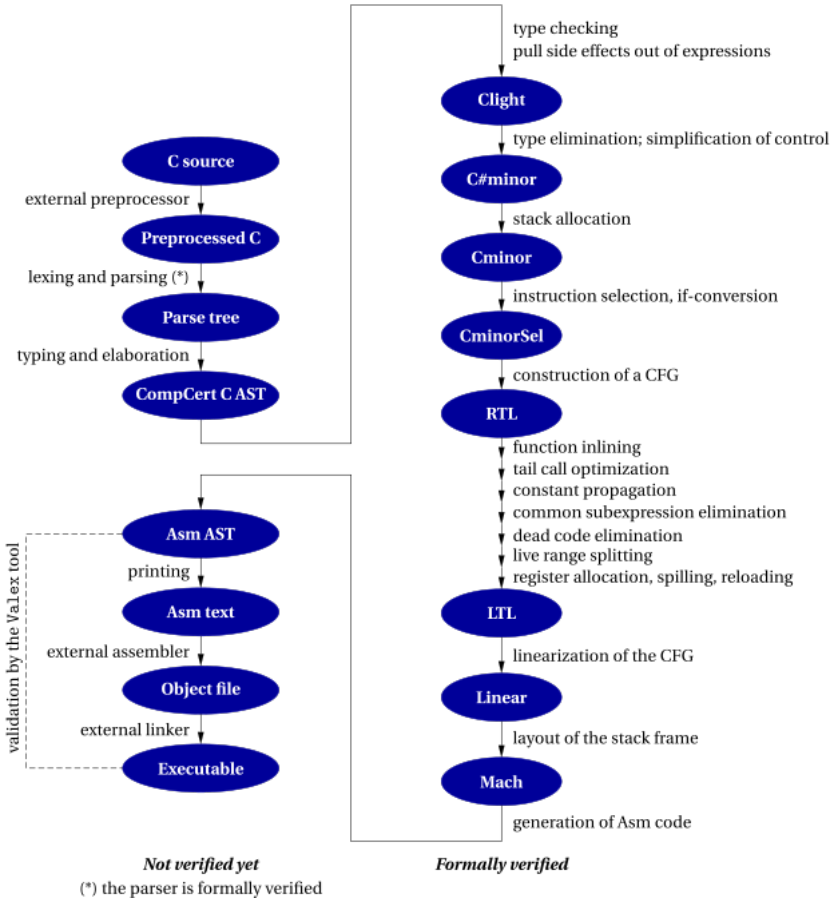


Figure 2.2 *– General structure of the CompCert C compiler*
Source: https://compcert.org/man/manual001.html

Regarding this figure, the CompCert compiler can be decomposed into the following phases:

1. Preprocessing: file inclusion, macro expansion, conditional compilation, etc. Currently performed by invoking an external C preprocessor (not part of the CompCert distribution), which produces preprocessed C source code.

2. lexing and parsing: The lexer is not yet verified but the parsing is formally verified using *menhir* [Régis-Gianas, 2016]
3. type-checking, elaboration, and construction of a CompCert C abstract syntax tree (AST): In this phase, some simplifications to the original C parse tree are performed to better fit the CompCert C language and other mere cleanups such as collapsing multiple declarations of the same variable.
4. Verified compilation proper: From the CompCert C AST, the compiler produces an Asm code, going through 8 intermediate languages and 15 compilation passes. Asm is a language of abstract syntax for assembly language. The 8 intermediate languages bridge the semantic gap between C and assembly, progressively exposing an increasingly machine-like view of the program.
5. Production of textual assembly code, followed by assembling and linking. The last 2 parts are made by external tools (not part of CompCert distribution). Thus, we have no formal guarantees yet, but the Valex tool, available from AbsInt, provides additional assurance via a posteriori validation of the executable produced by those external assembler and linker.

One reason why all the phases of CompCert are not formalized and proved correct is that some of these phases (ex: preprocessing) lack a mathematical specification, making it impossible to state, let alone prove, a correctness theorem about them. Another reason is that the CompCert effort is focused on optimizations, which are all part of the verified phase 4.

All intermediate languages are given formal semantics, and each of the transformation passes is proven to preserve semantics.

The semantic preservation theorem: For all source programs S and compiler-generated code F, if the compiler, applied to the source S, produces the code F without reporting a compile-time error, then the observable behaviour of F is one of the possible observable behaviours of S.

In CompCert, this theorem has been proved, with the help of the Coq proof assistant, taking S to be abstract syntax trees for the CompCert C language (after preprocessing, parsing, type-checking and elaboration), and F to be abstract syntax trees for the assembly-level Asm language (before assembling and linking).

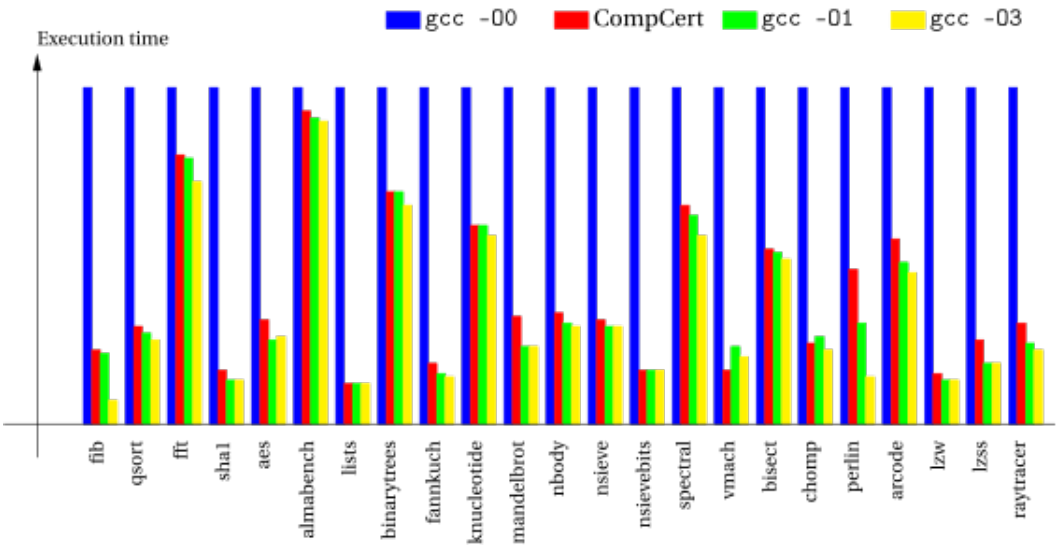


Figure 2.3 *– Performance of CompCert-generated code relative to GCC 4.1.2-generated code on a Power7 processor. Shorter is better. The baseline, in blue, is GCC without optimizations. CompCert is in red.*
Source: https://compcert.org/man/manual001.html

Figure 2.3 shows that the code generated by CompCert runs at least twice as fast as the code generated by GCC without optimizations (gcc -O0), and approximately 10% slower than GCC 4 at optimization level 1 (gcc -O1), 15% and 20% slower at optimization level 3 (gcc -O3). Those results come from a homemade benchmark mix shown in this figure and executed on a Power7 processor.

There exist other formally verified compilers such as CakeML [Kumar et al., 2014], MLCompCert [Dargaye, 2009], a verified compiler for MiniML, and Vélus [Bourke, 2021], a verified compiler for Lustre [Bergerand, 1986]. The last 2 examples are implemented by replacing the CompCert front-end with suitable front-ends for their languages.

Chapter 3

Material and method: the DigiSafe® safety-critical software development process

The challenge of developing safety-critical railway functions in software was met early on with specific technologies. Opened to the public in 1998, **Meteor** [Behm et al., 1999] is a driverless system and the first application of the DigiSafe® software development process. This process ensures that the executable program will run as defined in its specifications. This assurance is based on the error detection system described in Section 2-III. The origin of those errors can be in the source code, the compilers or the processors. The DigiSafe® software development process uses the B-method to build program specifications and generate source code that meets those specifications, and the Vital Coded Processor (VCP) technique associated with the DigiSafe® hardware architecture to detect inconsistencies between the generated source code and the behaviour of the program.

During this thesis, the term safety-critical software will refer to software implemented using the DigiSafe® software development process. In this chapter, we present a version of the implementation of this technique as done by Siemens Mobility. It poses the basis of the technical concepts that had to be taken into account for this research work.

3-I Implementing vital coded processor

The previous chapter presented the Vital Coded Processor theory, a technique to detect inconsistencies between source code and program behaviour. During the implementation of this technique, Siemens engineers wanted to propose transparent management of signatures, meaning that the developer should not manipulate directly the signature values. To do that, the developers implemented an Ada library (VCP-lib) that implements the necessary functions for handling vital variables (signatures), then, to ensure that those functions are used correctly, to forbid the definition and the use of features that are incompatible with the VCP technique, the developers defined a new language: the Vital Coded Processor Ada (VCP Ada), that is a subset of Ada [Barnes, 1984]. In this section, we will present more implementation details. We restricted those details to the required elements needed to understand our research work.

3-I.A VCP Lib

VCP Lib is an Ada file that defines the constructs necessary for making the management of the signatures of variables transparent. This section can be read in parallel with or after Section 2-III.
This library defines 8 basic types:

- `FINT`: that refers to non-vital 32-bit integers.
- `TABLE`: an alias for FINT.

- `FBOOL`: that refers to non-vital booleans coded in 8-bits.
- `SINT`: that refers to vital integers. This data type can be defined using a record containing two fields: the functional value (FINT) of the variable and its control part (see Section 2-III.A on page 20).
- `SBOOL`: that refers to vital booleans.
- `SINT_ARRAY_1`: that refers to a 1D-array of SINT equipped with a control part.
- `SINT_ARRAY_2`: that refers to a 2D-array of SINT equipped with a control part.
- `SBOOL_ARRAY_1`: that refers to a 1D-array of SBOOL equipped with a control part.
- `SBOOL_ARRAY_2`: that refers to a 2D-array of SBOOL equipped with a control part.

Non-vital types are aliases for Ada data-types. For example, *FINT* is an alias for *Integer*. The values of those variables (non-vital values) are not associated with signatures. By convention, non-vital variables are constant and pre-determinable. This means that the values of such variables can be computed at compile-time. Vital types are defined as records that have 2 fields: a *data* field that contains the functional value of the variable and a *sig* field that contains the signature (control part) of the variable. In the current version, the *k* value (see Section 2-III.A on page 20) is equal to 48, that is, signature fields are encoded on 48-bits.

Example: A possible definition for SBOOL can be written as follows

```
1 type Sig is mod A; -- A is a large prime number lower than 2**48
2 type SBOOL is record
3         data : Boolean;
4         sig: Sig;
5 end record;
```

In the current implementation, signatures are computed and stored on the Vital Co-processor, a processor that is different from the one that computes the functional values of variables. As a consequence, the `sig` field contains the address of the signature value in the Vital Co-processor.

Operations on vital variables have to be controlled because their signatures must be correctly set. For that reason, VCP Lib overloads usual operators such as addition, provides procedures to initialize variables and interacts with arrays and hardware. The most important functions and procedures defined in this library are:

- `assign`: is used to assign the value of a variable to another (signature changes).
- `begin_loop`: is used to perform compensations (see the next paragraph or Section 2-III.C) for while instructions. It takes an integer as parameter. This integer is the pre-determinable and unique identifier of the conditional branch in the whole program.
- `converge`: is used to perform compensations for if/else and case instructions. It takes an integer as parameter. This integer is the pre-determinable and unique identifier of the conditional branch in the whole program.
- `diff`: is used to test non-equality for vital variables. It also intervenes in compensations.
- `div2`: is used for right shifting a vital variable.
- `is_valid`: returns true if and only if there is no problem found in the Vital Co-processor.
- `hard_input_valid`: returns true if and only if the last reading on hardware is returned a valid result.

- `end_iteration`: is used to report the end of a while loop iteration to the Vital Co-processor.
- `end_branch`: is used to report the end of an if/else or case branch to the Vital Co-processor. This procedure takes the identifier of that branch (an integer) as parameter.
- `equal`: is used to test equality for vital variables. This function returns a boolean: true for equality and false for non-equality.
- `failure_alarm`: is used to put the system in a safe state (a fail-safe mechanism). In operation, this causes an emergency brake of the train.
- `init`: initializes a vital variable with or without a value. When the variable is an array, every item of the array is initialized with the parameter value.
- `read`: is used to access a vital array.
- `read_hard_input`: is used to read data from an input device.
- `write`: is used to modify an item in a vital array.
- `write_hard_output`: is used to write data on an output device.

VCP Lib also overwrites the usual Ada boolean and arithmetic operators to allow their use with vital variables and values.

For a reminder, compensations are performed as follows:

1. The tracer is updated regarding the result of the branch (or loop) condition (computed using VCP Lib functions such as diff, equal).
2. According to the branch taken or the number of iterations in the loop, compensations have to be performed.

This is the reason why, VCP lib defines functions and operators, to notify the selected branch (`end_branch`) or the number of iterations (`end_iteration`) and to execute the updates of signatures (`converge` for if-case instructions, `begin_loop` for loops). More precisely :

- **end_branch:** All sub-branches of a conditional branch are enumerated (a different positive number for each branch of a given if/case instruction). The end_branch function is in charge of stating the branch taken in order to use the correct constants for convergence. This function call must be the last instruction of any branch instructions. In while loop cases, ends of iterations are notified by the `end_iteration` instruction. `end_iteration` does not take parameters and triggers signature modifications (see Section 2-III.C.2).
- **converge:** A function that is in charge of performing convergence. All the compensations tables of the program are generated by the Signature Predetermination Tool (SPT), described in Section 3-I.C.1, and identified by a unique integer that is associated with a specific if/case instruction. This integer is called table number and must be a pre-determinable constant as parameter. The converge function takes this number as input and performs the convergence. This function call must be the next instruction executed after `end_branch` instruction. `begin_loop` is very similar to converge except that this instruction comes before the loop instruction.
- the use of *continue* or *break* instruction is forbidden. In fact, those instructions can impact the execution flow. Their use could cause an error in the value of the tracer. Their semantics can be simulated using if-else branches and additional variables.
- the use of *exceptions* is forbidden because of their impact on the execution flow. VCP Lib provides a procedure called `failure_alarm` to raise an uncatchable exception in a vital context. This means that this instruction will make sure that the whole system will enter a safety mode. For example, for train autopilot software, this means emergency brake associated with notifications.

The error detection, tracer and dynamic signature management are automatically performed by the Vital Co-processor and triggered by the VCP Lib functions. To make sure that the VCP Lib procedures are correctly used, to prevent the use of forbidden features (use of *exceptions*, *break* and *continue* instructions) and contribute to make signature management more transparent, Siemens researchers suggested creating a programming language VCP Ada that is a subset of Ada.

3-I.B VCP Ada

This section describes the syntax of *VCP Ada*, a subset of *Ada95* that contributes to increase the transparency of VCP signature details for programmers. This description will be done using a variant of *Backus-Naur Form (BNF)* [McCracken and Reilly, 2003].

3-I.B.1 Notation description

The notation we are going to use to describe our language (context-free grammar [Parikh, 1966]) can be summarized as follows:

- <**elem**> refers to a non-terminal symbol whose name is *elem* and can be interpreted as a syntactic category.
- <**elem**>::= **symbs** indicates that *symbs* is the definition of *elem*.
- **Quoted symbols are terminal**. So, any character, including one used in our description notation, can be defined as a terminal symbol of the language being defined.
- {**elem**} indicates a finite repetition (0 to n) of the symbol *elem*.
- [**elem**] indicates that *elem* is optional (0 to 1 repetition).
- **elem**$_1$| **elem**$_2$ indicates alternative between $elem_1$ and $elem_2$
- **elem**$_1$ **elem**$_2$ indicates concatenation of $elem_1$ followed by $elem_2$
- **/*comments*/** indicates comments

3-I.B.2 Accepted identifiers and literals

Identifiers indicate the name of variables, functions and libraries. As in *Ada*, identifiers are case-insensitive. There are forbidden keywords that must be recognized and rejected by VCP Ada: ***abort accept all declare delay do entry exception exit function generic goto new raise return reverse select separate task terminate***. This forbids the declaration and the raise of exceptions, the use of break instruction (exit statement), multi-threading and other features of *Ada* that can be incompatible with this VCP implementation.
Accepted literals are booleans and integers. The real numbers and characters are not implemented in VCP Ada and consequently, their use is forbidden. This is due to the fact that the VCP technique implemented in the Siemens development process is not designed to support real/floating point numbers because it is based on modular arithmetic.

3-I.B.3 Syntax description

3-I.B.3.1 Item names and expressions

Names are used to designate variables, functions or packages but also elements inside record variables or packages (to access package variables or functions).

```
<name>  ::=  <identifier> {‘.‘ <identifier>}
```

In VCP Ada, there are 2 types of variables: vital ones, and non-vital ones. Non-vital variables are variables whose value must be pre-determinable – their values are concrete numbers that can be evaluated at compile-time.

Function and procedure expression parameters can be vital or non-vital values. Their syntax is defined as follows:

```
<expression>       ::=  <relation> {‘and‘ <relation>}
                     |  <relation> {‘or‘ <relation>}
                     |  <relation> {‘xor‘ <relation>}
<relation>         ::=  <simple-ex> [<compare-op> <simple-ex>]
<compare-op>       ::=  ‘=‘
                     |  ‘/=‘
                     |  ‘<‘
                     |  ‘<=‘
                     |  ‘>‘
                     |  ‘>=‘
<simple-ex>        ::=  [‘-‘] <term> {<add-or-minus> <term>}
<term>             ::=  <factor> {<other-binary-op> <factor>}
<factor>           ::=  <primary> [‘**‘ <primary>]
                     |  ‘not‘ <primary>
                     |  ‘abs‘ <primary>
<primary>          ::=  <integer>
                     |  <boolean>
                     |  <name>
                     |  ‘(‘ <expression> ‘)‘
<other-binary-op>  ::=  ‘*‘
                     |  ‘/‘
                     |  ‘mod‘
                     |  ‘rem‘
<add-or-minus>     ::=  ‘+‘
                     |  ‘-‘
```

Examples:
The following function calls are valid:

- proc(5);
- proc(froid and ensoleille or chaud);
- proc(a**(b**c));

The following procedure calls that are valid in Ada are not accepted:

- proc(f(3));
- proc(-4.0 + j);

The first instruction is not valid because of the use of a function call as a procedure argument. The second one is invalid because of the use of real number literal (4.0).

A function expression parameter must either be pre-determinable (for non-vital values) or composed solely of a variable name (for vital values). To make sure that assignments and condition functions from VCP Lib are correctly used, VCP Ada defines the following objects:

```
<lit-or-var>       ::=  <integer>
                     |  <boolean>
                     |  <name>
<vital-var>        ::=  <name>
<vital-logicop>    ::=  ‘and‘
                     |  ‘or‘
<vital-compareop>  ::=  ‘<‘
                     |  ‘<=‘
                     |  ‘>‘
                     |  ‘>=‘
<cond-exp>         ::=  ‘not‘ <vital-var>
                     |  <vital-var> <vital-compareop> <lit-or-var>
                     |  <vital-var> <vital-logicop> <vital-var>
                     |  ‘equal‘ ‘(‘ <vital-var>‘,‘ <lit-or-var> ‘)‘
                     |  ‘diff‘ ‘(‘ <vital-var>‘,‘ <lit-or-var> ‘)‘
<assign-exp>       ::=  ‘-‘ <vital-var>
                     |  <vital-var> ‘+‘ <lit-or-var>
                     |  <vital-var> ‘*‘ <lit-or-var>
                     |  <vital-var> ‘-‘ <lit-or-var>
                     |  <lit-or-var> ‘-‘ <vital-var>
                     |  <cond-exp>
                     |  ‘div2‘ ‘(‘ <vital-var>‘,‘ <expression> ‘)‘
                     |  ‘assign‘ ‘(‘ <lit-or-var> ‘)‘
                     |  ‘read‘ ‘(‘ <vital-var>‘,‘ <vital-var> [‘,‘ <name>] ‘)‘
<while-cond>       ::=  ‘w‘ ‘(‘ <vital-var> ‘)‘
                     |  ‘w‘ ‘(‘ <cond-exp> ‘)‘
<if-cond>          ::=  <cond-exp>
                     |  ‘t‘ ‘(‘ <vital-var> ‘)‘
                     |  ‘hard_input_valid‘
                     |  ‘is_valid‘
```

<vital-var> refers to a name that binds a vital variable (see Section 3-I.A for more details). <assign-exp> refers to assignment expression. <while-cond> and <if-cond> refer to the while and if instruction conditions. Generally, a vital expression must contain at least the name of a vital variable. <div2> refers to right-shift, <read> is used to get an item from a vital array, hard_input_valid and is_valid are functions that allow to check if the last interaction with hardware went correctly. All the operators and functions used in this description are defined or overloaded in VCP Lib.

3-I.B.3.2 Procedure call and instructions

Procedure calls and instructions are defined as follows:

```
<proc-call>  ::=  <user-proc>
                |  <vcp-proc>
<user-proc>  ::=  <name> [‘(‘<params>‘)‘ ]
<params>     ::=  <param> {‘,‘<param> }
<param>      ::=  <expression>
                |  <string>
<vcp-proc>   ::=  ‘begin_loop‘ ‘(‘<expression>‘)‘
                |  ‘converge‘ ‘(‘<expression>‘)‘
                |  ‘init‘ ‘(‘<vital-var> [‘,‘<expression>]‘)‘
                |  ‘failure‘
                |  ‘end_iteration‘
                |  ‘end_branch‘ ‘(‘<expression>‘)‘
                |  ‘write‘ ‘(‘ <vital-var>‘,‘<name>‘,‘<name>‘,‘ [‘,‘<name>] ‘)‘
                |  ‘write_hard_output‘ ‘(‘ <vital-var>‘,‘ <name>‘,‘ <expression>‘,‘ <expression>‘)‘
                |  ‘read_hard_input‘ ‘(‘ <vital-var>‘,‘<name>‘,‘ <expression>‘,‘ <expression> ‘)‘
<inst>       ::=  ‘null‘ ‘;‘
                |  <vital-var> ‘:=‘ <assign-exp> ‘;‘
                |  ‘if‘ <if-cond> ‘then‘ <insts> {<else-if>} ‘else‘ <insts> ‘end‘ ‘if‘ ‘;‘
                |  ‘case‘ ‘c‘ ‘(‘<vital-var>‘)‘ {<case-v>} ‘when‘ ‘others‘ ‘=>‘ <insts> ‘end‘ ‘case‘ ‘;‘
                |  ‘while‘ <while-cond> ‘loop‘ <insts> ‘end‘ ‘loop‘ ‘;‘
                |  <proc-call> ‘;‘
<else-if>    ::=  ‘elsif‘ <if-cond> ‘then‘ <insts>
<case-v>     ::=  ‘when‘ <simple-ex> ‘=>‘ <insts>
<insts>      ::=  <inst> {<inst>}
```

The `write` procedure is used to modify an item of 1D and 2D arrays. `write_hard_output` and `read_hard_input` are used to make interactions with other devices. The `init` procedure is used to initialize vital variables. The `failure` procedure is used to signal a problem: in that case, the system must enter safe mode. More technical details about their semantics can be found in Chapter 5. All the functions defined by the `<vcp-proc>` non-terminal symbol such as `end_branch` or `converge` are procedures defined in VCP lib. Strings are only usable as function parameters.

In *if-else*, the *else* part is mandatory. Similarly, the *default* part is mandatory for *case* instruction. By convention, the values of `<case-v>` (the value after *when* keyword) cover all the possible values of the *case* parameter (the variable used *c* function call after *case* keyword). As a consequence, the *default* branch must never be taken during the program execution. If this branch is taken, then it means an error is detected. In that branch, the developer can manually report an error by calling the *failure* procedure. If no error is reported, then the program enters safety mode and reports an *out-of-code* error at the end of the *default* branch.

As already mentioned in Section 3-I.A, there are additional rules about `end_branch` and `converge` procedure calls:

- `end_branch` must be the last instruction of each of the branches of an if/case instruction. This procedure takes an argument that is the identifier of the branch in the instruction. This argument's value must be pre-determinable and start from 0 for the first branch to n-1 for the last branch. The `end_iteration` procedure must be the last instruction of a while instruction.
- `converge` must be the first instruction executed after a call to an `end_branch` instruction. This procedure also takes an argument that is the identifier of the conditional branch instruction (if-else, case) in the whole program. This identifier must be unique. `begin_loop` is similar to `converge` but it must be the last instruction before a `while` instruction. Arguments used in `converge` and `begin_loop` must be pre-determinable and unique. This uniqueness allows to identify the compensation table to use.

Example:

```
if t(a) then
  a := assign(b);
  end_branch(0);
else
  begin_loop(0);
  while w( c.d < b ) loop
    proc(c.d);
    c.d := c.d + 1;
    end_iteration;
  end loop;
  a := assign(d);
  end_branch(1);
end if;
converge(1)
b := assign(e);
```

```
if t(a) then
  a := assign(b);
else
  while w( c.d < b ) loop
    proc(c.d);
    c.d := c.d + 1;
  end loop;
  a := assign(d);
end if;
b := assign(e);
```

Figure 3.1 *– Accepted vs. rejected code*

Remarks:

- The restriction about the uniqueness is also valid for function calls. For example, if a function contains a converge instruction with a constant parameter like `converge(10)`, a call to this function must not appear in two different lines of code. In fact, during the execution this would cause the use of the same compensation table for two different positions, causing an error in signature management.
- From the point of view of a compiler, instructions are seen as sequences of procedure calls that are either defined in VCP Lib or defined by developers. As most of those procedures are written in different files, most compilers are not good at optimizing such programs because they do not implement the semantics of VCP Lib procedures and their optimization phase comes after the linking phase, the phase that allows to have access to the definition of those procedures.

3-I.B.3.3 Variables and procedures

In VCP Ada, only constant variables must be defined at declaration. Only records and subtypes can be declared by the user. Type and variable declarations can be described as follows:

```
<id>               ::=  <identifier>
<var-decl>         ::=  <const-decl>
                     |  <non-const-decl>
<const-decl>       ::=  <name> ‘:‘ ‘constant‘ <name> ‘:=‘ <expression> ‘;‘
<non-const-decl>   ::=  <name> ‘:‘ <type-designator> ‘;‘
<record-decl>      ::=  ‘type‘ <id> ‘is‘ ‘record‘ <rec-comp> ‘end‘ ‘record‘ ‘;‘
<rec-comp>         ::=  <id> ‘:‘ <name> ‘;‘ {<id> ‘:‘ <name> ‘;‘}
<subtype-decl>     ::=  ‘subtype‘ <id> ‘is‘ <type-designator> ‘;‘
<type-designator>  ::=  <name>
                     |  <name> ‘(‘ <expression> ‘)‘                          1D array
                     |  <name> ‘(‘ <expression> ‘,‘ <expression> ‘)‘         2D array
```

In VCP Ada, users can only define procedures (subprograms without return values). To modify values in procedures, developers can use parameter modes as defined in Ada. *renames* keyword is used only to rename procedures. The syntax used to declare and define procedures is defined as follows:

```
<proc-renm>  ::=  ‘procedure‘ <id> [‘(‘<args-desc>‘)‘] ‘renames‘ <name> ‘;‘
<arg-desc>   ::=  <id> ‘:‘ [mode] <name>
<args-desc>  ::=  <arg-desc> {‘,‘ <arg-desc>}
<proc-decl>  ::=  ‘procedure‘ <id> [‘(‘<args-desc>‘)‘] ‘;‘
<proc-def>   ::=  ‘procedure‘ <id> [‘(‘<args-desc>‘)‘] ‘is‘ {<var-decl>} ‘begin‘ <insts> ‘end‘ <id> ‘;‘
<mode>       ::=  ‘in‘
              |   ‘in‘ ‘out‘
              |   ‘out‘
```

3-I.B.4 Compilation unit

The syntax of a VCP Ada file is defined as follows:

```
<comp>            ::=  {<with-statement>} <unit>                                          AST
<with-statement>  ::=  ‘with‘ <id> {‘,‘ <id>}‘;‘
<unit>            ::=  <proc-def>                                                         entrypoint
                   |   ‘package‘ <id> ‘is‘ {<decl-d>} ‘end‘ <id> ‘;‘                      header
                   |   ‘package‘ ‘body‘ <id> ‘is‘ { <b-1> } { <b-2> } ‘end‘ <id> ‘;‘      implementation
<decl-d>          ::=  <var-decl>
                   |   <proc-decl>
                   |   <record-decl>
                   |   <subtype-decl>
                   |   <proc-renm>
<b-1>             ::=  <var-decl>
                   |   <record-decl>
                   |   <subtype-decl>
<b-2>             ::=  <proc-decl>
                   |   <proc-renm>
                   |   <proc-def>
```

3-I.C Compiling and executing VCP Ada

This section defines the DigiSafe® XME advanced vital computer, a set of software and hardware that implements the VCP technique.

3-I.C.1 Compilation

The last section defined the syntax of VCP Ada using a context free grammar. However, this grammar was not sufficient to control all the rules of the VCP Ada language. For example, this grammar does not control if a variable is pre-determinable or not or if there exist different calls to the `converge` procedure with the same parameter (see Section 3-I.A). Performing such verifications is under the responsibility of the Signature Predetermination Tool (SPT). In addition to that, the SPT is in charge of computing and storing the compensation and all the other necessary data to perform the error detection handled of the VCP technique in a file called a *trace*.

After the SPT processing, the VCP Ada code is compiled by an Ada compiler, GNAT [Schonberg and Banner, 1994], to produce an object file that the linker will associate with the trace to build an executable.

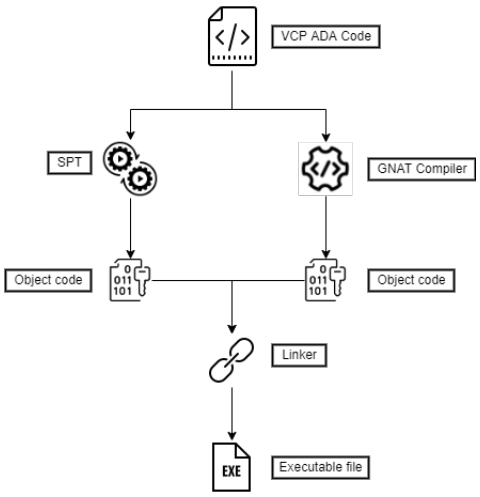


Figure 3.2 *– VCP Compilation*

3-I.C.2 Execution

The execution of VCP Ada safety-critical software is done using 2 processors: a standard processor that is in charge of computing the functional values of variables and a calculation co-processor (the vital co-processor) that handles the computation of control parts. In addition to that, there is a dynamic controller that is in charge of result checks. During the program execution, the standard processor and the vital co-processor send a trace of their execution (instruction code, parameter addresses, calculation results and signature) to a dynamic controller that performs the runtime signature checks (comparing the received results to the expected one, computed by the SPT) and raises *out-of-code* error.

This compilation and execution organisation makes the production system non-dependent on compiler and linker errors.

3-I.C.3 Safety guarantees and Certification

Those tools and architecture allow the detection of various kinds of errors: incorrect execution flows (incorrectly incremented program counter, error of not entering a branch/loop or exiting it prematurely, error of (not) calling a function), incorrect arithmetical or logical calculations (incorrect operand, incorrect operator), error of functional and signature values being corrupted in RAM. During the program execution, the system relies on the vital co-processor to compute signatures and on a dynamic controller that will compare them to those computed offline by the SPT. It also relies on the VCP Ada language to restrict program instructions to arithmetical and logical calculation, branch, loop and function call instructions.

The absence of detection means that the computed signatures are equal to those computed by the SPT and that the program instructions have been executed in the right order. It also means that the right calculation instruction is sent to the processor. All this ensures that the result is very probably correct.

Siemens researchers have proven that the probability P_{abs} of not detecting an error is linked to the size of the coding k (as A, the large prime chosen randomly is lower than 2^k): $P_{abs} = 1/2^k = 1/2^{48} = 3.55 \times 10^{-15}$. The combination of the failure rate of each piece of equipment leads to a probability of failure lower than 10^{-9} per hour for the whole system. This safety level achievement is better than the target of 10^{-8} dangerous failures per hour, the level required for SIL4 systems.

The latest version of the DigiSafe® XME advanced vital computer is SIL4 certified by TUV [Gall, 2004], independent service companies from Germany and Austria that test, inspect and certify technical systems, facilities and objects of all kinds in order to minimize hazards and prevent damage.

The validation approach for DigiSafe® XME advanced vital computer is a different one from conventional approaches, as safety is independent of the hardware used (microprocessor, memories, trans-

mission media, bus...), of the computer system platform (real-time monitor), the production system (compiler, link editor). The DigiSafe® XME computer is validated by simply analyzing its specifications and the VHDL implementation of the co-processor and dynamic controller tasks. As vital processing functions are isolated in specific hardware (dynamic controller, coprocessor), computer validation is restricted to the analysis of this equipment.

3-II Filling the gaps in the development process

The previous section details the implementation of the VCP technique. This technique allows to detect incoherences between the VCP Ada source code and its execution. However, the VCP technique cannot detect design errors caused by the source code. This is the reason that motivated the use of the B-method [Cansell and Méry, 2003]. As a reminder, the B method allows to write and verify (by using a proof assistant) formal specifications of programs and then, by successive verified refinements, provide verified implementations of those programs. This implementation is proven to conform to the initial specification and is written in B0 notation, a notation that is close to usual imperative language notations.

At this point, the mentioned technologies ensure:

- the coherence of a specification and between a specification and its implementation written in B0
- the coherence between VCP Ada code and its execution

The remaining problem is to ensure the correctness of the transformation from B0 code to VCP Ada code. To overcome this problem, Siemens engineers applied the *double system principle* also called *N version programming* or *design diversity*: 2 B0-to-VCP Ada translators have been developed fully redundantly (teams, design and programming language were different) on a common specification.

Here are some details of the specification of those translators:

- For any B implementation file, generated 2 Ada files:

  Ada header (.ads): that contains variables, constants and procedure declarations

  Ada implementation (.adb): that contains variables and procedure definitions

- Constant variables are defined in the Ada header.
- Non-constant variables are vital variables.
- Operations in B0 are transformed into procedures with similar parameter types and modes – `in` (passing by copy) for operation input or `in out` (passing by reference) for operation output.
- Instruction such as `end_branch`, `end_iteration`, `converge` and `begin_loop` are added automatically (during the B0-to-VCP Ada translation) to respect the VCP Ada syntax.

With this double system, the whole development process can be described as follows:

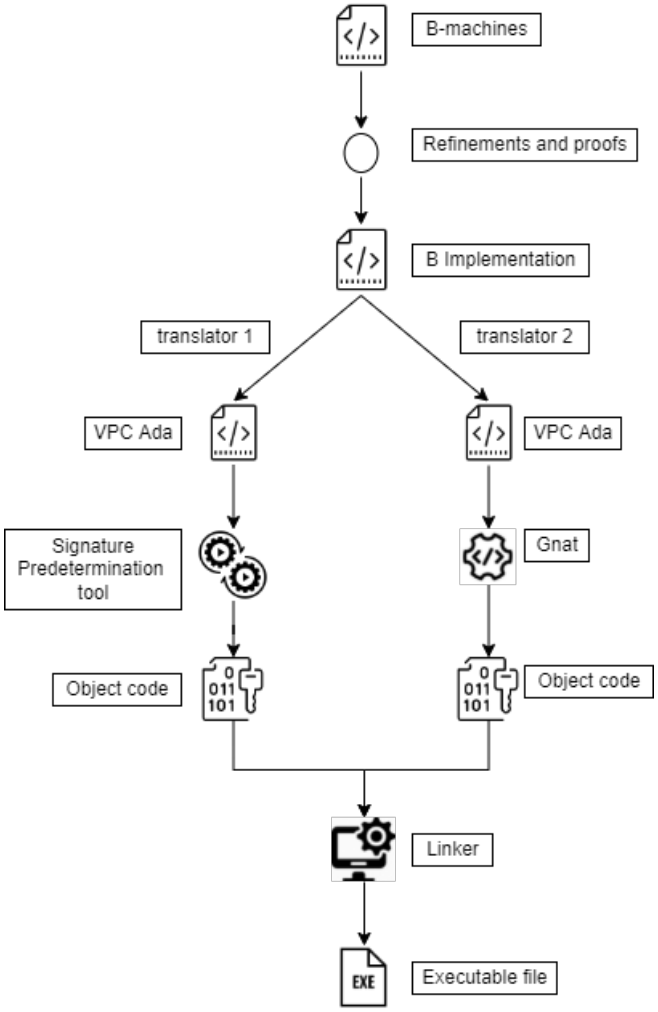


Figure 3.3 *– DigiSafe® Development process*

As the 2 translators are supposed to generate similar codes, they are interchangeable. If one of them produces a different piece of code, the trace generated by the SPT and the execution values will be different and lead to an *out-of-code* error. This development and execution process allows to produce SIL4 software.

3-III Conclusion

This chapter described the Siemens development process allowing to produce the programs that this thesis aims to optimize. Regarding that process, there can be various possibilities to use for such optimizations. The next chapter focuses on the study of those possibilities and justifies the solution we chose.

Chapter 4

Exploring the optimization possibilities

In the previous chapter, we presented a detailed definition of the safety-critical programs we will study and their development process. As a reminder, this development process starts with specifications – B-machines – of the program to implement. Those abstract specifications are successively refined into more concrete descriptions of the program called B-implementations. Those refinements are proven to be correct regarding the initial specifications thanks to formal proofs. The program is then translated into *VCP Ada*, a subset of *Ada*, that implements the vital coded processor principles. This code is then verified and compiled into executables. Optimizations can be accomplished at different levels (source code, intermediate representation) and by different parties (developer, compiler, optimizer). In this chapter, we will present the main levels, types and scopes of optimizations, and justify why optimizing *VCP Ada* code is the best optimization option.

4-I Levels of optimization

In this section, we reuse the abstraction levels identified by Falk and Marwedel [Falk and Marwedel, 2004] and their conclusion that massive improvements are achievable when considering a very high level of abstraction for the application of optimization techniques. It describes 4 abstraction levels of code optimization (from high to low):

1. Description or design optimization: At this level, optimizations are focused on design choices. The design may be optimized to make better use of the available resources, given goals, constraints, and expected use/load. The choices made in this part of the development have the most significant impacts on the final system performance. The research(ers) showed that optimizations at this level could lead to executing up to 20 times faster. This kind of optimization is done by the system or software top-level architect designer.

2. Algorithm and data type selection: For a given design, there may exist many algorithms and data structures that allow to implement a system that executes. Choosing a more efficient algorithm could reduce considerably the execution time.Those choices are made by the developer.

3. Source code optimization: Beyond general algorithms and their implementation, concrete source code level choices can make a significant difference. For example, on early C compilers, while(1) was slower than for(;;) for an unconditional loop, because while(1) evaluated 1 and then had a conditional jump which tested if it was true, while for (;;) had an unconditional jump. Some optimizations (such as this one) can nowadays be performed by optimizing compilers.

4. Compiler optimization: This kind of optimization refers to those integrated into today's state-of-the-art compilers. The usual ones are function inlining, tail call optimization, register allocation, dead code and common sub-expression elimination. These optimizations can be divided into processor-specific and processor-independent techniques as explained in more detail in the following section.

As found by the authors, or just following common sense, performing optimizations at higher levels allows to have greater impacts on the final performances. In the next section, we will present the main constraints from the point of view of this study and determine the levels of optimization that can be performed.

4-II Presenting the possibilities

The scope of this research is to optimize programs generated using the DigiSafe® development process. Those programs can be executed on different hardware and implement different algorithms and are used for widely different purposes. For instance, some programs are used for the way-side equipment or the landing doors, while others are used to control the speed onboard the train. The techniques used to perform optimizations at design or algorithm level can differ a lot from one program to another and requires human actions. To perform optimizations automatically (without requiring human action) and to find techniques that can be used to optimize the various programs implemented using the DigiSafe® development process, we will work on lower levels (level 3 and 4) and suggest a modification of the development process.

However, the DigiSafe® development process is certified SIL4, that is it ensures that hazardous events are made "impossible". As a reminder, this toolchain can be divided into 6 types regarding their features (from higher to lower level):

- B-method components: that verify the consistency of B elements (machines, refinements, implementations and proofs),
- B0 to VCP Ada translators: that aim to convert B implementations into VCP Ada code,
- Signature Predetermination tool: that reads VCP Ada code to extract and generate compensation tables and signatures necessary for VCP computations and store them in object code,
- GNAT compiler: that compiles VCP Ada code into object code,
- linker: that transforms all the generated object codes into an executable file.

Modifying this development process could create a vulnerability in the generated software. This means that any change must be proposed with correctness proof ensuring that the change will not create a safety issue. For that reason, optimization is restricted to the addition or replacement of components by formally verified optimizing software in the current toolchain.

Replacing the SPT will not have any impact on the execution time as this tool just computes expected signatures offline. Replacing the 2 translators with a unique formally verified optimizing translator requires modifying the technical specifications of B0 to VCP Ada translation and solving problems that are similar to solutions based on B implementation optimization and those based on VCP Ada source code optimization. Generally, we found out that it was simpler, regarding the proof and testing efforts, to focus our attention on optimizers whose input language is identical to their output language.

In the following subsections, we will explore the optimization possibilities.

4-II.A Optimizing programs by changing B models

As a reminder, B codes are divided into 3 categories: B machines, B refinements, and B implementations. To those types of files, we can also add B proofs which are the formal proofs of machine-refinement, refinement-refinement, and refinement-implementation consistency. Optimizing B code would come with the following questions:

1. When optimizing B machines, how can we prove that 2 machines, the non-optimized one and the optimized one, are equivalent?

2. When optimizing B refinements and implementations, how can we automatically generate consistency proofs for the optimized B-refinements? Can software modify the consistency proofs of the non-optimized refinements to obtain consistency proofs for the optimized ones?

3. How difficult would it be for developers to maintain those proof reparations or generation tools?

During our exploration, we noticed that the equivalence between B machines can be implemented and proven by applying the technique used in the CompCert project: describing formal semantics and using it to prove semantic equivalence (see chapter 2 for more details). The main problem is that a change in a machine can invalidate the current proofs, refinements and implementation. So, trying this solution requires propagating the changes performed in the machines or refinements to all the objects (refinements, implementations and proofs) that depend on them. Modifications in implementation files imply a modification of their consistency proofs. Works related to proof repair [Ringer, 2021, Ringer et al., 2018] have been done in formal proof assistants such as Coq, but nothing has been done for B models.

4-II.B VCP Ada code optimization

As a reminder, *VCP Ada* is a subset of Ada that is designed by Siemens engineers for Vital Coded Processor programming. It is the second language used in the DigiSafe® development process and the one that is compiled to generate an executable. Consequently, this code has a direct impact on the execution time. Optimizing VCP Ada code would come with the following questions:

1. Are there effective optimizations – not already performed by the compiler?

2. How can we prove the correctness of those optimizations?

In a nutshell, performing formally verified optimizations of *VCP Ada* code refers to building a formally verified *VCP Ada* to *VCP Ada* compiler. The CompCert experience provides materials for this kind of work. The main problem would be to find effective optimizations, as the B-to-Ada transcoder and the compiler are already designed to perform some.

4-II.C Compiler optimization

Compiler optimization consists in changing the optimization flags in the current compiler or using a more efficient one. The main problem is that VCP Ada is implemented to make vital coded computation transparent for the developer. This has an impact on the source code where most instructions are external function calls. So, except optimizations that are relative to constant values such as constant propagation and constant expression simplification, the compiler has not enough visibility to make efficient optimizations. In addition, some optimizations such as those that change the order of instructions can cause inconsistencies between the behaviour of the compiled program and the signatures generated by the SPT.

However, it is possible to replace GNAT with CompCert. To do that, we can write a VCP Ada front-end for CompCert. Similar projects such as MLCompCert [Dargaye, 2009], a verified compiler for MiniML have been implemented. The *sparkformal* project [Courtieu et al., 2013] on the formal verification of Ada and Spark [Barnes, 2012] provides a formally verified semantics that allows reading the abstract syntax tree produced by the GNAT compiler frontend and checking that the equivalent generated abstract syntax tree (AST) in Coq was well-formed and had the desired run-time checks. Building a formally verified function that transforms this Coq AST into data structures that CompCert can process would lead to a formally verified compiler that can be used to replace GNAT.

Furthermore, the current development process does not require a verified compiler that protects from compiler errors. However, a formally verified compiler ensures that no error due to miscompilation will be detected.

4-II.D Comparing the different possibilities

According to the levels of optimization description, higher optimization levels have more impact on the execution time of executable programs. So, by order of impact on the execution time, B model modifications come first, then VCP Ada optimization and finally compiler optimization. However, only B implementations have a direct impact on the execution time – these codes are the ones translated to VCP Ada – but their modification still requires proof repairs. VCP Ada presents similar opportunities except that it allows more manipulations relative to VCP Lib function semantics.

In addition, performing optimizations on VCP Ada source code provides other advantages:

- files can be treated stand-alone: Ada code is modular, separated into semantically significant units. Consequently, we do not have to propagate optimizations to other files.
- correctness proofs have to be done only for the optimizers.
- semantics preservation relies on unique formal semantics instead of using one semantics for the source and another semantics for the target language (and for all the intermediate languages). For instance, the Compcert project implemented 8 formal semantics for each of the 10 intermediary languages it manipulates.
- the impact of an optimization can be easily evaluated: benchmark on a source code where optimizations are made by hand
- optimizations are human-readable and human-understandable

Compiler optimizations appear to be less efficient that source code optimization regarding its compilation level and the visibility limit due to VCP Lib implementation. So, VCP Ada source code optimization appears to be more strategic because of its abstraction level and its implementation difficulty.

4-III Conclusion

This chapter studied the optimization possibilities regarding the Siemens development process. It also showed that VCP Ada source code optimization appears to be the most strategic option for programs implemented using that Siemens development process. As this optimization must conserve the safety guarantees that the Siemens development process provides, we decided to implement a VCP Ada to VCP Ada formally verified compiler. According to the research on compiler design [Wilhelm and Maurer, 1995], this implementation implies:

1. A lexer implementation to generate a token stream from the input source code
2. A parser implementation to generate an abstract syntax tree (AST) of the source code
3. Optimizers to perform changes on the AST
4. A pretty printer to transform the AST into VCP Ada source file

There exist techniques and tools to formally verify the parser and optimizers. At the beginning of our research, no tool was implemented to verify lexers. In the next chapter, we will present our approach to secure lexers and how to combine them with formally verified parsers to build verified front-ends. Chapter 6 presents the formal semantics of VCP Ada and the semantic preservation definition that is used to prove the correctness of the middle-end described in Chapter 7. The pretty printer is presented in Chapter 8.

Chapter 5

Securing the front-end

A compiler consists of a sequence of phases going from lexical analysis to code generation. Its formal verification includes the formal verification of every component of the tool-chain. This chapter focuses on the front-end that is made up of 2 tools: a lexer and a parser. This chapter first presents our contribution to lexer formal verification and how we combined it with **menhir**, a verified parser generator [Régis-Gianas, 2016], to build a verified compiler front-end.

5-I Generating formally verified lexers

Lexing is one of the first components in compilers and other language processing tools. Combined with parsers, lexers whose goal is to perform lexical analysis, help to transform input texts into structured data, usually abstract syntax trees. The theoretical foundations of lexing are quite mature today and there exist sophisticated tools, libraries and generators to help in implementing efficient lexers.

The issue this section focuses on is the *correctness of lexical analysis*: how can we prove that lexer behavior meets its specifications? How can we have a formal guarantee that a lexer generator generates a lexer that respects the input specification?

This section aims to contribute to the end-to-end verification of compilers by presenting an evolution of the implementation of a realistic formally verified lexer generator called CoqLex [Ouedraogo et al., 2021]. This lexer generator, based on an existing Coq implementation of Brzozowski derivatives [Brzozowski, 1964], is intended to be user-friendly with usage similar to ocamllex [Smith, 2007].

5-I.A Background

Lexical analysis, lexing, or tokenization is the process of converting a sequence of characters into a sequence of tokens (strings with an assigned and thus identified meaning). A program that performs lexical analysis may be termed a lexer, tokenizer, or scanner, although scanner is also a term for the first stage of a lexer. This tokenizer is usually the first tool of compiler tool-chains and combined with parsers, it helps to analyze the syntax of programming languages, web pages, and so forth. Most generators such as lex/flex [Levine et al., 1992] and ocamllex produce lexers from a specification essentially comprised of lexical rules that are sets of regular expressions [Thompson, 1968, Karttunen et al., 1996] also called regexp or regex with associated semantic actions.

In this section, we define and formalize regexps made from Brzozowski derivatives, lexical rules and their interpretation.

5-I.A.1 Regular expression

Regular expressions [Thompson, 1968, Karttunen et al., 1996] (shortened as regex or regexp) refer to search patterns that can recognize a set of string characters called a language. They appeared with

the formalization of the description of a regular language [Yu, 1997] and are used to provide lexer specifications.

Given an alphabet (set of symbols or characters) Σ, the symbol ϵ that refers to the empty string, the operator $+\!\!+$ that refers to string concatenation, the notation s^n (with $n \in \mathbb{N}$) that refers to the concatenation of n copies of the string s and the notation $L(r)$ that refers to the language described by the regexp r, we can provide an inductive definition of regexp constructions as described in figure 5.1.

Using all the notations above, we say that a regular expression r matches a string s if $s \in L(r)$. Similarly, when $s \notin L(r)$ we say that r does not match s.

$regex ::=$	
$\varnothing_r$	The empty regexp
	$L(\varnothing_r) = \varnothing$
$\mid \epsilon_r$	The empty string regexp
	$L(\epsilon_r) = \{\epsilon\}$
$\mid [[a]]$	The one-symbol regexp ($a \in \Sigma$)
	$L([[a]]) = \{a\}$
$\mid e_1 + e_2$	The alternative
	$L(e_1 + e_2) = L(e_1) \cup L(e_1)$
$\mid e_1 \cdot e_2$	The concatenation
	$L(e_1 \cdot e_2) = \{p_1 +\!\!+ p_2 \mid p_1 \in L(e_1) \wedge p_2 \in L(e_2)\}$
$\mid e^*$	The Kleene star
	$L(e^*) = \bigcup_{n\in\mathbb{N}} L(e)^n$

Figure 5.1 *– Definition of regular expressions associated with the language they describe. Variables e e_1 and e_2 are regular expressions. The symbol $\{a\}$ denotes the set containing a unique string that is made up of a unique symbol which is a. The symbol $\emptyset$ denotes the empty set and the operation $\cup$ denotes the set union.*

5-I.A.2 The longest match and the priority principle

In most lexer generators, lexers are specified using a set of rules. Each rule is composed of two elements: a regex and a semantic action. During lexical analysis, the specified lexer has to choose the semantic action to perform. This choice (the *election*) is made using the longest match and the priority rules.

The longest match principle uses the concepts of string prefix and score.

prefix: A string p is said to be a prefix of a string s if and only if there exists a string s' such that $s = p +\!\!+ s'$. For example ϵ is a prefix of any string.

score: Given a regexp r, a string s and a natural number n, we say that the score of r on s is n (we note $\mathbb{S}(r, s) = n$) if and only if the length of longest prefix of s that r can match is n. For example, the score of $[[a]]*$ in 'aabaaaa' is 2 as the longest prefix of 'aabaaaa' that $[[a]]*$ can match is 'aa' whose length is 2. The score of $[[a]]*$ in 'cba' is 0 because $[[a]]*$ can match ϵ, which is a prefix whose length is 0. There exist cases where there is no score (e.g: $\mathbb{S}([[a]], `bac`)$). In that case, we note $\mathbb{S}(r, s) = -\infty$.

In a nutshell, the longest match principle says that, given an ordered list of lexical rules l_r and a string s, the semantic action to perform is the one associated with the regex that has the highest score on s. There exist situations where more than one lexical rule can have the highest score. In this case, the priority rule applies. This rule says that in those cases, the semantic action to perform is the one that comes first in l_r between those that have the highest score. There exist cases where there is no score for the input string (e.g when no prefix of the input string can be matched by the regexps of the lexer specification rules). In those cases, the lexical analysis cannot analyse the input text.

Ocamllex also provides the shortest match principle that is the opposite of the longest match rule: this principle selects the semantic action that is associated with the first regex that matches the shortest

prefix of the input string. Ocamllex also defines a special rule called the `EOF` rule that is elected regardless of the election principle (longest or shortest match) when the input text is empty.

Coqlex provides constructions for the `EOF` rule, the longest and shortest match principle.

5-I.B Representing a lexer in Coq

From a functional point of view, lexers are in charge of producing tokens (a user-defined type that we will note `T`) from a text (`string`). A natural type would be

```
lexer(T) := string -> list T
```

Instead of processing a list of tokens, most parsers like ocamlyacc [Smith, 2007] process tokens one by one, and call lexers. A function that performs *one step* of lexical analysis (lexing) consumes a string and returns a token and the remaining string. The type of such a function is

```
lex1(T) := string -> T * string
```

Lexing can fail for various reasons. In case of failure, lexers should provide useful error messages. For that reason, we defined a position data type and an error data type to encapsulate the lexing result. A function that performs one step of lexing becomes a function that takes an input string, a start position and in case of success, returns a token, the remaining string and the end position. Consequently, the type of one step of lexing becomes

```
lex1(T):=  string ->
           position ->
           Result(T * string * position)
```

In addition to returning a token, users commonly write lexers that manipulate some internal state. A typical use case would be the lexing of nested structures such as OCaml comments.[1] So, one step of lexing becomes a function that takes a string, a position, a storage (a user-defined type that we will note `S`) and returns a lexing result associated with a storage.

```
lex1(T, S):=  string ->
              position ->
              S ->
              Result(T * string * position) * S
```

Most lexer generators generate lexers using a set of lexical rules that are regular expressions [Thompson, 1968, Karttunen et al., 1996] associated with semantic actions that are in charge of producing the lexing result. The semantic action that will produce the returned lexing result is the first one associated with the regexp that matches the longest prefix of the input string (the lexeme): this is the longest match and the priority rules. Semantic actions have access to the lexing buffer (lexbuf), a data structure containing the lexeme, the start position (the position of the first letter of the lexeme), the end position (the position of the letter after the last letter of the lexeme) and the remaining string (the input string without the lexeme). The semantic action also specifies how the internal state of the lexer (at type `S`) should be updated. So, a natural type for semantic actions would be:

```
action(T, S):=  lexbuf ->
                S ->
                Result(T * string * position) * S
```

Those semantic actions can perform various operations, including recursive calls to the lexer that calls them. This could then lead to an infinite loop.[2] As Coq forbids the implementation of functions that

[1] The OCaml grammar accepts nested, well-bracketed comments (* ... (* ... *) ... *), and the OCaml lexer maintains a stack to correctly process them as a single token.

[2] Section 5-I.F.1 provides a typical OCamllex example of a lexer that can loop due to recursive calls.

loop [Chlipala, 2013], Coqlex had to find a solution to deal with those kinds of situations. We explored two possibilities:

1. Making restrictions on semantic actions that ensure terminations. For example, we could require that each semantic action discards at least one character from the input string.

2. Using the fuel technique: this technique consists in ensuring the termination of lexers using a natural number (`nat`) that decreases at every recursive call.

Requiring that each semantic action discards at least one input character is too strict in practice. Studying lexers in the wild, we have found many cases of lexers designed to "skip" an optional part of the input, that accept the empty string if nothing needs to be skipped. For example, the lexer of the OCaml compiler contains the following lexer:

```
rule skip_hash_bang = parse
  | "#!" [^ '\n']* '\n' { new_line lexbuf }
  | "" { () } (* accepts the empty string *)
```

We thus chose to express general, potentially non-terminating lexers using fuel. Consequently, the type of one step of lexing becomes

```
lex1(T, S):= nat ->
             string ->
             position ->
             S ->
             Result(T * string * position) * S
```

To make it simple to call a lexer from a semantic action, we replace the separate arguments `string` and `position` by the more informative `lexbuf` type already used by semantic actions.

```
  lex1(T, S):= nat ->
               action(T,S)

action(T, S):= lexbuf ->
               S ->
               Result(T * lexbuf) * S
```

5-I.C Coqlex in practice

Coqlex comes with a Coq library that allows to write lexers using sets of lexical rules. It also provides a text processor that will convert a markup language (.vl syntax), that is similar to the OCamllex [Smith, 2007] specification language (.mll syntax), into its equivalent Coq code (.v file). Figure 5.2 presents the .vl version of the mini-cal (a micro language for arithmetic expressions : numbers, idents, + * - / and parentheses) lexer. This .vl definition has four parts:

1. The header section: The header section is arbitrary Coq text enclosed in curly braces. If present, the header text is copied as it is at the beginning of the output file. Typically, the header section contains the Coq `Require Import` directives, possibly some auxiliary functions and token definitions used for lexer definitions.

2. The regexp definition section: This section allows to give names to frequently-occurring regular expressions. This is done using the syntax `let` $ident$ `=` re to associate the name $ident$ to the regexp re. The syntax of regexp is defined in Figure 5.3.

3. The lexer definition section: This section allows to define lexers using sets of rules. A rule is defined using the syntax `|` p `{`a`}` (the '|' symbol is not mandatory for the first rule) to associate the pattern p to the Coq text representing a semantic action a. This pattern is either a regexp or a `string -> bool` function (defined using the syntax `$(`f`)` where f is the Coq code of this function). Typically, this kind of pattern is used to detect situations in which the lexing must stop (e.g when the input string is empty). When the pattern is a regexp, the semantic rule is said to be *regexp based*. Otherwise, the semantic rule is said to be *function based* (example: `eof`).

4. The trailer section: This section is similar to the header section, except that its text is copied as it is at the end of the output file. Typically, this section contains Coq extraction directives.

```
(* header section *)
{
Require Import TokenDefinition.
}

(* regexp definitions *)
let ident = ['a'-'z']+
let numb = ['0'-'9']+

(* lexer definitions*)
rule minlexer = parse
| '\n' { sequence [new_line; minlexer] }
| ident {ret_l ID}
| numb { ret_l Number }
| '+' { ret PLUS }
| '-' { ret MINUS }
| '*' { ret TIMES }
| '(' { ret LPAREN }
| ')' { ret RPAREN }
| eof { ret Eof }
| _ { raise_l "unknown token :"}

(* trailer section *)
{}
```

Figure 5.2 – *mini-cal.vl file*

$re ::=$	
$'c'$	Character constant
$\mid "string"$	String constant
$\mid \_$	Char wildcard
$\mid [s_1 s_2 ... s_n]$	Union of character sets
$\mid [\hat{}s_1 s_2 ... s_n]$	Union of negation of character sets
$\mid re_1 \mid re_2$	Alternative
$\mid re_1\ re_2$	Concatenation
$\mid re_1 - re_2$	Difference
$\mid re*$	Kleene star
$\mid re+$	Strict repetition
$\mid re?$	Option
$s ::=$	
$'c'$	Character constant
$\mid 'c_1'$ - $'c_2'$	Character range

Figure 5.3 *– Syntax of Coqlex regexps*

Remarks:

- A .vl file allows to define multiple lexers. Those lexers are gathered in groups (made of mutually recursive lexers) using the keyword *and*. To define non-mutually recursive lexers, the user must use the keyword *then* instead.
- The command `%polymorphic_storage` in Figure 5.2 specifies that the storage type of `minlexer` is parametric. Developers can replace that command by `%fix_storage_type` s `;` to set the storage type to s. Similarly, developers can use the command `%polymorphic_token` or `%fix_token_type` t `;` to specify the token type of `minlexer`. When not specified, Coqlex lets the Coq system infer the types of tokens and storage.
- *Coqlex generator* users do not need to worry about the management of fuel when writing .vl files.
- For each lexer defined in the .vl file, the *Coqlex generator* produces a lexer function with same name.

Using the code in Figure 5.2, the *Coqlex generator* outputs the Coq code in Figure 5.4. Typically, the generator translates the regexp written in .vl syntax into the Coqlex regexp data type. The generated lexing function also calls Coqlex functions such as `generalizing_elector`, `longest_match_elector` and `exec_sem_action`. It also recurses over the fuel explicitly; we could instead generate a call to a fixpoint combinator, but this would be difficult to scale to mutually-recursive lexers.

Similarly to OCamllex, Coqlex also allows to choose the semantic action by matching the shortest prefix. In that case, the function `longest_match_elector` is replaced by the function `shortest_match_elector`.

```
Require Import TokenDefinition.
Definition ident := Cat ((CharRange  "a"%char   "z"%char ))
        (Star ((CharRange  "a"%char   "z"%char ))).

Definition numb := Cat ((CharRange  "0"%char   "9"%char ))
        (Star ((CharRange  "0"%char   "9"%char ))).

Fixpoint minlexer {Storage: Set} fuel lexbuf storage
{struct fuel} := match fuel with
| O => (AnalysisNoFuel lexbuf, storage)
| S n => (match generalizing_elector
    (Action := semantic_action (Storage := Storage))
     LexerDefinition.longest_match_elector (
     [(Char  "010"%char ,  sequence [new_line; (minlexer n)]);
      (ident, ret_l ID);
      (numb,  ret_l Number );
      (Char  "+"%char ,  ret PLUS );
      (Char  "-"%char ,  ret MINUS );
      (Char  "*"%char ,  ret TIMES );
      (Char  "("%char ,  ret LPAREN );
      (Char  ")"%char ,  ret RPAREN );
      (RValues.regex_any,  raise_l "unknown token : ")],
      [(CoqlexUtils.EOF,  ret Eof )]) (remaining_str lexbuf) with
    | Some elt => exec_sem_action elt lexbuf storage
    | None => (AnalysisFailedEmptyToken lexbuf, storage)
    end)
end.
```

Figure 5.4 – *mini-cal.v*

5-I.D *Coqlex generator* specification

Given an ordered list of regexp-based rules l_r, an ordered list of function based rules l_f, and a matching policy e, the generated Coq code implements a lexer — a function that takes a fuel n_f, lexbuf b, a storage s and returns a lexing result — that works as follows:

1. If n_f is equal to 0, then the result is an error. This error is a direct consequence of the fuel technique.
2. Otherwise, from the input string, l_r, l_f and e, the lexer chooses a rule whose semantic action will be in charge of returning the lexing result.
   (a) if the selected rule is a function-based one, made up with a function f associated with a semantic action a, then there is no consumption. Consequently, the lexing result is the result of a called with b and s.
   (b) if the selected rule is a regexp-based rule c – made up with a regexp r associated with a semantic action a – and if the length of the prefix matched by r using the policy e is a natural number n, then the lexing result is the result of a applied with the updated lexbuf b_u and the input storage s. The updated lexbuf is defined as follows:
      - the lexeme of b_u is the n first characters of the input string.
      - the remaining string of b_u is the input string without the lexeme.
      - the end position of $b_u$is the end position of b where the column number is incremented by n.
      - the start position of b_u is the end position of b.

(c) if no rule is selected, the lexer must return an error meaning that the input string contains elements that cannot be analysed by the lexer.

Except for the use of fuel, the functioning of the generated lexer defined above is standard (similar to the functioning of the lexers generated using the usual lexer generators such as OCamllex).

A .vl file provides the description of lexers using lexical rules. That description is processed by the *Coqlex lexer generator* whose architecture is detailed in Figure 5.5. It has three components:

1. The lexer, that is in charge of generating a set of tokens from the text of the .vl file, is written in Coq using the Coqlex library and is formally verified.

2. The parser, that is in charge of generating an abstract representation from the set of token produced by the lexer, is implemented using menhir [Régis-Gianas, 2016] with `-coq` switch to generate verified parsers.

3. The code printer, that is in charge of generating the .v file from the abstract representation produced by the parser, is written in OCaml and is not formally verified.

The code printer does not include formal semantics equivalence between the representation of the .vl code and the generated .v code. This means that, *a priori*, a critical user should review the generated .v code. This does not take great efforts because the transformation does not include a complex compilation process: the .vl and .v files have similar structures and are human readable.

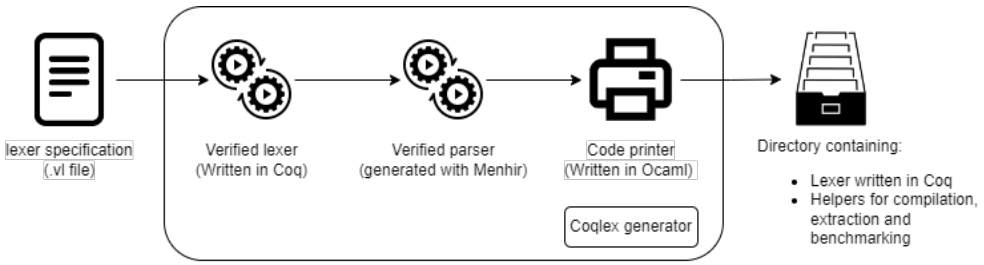


Figure 5.5 – *General structure of the* Coqlex lexer generator.

Comparing to other existing approaches, Verbatim++ [Egolf et al., 2022] does not provide such generation tool, and OCamllex generates an OCaml code in which the patterns of the lexical rules are compiled into a non deterministic automaton [Becchi and Crowley, 2013] represented by a compact table of transitions, making the generated code non human readable.

In our case, the regexps, rule selection and associated policies used in the generated file are implemented and proved correct in Coq. Consequently, the attention of the critical user who wants to check the generated .v file must be focused on the following elements:

- The translation of regexps: The user must be assured of the correspondence between the regexps written in the input .vl files and those generated in the output.v files. This requires reading and understanding the regexps constructors that will be defined in Section 5-I.E.

- The matching policy: The user has to make sure that matching policy corresponds to the one that is described in the .vl file. The keyword `parse` must correspond to `longest_match_elector` and `shortest` must correspond to `shortest_match_elector`.

- For every lexer, the user must be assured that the right regexps are associated with the right semantic actions and in the same order. In the Coq code, a difference is made between lexical rules made up with regexps associated with semantic actions (regexp-based rules) and those made up with `string -> bool` functions associated with semantic actions (function-based rules).

$$
\begin{aligned}
&\texttt{nullable}\ \varnothing_r = \texttt{false}\\
&\texttt{nullable}\ \epsilon_r = \texttt{true}\\
&\texttt{nullable}\ [\![a]\!] = \texttt{false}\\
&\texttt{nullable}\ (e_1 + e_2) = \texttt{nullable}\ e_1 \vee \texttt{nullable}\ e_2\\
&\texttt{nullable}\ (e_1 \cdot e_2) = \texttt{nullable}\ e_1 \wedge \texttt{nullable}\ e_2\\
&\texttt{nullable}\ e^* = \texttt{true}
\end{aligned}
$$

Figure 5.6 *– Definition of the* `nullable` *function. The variable a stands for a symbol and variables e, e_1 and e_2 for regular expressions.*

In a nutshell, a potential user has to (i) review the Coq implementation and verification of regexps, the rule selection together with the associated policies and helpers, that are written and proved in Coq, once; and (ii) either review the code printer of the *Coqlex generator* once, or review the elements listed above at every generation.

5-I.E Coqlex implementation details

Most lexer generators such as OCamllex speed up lexical analysis by compiling lexical rules into finite automata during lexer generation. In Coqlex, lexical rules are interpreted on the fly, using Brzozowski derivatives [Brzozowski, 1964] for regexps and simple functions for matching policies.

5-I.E.1 Brzozowski derivatives for regexps matching

Coqlex uses regexp constructions and matching algorithms based on the concept of Brzozowski derivatives. This concept introduces two functions: the `nullable` function and the derivative of a regexp.

The `nullable` function takes a regexp r and returns the boolean `true` if r matches ϵ (the empty string) and `false` otherwise. Its inductive definition is given in Figure 5.6.

Using the notation az to denote the string built from the symbol a as first element and the string z, the derivative of a regular expression r by a symbol a is the regexp r/a that denotes the language $\{z | az \in L(r)\}$. Its inductive definition is given in Figure 5.7.

$$
\begin{aligned}
\varnothing_r / c &= \varnothing_r\\
\epsilon_r / c &= \varnothing_r\\
[\![a]\!] / c &= \begin{cases} \epsilon \text{ if } a = c \\ \varnothing_r \text{ otherwise} \end{cases}\\
(e_1 + e_2)/c &= (e_1/c) + (e_2/c)\\
(e_1 \cdot e_2)/c &= \begin{cases} (e_1/c \cdot e_2) + e_2/c \text{ if } \texttt{nullable}\ e_1 = \texttt{true} \\ (e_1/c \cdot e_2) \text{ otherwise} \end{cases}\\
e^*/c &= (e/c) \cdot e^*
\end{aligned}
$$

Figure 5.7 *– Definition of the derivative of a regexp. The variables a and c stand for symbols and variables e, e_1 and e_2 for regular expressions.*

$$
\begin{aligned}
r \| \epsilon &= r\\
r \| az &= (r/a) \| z
\end{aligned}
$$

Figure 5.8 *– Extension of the derivative of a regexp to strings. Variables r, ϵ, a and z denote, respectively, a regex, the empty string, a symbol and a string. The operator / refers to the derivative operation described in Figure 5.7. The notation az denotes the string composed of the symbol a as first element and the string z.*

Brzozowski [Brzozowski, 1964] extended the derivative operation to strings (denoted by $\|$) as described in Figure 5.8, and showed that for every regular expression r and every string s

$$s \in L(r) \iff \texttt{nullable}\ (r\|s) = \texttt{true}$$

Coqlex uses an existing Coq implementation of Brzozowski derivatives for regexp matching. That implementation provides a Coq proof showing that this Brzozowski derivative implementation is a Kleene algebra [Kozen, 1997, Armstrong et al., 2013] and defines an equivalence relation ($\equiv$) for regexps whose formal definition is $e_0 \equiv e_1 \iff L(e_0) = L(e_1)$. It also provides additional regex constructors such as the conjunction and negation constructors that are not used in the regexp constructors that are provided by the *Coqlex generator* (see Figure 5.3). On the other hand, some of the constructions of regexps presented in Figure 5.3 are missing. For this reason, we modified the existing Coq implementation [Miyamoto, 2011] of Brzozowski derivatives as follows:

1. We removed the conjunction and negation regexp constructors
2. We added four regexp constructors:
   - **the char wildcard:** The notation ω_r denotes a regexp that matches any 1-length-string. This regexp is defined by the following two properties: $\texttt{nullable}\ \omega_r\ =\ \texttt{false}$ and for all symbol c, $\omega_r / c = \epsilon$. Then, we proved that for all strings s, we have $s \in L(\omega_r)$ if and only if s consists of a single character.
   - **the character set:** The notation Σ_l^u (where l and u are symbols) denotes a regexp whose language is $L(\Sigma_l^u) = \{c | l \leq c \wedge c \leq u \wedge c \in \mathbb{A}\}$ (where $\leq$ is a reflexive, anti-symmetric and transitive order relation on symbols). This constructor is defined using the following two properties: $\texttt{nullable}\ \Sigma_l^u\ =\ \texttt{false}$ and for all symbol c

     $$\Sigma_l^u / c = \begin{cases} \epsilon_r \text{ if } l \leq c \wedge c \leq u \\ \varnothing_r \text{ otherwise} \end{cases}$$

     We proved that if $\neg(l \leq u)$, then $\Sigma_l^u \equiv \varnothing_r$ and that for every string s, $s \in L(\Sigma_l^u)$ if and only if s consists of only one symbol c such that $l \leq c \wedge c \leq u$.
   - **the negation of character set:** The notation $\overline{\Sigma_l^u}$ (where l and u are symbols) denotes a regexp whose language is $L(\overline{\Sigma_l^u}) = \{c | \neg(l \leq c \wedge c \leq u) \wedge c \in \mathbb{A}\}$. This constructor is defined using the following two properties: $\texttt{nullable}\ \overline{\Sigma_l^u}\ =\ \texttt{false}$ and for all symbol c

     $$\overline{\Sigma_l^u} / c = \begin{cases} \epsilon_r \text{ if } \neg(l \leq c \wedge c \leq u) \\ \varnothing_r \text{ otherwise} \end{cases}$$

     We proved that if $\neg(l \leq u)$, then $\overline{\Sigma_l^u} \equiv \omega_r$ and that for every string s, $s \in L(\overline{\Sigma_l^u})$ if and only if s consists of only one symbol c such that $\neg(l \leq c \wedge c \leq u)$.
   - **the difference:** The notation $e_1 - e_2$ (where e_1 and e_2 are regexps) denotes a regexp whose language is $L(e_1 - e_2) = \{s | s \in L(e_1) \wedge s \notin L(e_2)\}$. This construction is defined using the following two properties: $\texttt{nullable}\ e_1 - e_2\ =\ (\texttt{nullable}\ e_1)\ \wedge \neg(\texttt{nullable}\ e_2)$ and for all symbol c, $(e_1 - e_2)/c = e_1/c - e_2/c$. We proved that for all strings s, we have $s \in L(e_1 - e_2) \iff s \in L(e_1) \wedge s \notin L(e_2)$.

These constructors have also been added for performance reasons. In fact, the regexp $\Sigma_{c_n}^{c_{n+m}}$ could be written as $[[c_n]] + [[c_{n+1}]] + ... + [[c_{n+m}]]$. However, using the first representation ($\Sigma_{c_n}^{c_{n+m}}$), the derivation function will perform 2 comparisons, while the second one will perform $m+1$ comparisons (see Figure 5.7).

5-I.E.2 Matching policies

Coqlex defines two types of rules: the function based and the regexp based ones. During the lexical analysis, the generated lexer has to select a rule. This selection starts by the choice of a function based rule (noted E_f). This function based rule selection, whose formal definition is given in Figure 5.9, consists of finding the first rule that is made of a function whose application with the input string returns true.

$$\frac{}{E_f([],s) = \bot}$$

$$\frac{f\ s = \texttt{true}}{E_f((f,a)::t,s) = (f,a)} \qquad \frac{f\ s = \texttt{false}}{E_f((f,a)::t,s) = E_f(t,s)}$$

Figure 5.9 *– The formal description of the selection of a function based-rule. This description uses the list notation: [] denotes the empty list and $h :: t$ denotes a list whose first element is h and whose tail is t. The symbol $\bot$ means that no rule is selected.*

If no such function based rule is found, then lexer has to choose a regexp based rule.

Most lexers perform regexp based rule election using a longest match selection policy based on the longest match and priority rules. That selection policy allows to select the first lexical rule whose regexp matches the longest prefix of the input string.

The longest match (and priority) rule of Coqlex starts by the implementation of *l-score* that is the function in charge of computing the score for the longest match rule. The inductive definition of that function noted $\mathbb{S}_l$ is given in Figure 5.10.

$$\frac{\texttt{nullable}\ r\ \texttt{=}\ \texttt{true}}{\mathbb{S}_l(r,\epsilon) = 0} \qquad \frac{\texttt{nullable}\ r\ \texttt{=}\ \texttt{false}}{\mathbb{S}_l(r,\epsilon) = -\infty}$$

$$\frac{\mathbb{S}_l(r/a,z) = n}{\mathbb{S}_l(r,az) = n+1} \qquad \frac{\mathbb{S}_l(r/a,z) = -\infty \quad \texttt{nullable}\ r\ \texttt{=}\ \texttt{true}}{\mathbb{S}_l(r,az) = 0}$$

$$\frac{\mathbb{S}_l(r/a,z) = -\infty \quad \texttt{nullable}\ r\ \texttt{=}\ \texttt{false}}{\mathbb{S}_l(r,az) = -\infty}$$

Figure 5.10 *– The formal description of* l-score *computation.*

To prove the correctness of $\mathbb{S}_l$, we used the Coq substring function of Coq string module [Thery, 2020] to define the prefix. This function takes two natural numbers n m and a string s and returns the substring of length m of s that starts at position n denoted by $\delta_n^m(s)$. Here, the position of the first character is 0. If n is greater than the length $|s|$ of s then ϵ is returned. If $m > (|s| - n)$, then $\delta_n^m(s) = \delta_n^{|s|-n}(s)$. Consequently, if $m \leq |s|$, then $\delta_0^m(s)$ is the prefix of length m of s. For all strings s and regexps r, we provided Coq proofs of the following theorems:

1. if there exists a natural number n such that $\mathbb{S}_l(r,s) = n$, then $n \leq |s|$. This helps to make sure that n can be used with δ to extract the prefix of length n.

2. if there exists a natural number n such that $\mathbb{S}_l(r,s) = n$, then $\delta_0^n \in L(r)$. This means that the input regexp matches the prefix of length n of s.

3. if there exists a natural number n such that $\mathbb{S}_l(r,s) = n$, then for all m such that $n < m \leq |s|$, $\delta_0^m(s) \notin L(r)$. This means that *l-score* is maximal. Therefore, there exists no prefix of length higher than n that r can match.

4. $\mathbb{S}_l(r,s) = -\infty$ if and only if for all natural number m, $\delta_0^m(s) \notin L(r)$.

Properties 1, 2 and 3 show that $\mathbb{S}_l$ is correct. This means that if a score is returned, this score is the length of the longest prefix of the input string that the input regexp can match. Property 4 shows the completeness and the soundness of $\mathbb{S}_l$. This means that if no score is found, then there is no score, and if there exists a score, $\mathbb{S}_l$ will return it.

Using $\mathbb{S}_l$, the longest match policy (noted E_l) consists in choosing the regexp based rule whose regexp has the highest *l-score*. The Coqlex formal definition of that policy is defined in Figure 5.11.

$$\frac{}{E_l([], s) = \bot} \qquad \frac{\mathbb{S}_l(r, s) = -\infty}{E_l((r, a) :: t, s) = E_l(t, s)}$$

$$\frac{\mathbb{S}_l(r, s) = n \qquad E_l(t, s) = (r_t, a_t, n_t) \qquad n_t > n}{E_l((r, a) :: t, s) = (r_t, a_t, n_t)}$$

$$\frac{\mathbb{S}_l(r, s) = n \qquad E_l(t, s) = \bot}{E_l((r, a) :: t, s) = (r, a, n)}$$

$$\frac{\mathbb{S}_l(r, s) = n \qquad E_l(t, s) = (r_t, a_t, n_t) \qquad n_t \leq n}{E_l((r, a) :: t, s) = (r, a, n)}$$

Figure 5.11 *– The formal description of the longest match selection policy. The symbol $\bot$ means that no rule is selected.*

To prove the correctness of E_l, we proved with Coq that for every string s and list l_r of regexp based rules:

1. if $E_l(l_r, s) = \bot$ then for every regexp r and semantic action a such that $(r, a) \in l_r$, $\mathbb{S}_l(r, s) = -\infty$
2. if there exists a regexp r, a semantic action a and a natural number n such that $E_l(l_r, s) = (r, a, n)$ then for every regexp r', semantic action a' and natural number n' such that $(r', a') \in l_r$ and $\mathbb{S}_l(r', s) = n'$, $n' \leq n$
3. for every regexps r and r', semantic actions a and a' and natural number n, if $E_l(l_r, s) = (r, a, n)$ and $\mathbb{S}_l(r', s) = n$ then $E_l((r', a') :: l_r, s) = (r', a', n)$

Besides the longest match policy, Coqlex defines the shortest match policy that allows to select the first regexp based rules whose regexp matches the shortest prefix of the input string. The implementation technique of the shortest match policy is similar to the longest match policy. This implementation starts by the definition of the **s-score** (noted: $\mathbb{S}_s$) that allows to compute the length of the shortest prefix that a regexp can match. The formal definition of **s-score** is described in Figure 5.12.

$$\frac{\texttt{nullable } r \texttt{ = true}}{\mathbb{S}_s(r, s) = 0} \qquad \frac{\texttt{nullable } r \texttt{ = false}}{\mathbb{S}_s(r, \epsilon) = \infty}$$

$$\frac{\mathbb{S}_s(r/a, z) = n \qquad \texttt{nullable } r \texttt{ = false}}{\mathbb{S}_s(r, az) = n + 1}$$

$$\frac{\mathbb{S}_s(r/a, z) = \infty \qquad \texttt{nullable } r \texttt{ = false}}{\mathbb{S}_s(r, az) = \infty}$$

Figure 5.12 *– The formal description of* s-score *computation.*

Similarly to the $\mathbb{S}_l$, we proved the correctness and completeness of $\mathbb{S}_s$ through Coq proofs of the following theorems:

1. if there exists a natural number n such that $\mathbb{S}_s(r,s) = n$, then $n \leq |s|$. This helps to make sure that n can be used with δ to extract the prefix of length n.
2. if there exists a natural number n such that $\mathbb{S}_s(r,s) = n$, then $\delta_0^n \in L(r)$. This means that the input regexp matches the prefix of length n of s.
3. if there exists a natural number n such that $\mathbb{S}_s(r,s) = n$, then for all m such that $m < n$, $\delta_0^m(s) \notin L(r)$. This means that *s-score* is minimal. Therefore, there exists no prefix of length lower than n that r can match.
4. $\mathbb{S}_s(r,s) = \infty$ if and only if for all natural number m, $\delta_0^m(s) \notin L(r)$.

Using $\mathbb{S}_s$, the shortest match policy consists into choosing the regexp based rule whose regexp has the lowest *s-score*. The Coqlex formal definition that policy is defined in Figure 5.13.

$$\frac{}{E_s([],s) = \bot} \qquad \frac{\mathbb{S}_s(r,s) = \infty}{E_s((r,a) :: t, s) = E_s(t,s)}$$

$$\frac{\mathbb{S}_s(r,s) = n \qquad E_s(t,s) = (r_t, a_t, n_t) \qquad n_t < n}{E_s((r,a) :: t, s) = (r_t, a_t, n_t)}$$

$$\frac{\mathbb{S}_s(r,s) = n \qquad E_s(t,s) = \bot}{E_s((r,a) :: t, s) = (r,a,n)}$$

$$\frac{\mathbb{S}_s(r,s) = n \qquad E_s(t,s) = (r_t, a_t, n_t) \qquad n_t \geq n}{E_s((r,a) :: t, s) = (r,a,n)}$$

Figure 5.13 *– The formal description of the shortest match selection policy.*

5-I.E.3 Coqlex rule selection

Using E_f, E_l and E_s, the formal definition of the rule selection E can be defined as follows:

$$E(E', l_r, l_f, s) = \begin{cases} E_f(l_f, s) \text{ if } E_f(l_f, s) \neq \bot \\ E'(l_r, s) \text{ otherwise} \end{cases}$$

where E' is either E_l or E_s. In the Coq code presented in Figure 5.4, E is represented by `generalizing_elector`, E_l is represented by `longest_match_elector`. The implementation of E_s in the Coqlex library is represented by `shortest_match_elector`.

5-I.E.4 Optimization

The naive implementation suggested by the formal definition of $\mathbb{S}_l$ and E_l has a time complexity that is at least quadratic in the size of the input string. In fact, the implementation of *l-score* requires reading all the characters of the input string for every regexp based lexical rule and thus for every token. However, this is not necessary in some cases (e.g for all string s with $\mathbb{S}_l(\varnothing_r, s) = -\infty$).

To increase the performance of $\mathbb{S}_l$, $\mathbb{S}_s$, E_l and E_s, we implemented a regexp simplification function which is based on the following properties:

The alternative: $r + \varnothing_r \equiv r$ and $\varnothing_r + r \equiv r$

The concatenation: $r \cdot \varnothing_r \equiv \varnothing_r$, $\varnothing_r \cdot r \equiv \varnothing_r$, $r \cdot \epsilon_r \equiv r$, $r^* \cdot r^* \equiv r^*$ and $\epsilon_r \cdot r \equiv r$

The Kleene star: $\varnothing_r^* \equiv \epsilon_r$, $(r^*)^* \equiv r^*$ and $\epsilon_r^* \equiv \epsilon_r$

The difference: $r - \varnothing_r \equiv r$ and $\varnothing_r - r \equiv \varnothing_r$

These simplifications aim to detect when a given regexp is equivalent to a regexp whose score is trivial (e.g $\varnothing_r$ or ϵ_r). We proved these properties in Coq and then used the smart constructor technique [HaskellWiki, 2020] to write an optimized version of the regexp derivative function. That function works similarly to the original one, except that it returns a simplified version of the derivative. Then, we also rewrote the **s-score** and **l-score** functions to use the optimized version of the regexp derivative function and return the result for trivial cases. For example, we proved that for every r and s,

$$\mathbb{S}_s(r^*, s) = \mathbb{S}_s(\epsilon_r, s) = \mathbb{S}_l(\epsilon_r, s) = 0$$

$$\mathbb{S}_s(\varnothing_r, s) = \infty \qquad \mathbb{S}_l(\varnothing_r, s) = -\infty$$

We proved that the optimized **s-score** and **l-score** are equal to the original ones. We propagated this optimization to E_l, E_s and obtained performance in linear time in the size of the input string (see Section 5-I.F below).

5-I.F Evaluation

We are now going to compare Coqlex with OCamllex, the OCaml standard lexer generator, and Verbatim++, the only other well-documented and complete research work on lexer verification that we are aware of.

First, we noticed three conceptual differences between Verbatim++, Coqlex and OCamllex:

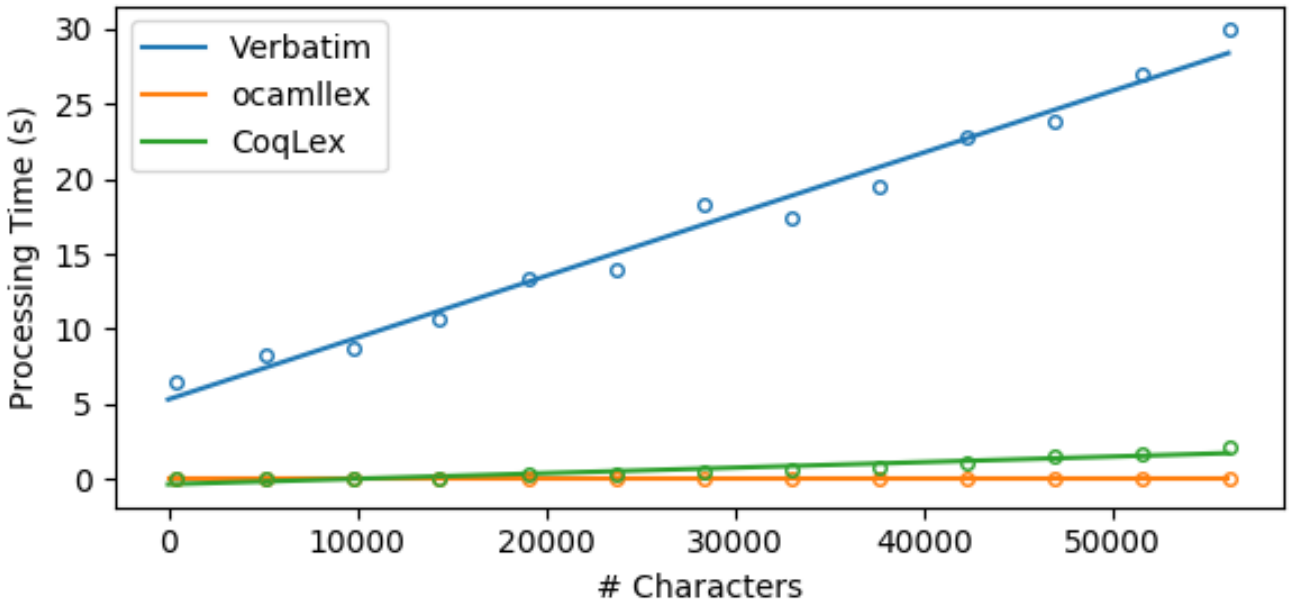
1. Verbatim++ uses the notion of label, a data-type returned after the election. Semantic actions are functions that take that label and lexeme to return a token. Therefore, semantic actions do not have access to the remaining string and thus, cannot perform recursive calls. A consequence of this is that Verbatim++ lexers cannot ignore parts of the input string (such as comments and extra spaces).

2. Verbatim++ lexers do not propose error handling features while Coqlex and OCamllex have one.

3. In Coqlex, regexps are interpreted on the fly while Verbatim++ and OCamllex compile them into finite automata [Becchi and Crowley, 2013] for fast regexps matching. In Verbatim++, this regexp compilation is made at every lexical analysis while in OCamllex, this compilation is made once and its result stored in the form of a compacted table that is read during lexical analysis.

Second, we evaluated the execution time of the generated lexical analysers in two phases. For the first phase, we evaluated their performance on the Verbatim++ JSON benchmark.

We started with analysing a JSON lexer implemented by the Verbatim++ developers using Verbatim++ Coq source code, then we used Coqlex and OCamllex generators to generate lexers with similar specifications. We compared the time performance and noticed a huge difference between the OCamllex and Coqlex generated lexers and the Verbatim++ lexer as presented in Figure 5.14.



	Verbatim++	Coqlex	OCamllex
Tokens per sec.	1.5×10^3	2.2×10^4	2.8×10^7
Time to process 50ko.	30 s	2 s	1.8×10^{-3} s

Figure 5.14 *– Comparison of execution time in seconds for Coqlex, OCamllex and Verbatim++ lexers on Verbatim++ JSON benchmark. The benchmark file contains 56154 characters and its lexical analysis should return 17424 tokens.*

The analysis of this Figure shows linear performance for all three lexers. Generally, lexers implemented using Verbatim++ components are around 15 times slower than those generated using Coqlex and OCamllex. A similar study with XML files showed equivalent results.

For the second phase of the evaluation of the execution time, we evaluated the performance of Coqlex and OCamllex generated lexers by implementing the lexer of 2 languages: JSON [Pezoa et al., 2016, Sikora, 2017] and the first version of MiniML [Rinderknecht, 2018], a toy subset of OCaml. We could not perform an evaluation of those languages with Verbatim++ because their definitions imply recursive calls, a feature that is not handled by Verbatim++. The results of those evaluations are presented in Figure 5.15.

	Coqlex	OCamllex
Time to process 1.6Mo MiniML	0.45 s	0.04 s
Time to process 1.6Mo JSON	1.5 s	0.03 s

Figure 5.15 *– Comparison of execution time in seconds for Coqlex and OCamllex lexers on Miniml and JSON benchmark. The Miniml benchmark file contains 1599999 characters (for 28800 tokens) and the JSON benchmark file contains 1620948 characters (for 160489 tokens).*

Generally, Coqlex executes faster when the ratio number-of-characters/number-of-tokens is higher. In fact, in our measurements, Coqlex has better performance for the MiniML analysis where the number of characters per token is ≈55.55 (see Figure 5.15). For a similar number of characters, this performance is three times slower for the JSON benchmark where the number of characters per token is five times lower (≈10.10).

We can observe that OCamllex generated lexers execute faster than Coqlex ones. However, Coqlex generated lexer performance is surprisingly good and does not pose limitations to its usefulness in real-

world settings. In fact, Coqlex has been used to generate the lexer for the VCP Ada to VCP Ada optimizing compiler described in this thesis. This compiler is used in an industrial setting to process thousands of source code files with not a too noticeable difference with respect to OCamllex – the compiler with a verified front-end (lexer and parser) executes 5 times slower than the one with a non-verified front-end. Furthermore, the use of the *Coqlex generator*, whose lexer is implemented using the components of the Coqlex library, does not show noticeable slowness.

In regard to regexp specification, Coqlex allows to generate richer regular expressions than Verbatim++ and OCamllex. In fact, OCamllex allows to perform the regexp minus-operation only for charsets, while Coqlex allows to perform this operation on general regexps. In addition, Coqlex also allows to define function based rules other than `end-of-file`. Verbatim++ does not allow such operations. However, OCamllex allows to bind substrings matched by a regexp to identifiers, but neither Coqlex or Verbatim++ have this feature.

In regard to the syntax of the .vl files, the *Coqlex generator* is built to process a language that is very close to OCamllex. This means that there are only few differences between .vl files and their equivalent .mll files. For example, Figures 5.16 presents the OCamllex equivalent of the Coqlex lexer presented in Figure 5.2.

```
(* header section *)
{
open Lexing
open TokenDefinition.
}

(* regexp definitions *)
let ident = ['a'-'z']+
let numb = ['0'-'9']+

(* lexer definitions*)
rule minlexer = parse
| '\n' { new_line lexbuf; minlexer lexbuf }
| ident {ID (Lexing.lexeme lexbuf)}
| numb { Number (Lexing.lexeme lexbuf)}
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '(' { LPAREN }
| ')' { RPAREN }
| eof { Eof }
| _ { failwith ("unknown token : " ^ (Lexing.lexeme lexbuf))}

(* trailer section *)
{}
```

Figure 5.16 – *mini-cal.mll file*

5-I.F.1 Looping lexers

Another advantage of Coqlex is that it provides protections against infinite loops. Let us consider the Coqlex lexer specified in Figures 5.17 and the OCamllex lexer specified in 5.18. Regarding those specifications, the lexers are supposed to work as follows:

- If the remaining string is ϵ then that lexer returns the natural number '1' as token.
- Else if the longest prefix of the input string in `lexbuf` matches $[[b]] \cdot [[a]]^* \cdot [[b]]$ then that lexer returns the natural number '0' as token
- Else if it matches $[[a]]^*$ it performs a recursive call on the remaining string of `lexbuf` (updated after the election). This is a common technique used to ignore elements such as comments during lexical analysis.

When such lexer is called with a string s that starts with a character that is different from ‘a‘ and ‘b‘, the election chooses the semantic action that is associated with the regex $[[a]]^*$ with a score of 0. This means that the lexeme is ϵ and the remaining string is s. As the semantic action associated with this regexp is a recursive call, it leads to an infinite loop. The lexer generated by OCamllex from the code in Figure 5.18 loops when the input string is ‘c‘, whereas the lexer generated by Coqlex from the code in Figure 5.17 returns an error. Verbatim++ does not handle this kind of problem because semantic actions do not allow recursive calls.

```
rule my_lexer = parse
    |  'b' 'a'* 'b' { ret 0 }
    |  'a'* { my_lexer }
    | EOF { ret 1 }
    %polymorphic_storage
```

Figure 5.17 *– The Coqlex example of a lexer whose execution can loop*

```
rule my_lexer = parse
  'b' 'a'* 'b'        { 0 }
  | 'a'*            { my_lexer lexbuf }
  | EOF             { 1 }
```

Figure 5.18 *– The OCamllex example of a lexer whose execution can loop*

Furthermore, the simplicity of the Coqlex implementation allows to write proofs on Coqlex lexers. For example, we have proven that the looping lexer defined in Figure 5.17 always returns an error related to the fuel when the first character of the input string is different from ‘a‘ and ‘b‘.

5-I.F.2 Using Coqlex to build a VCP Ada to VCP Ada compiler

Using Coqlex, we generated a Coq lexical analyser from using a `.vl` file (see section 5-I.C), a file containing the specification of that lexical analyser. That specification is written using regular expressions, election policy (using the keyword `parse` or `shortest`), and semantic actions. From that specification, the Coqlex generator produces Coq code using Coqlex data structures and functions, equipped with formal proofs. The generated lexer comes with the following guarantees:

- **the correctness of regexps:** The regexps used in Coqlex come with the formal proof that the languages those regexps describe are consistent with their specification (see Section 5-I.E.1).
- **the correctness of election policies:** Each of the election policies Coqlex defines come with a formal proof that the election is correct. This correctness proof is based on the correctness of the score computation systems and their use in the election process (see Section 5-I.E.2).
- **transparency and protection against looping:** The Coq code produced by the Coqlex generator is human readable, making it easy to review. Thanks to the fuel technique, Coqlex provides also guarantees that the generated lexer will terminate.

The implementation of the lexer used in the VCP Ada to VCP Ada compiler requires no particular techniques. The `.vl` code is very similar to its equivalent in Ocamllex.

5-II Generating a formally verified parser

Parsers are in charge of syntax analysis. Their role is to transform a token stream into an abstract syntax tree (AST). The most common approach to write a parser is to use parser generators, that generate parsers using a specification whose syntax is very similar to BNF notation [McCracken and Reilly, 2003]. This notation describes a grammar associated with semantic actions that indicate how to produce the AST of the input.

Obtaining a fully formally verified front-end requires the formal verification of its parser. Existing literature proposes solutions to generate formally verified parsers [Jourdan et al., 2012]. This formal verification is based on 3 properties:

1. **soundness:** the generated parser accepts only valid input.

2. **safety:** no internal error can occur during a syntax analysis using the generated parser.

3. **completeness:** the generated parser accepts all the valid input.

Menhir [Régis-Gianas, 2016] is a parser generator whose usage is very similar to ocamlyacc [Smith, 2007]. It aims to transform high-level grammar specifications, associated with semantic actions (fragments of executable code), into parsers. Menhir offers 3 back-ends allowing to specify the language of the output parser:

- **The "code" back-end:** It requires writing the semantic actions in OCaml and storing the specification grammar in a file whose extension is *.mly*. It is the default back-end and it writes the parser in a **stand-alone** Ocaml parser that it stores in 2 files: a *.mli* (interface file) and a *.ml* (implementation) file.

- **The "table" back-end:** That is very similar to the "code" back-end except that the generated parser needs to be linked to the library **MenhirLib** during the compilation. It is selected by passing `-table` on the command line.

- **The "coq" back-end:** It requires writing the semantic actions in Coq and to store the specification grammar in a file whose extension is *.vy*. The generated parser is written in Coq and stored in a *.v* file that contains the parser but also the proof that the parser is correct with respect to the grammar. It is selected by passing `-coq` on the command line.

For our implementation, we used the Coq back-end that comes with several restrictions such as the absence of error-handling mechanism, the obligation to precise the type of every non-terminal symbol and the absence of support for conflict resolution using priority and associativity declarations.

The implementation of the parser of our source code optimizer starts with the conversion of the syntax described in *VCP Ada syntax section* into Menhir .vy format. A naive translation comes with usual parser conflict resolutions [Clark, 2002]. The second step of our implementation consists in branching the CoqLex generated lexer to the Menhir generated parser.

Menhir generated parsers take a stream – a parametric data structure defined in MenhirLib – of tokens as input and generate the output AST while CoqLex generated lexers take a lexbuf and a storage then generates a data structure containing the returned token, a lexbuf and a storage. To allow branching, we implemented a function that takes a lexer, a lexbuf and a storage and returns a stream of tokens. This stream is obtained by successive calls to the lexer with the returned lexbuf and storage as parameters.

The OCaml code for this generation can be written as follows:

```
let _lexbuf_copy = ref None

let token_stream lexer  lexbuf storage : MenhirLibParser.Inter.buffer =
                let rec compute lexbuf storage () =
                        let loop tokenXlexbufXstorage =
                                let token, lexbuf, storage = tokenXlexbufXstorage in
                                let _ = _lexbuf_copy_ref := Some lexbuf in
                                MenhirLibParser.Inter.Buf_cons (token, Lazy.from_fun (compute lexbuf storage))
                        in loop (coqlexer lexbuf storage)
                in
                Lazy.from_fun (compute lexbuf storage)
```

Listing 5 – *From Coqlex lexer to token stream*

In this code, `token_stream` is in charge of the transformation. This function takes as input a lexer called `lexer`, a lexbuf called `lexbuf` and a storage `storage` and returns a token stream. In OCaml, this stream is a lazy function – a data structure allowing deferred computations. The `_lebuf_copy` variable is a lexbuf reference that aims to store the latest value of the lexbuf. This reference helps to provide a more explicit error message when a lexical or syntax error occurs. This error message can be written using the following function

```
let coq_report coq_lexbuf =
          Format.eprintf "Line %d, characters %d-%d : Error near \"%s\"@."
                  coq_lexbuf.cur_pos.l_number
                  coq_lexbuf.s_pos.c_number
                  coq_lexbuf.cur_pos.c_number
                  (string_of_char_list coq_lexbuf.str_val)
```

Listing 6 – *Reporting parsing errors*

The use of Menhir associated with the Coq back-end as option generates a parser equipped with the following guarantees:

- **Termination:** As in Coqlex, Menhir uses the fuel technique to ensure that the parser will terminate.
- **Correctness:** the Coq back-end generates a Coq proof of the correctness of the generated parser. This proof shows that: if a word (a prefix of the input stream) is accepted, then this word is valid (with respect to the grammar) and the semantic value that is constructed by the parser is valid as well (with respect to the grammar)
- **Completeness:** Menhir also generates a Coq proof of the completeness of the generated parser. This proof shows that: if a word (a prefix of the input stream) is valid (with respect to the grammar), then (given sufficient fuel) it is accepted by the parser.

These results imply that the grammar is unambiguous: for every input, there is at most one valid interpretation.

5-III Conclusion

The previous chapter suggested to optimize programs generated using the Siemens development process by implementing a VCP Ada to VCP Ada formally verified compiler. The compiler this thesis aims to implement consists of three parts: a front-end, a middle-end and a back-end. This chapter provides us with a solution to generate a front-end equipped with formal proof of its correctness. The next step consists in the implementation of the middle-end of that compiler. However, proving the correctness of the middle-end requires describing the formal semantics of VCP Ada. The next chapter focuses on that description.

Chapter 6

VCP Ada Formal semantics

The purpose of formal semantics is to describe the behavior of programs using mathematical constructions. Those descriptions are the core of semantic preservation – a theorem that ensures that a given optimization (program transformation) will not change the behavior of programs (see Section 2-IV.C on page 26) – proofs of the optimization we implemented for the VCP Ada – a subset of Ada that uses the Vital Coded Processor technique (see chapters 2 and 3) – source code we presented and implemented in Chapter 7.

For the description of the behaviour of the VCP Ada language, we will used the *big-step semantics* [Kahn, 1987], because its low level of detail makes the proof simple to understand and simplifies the implementation of the semantic preservation proof of program transformations. This chapter will thus provide a formal description and implementation details of the VCP Ada semantics that will be used for the semantic preservation proof.

6-I With statements, available source code and missing declaration problem

As in many modular programming languages – programming languages that allow program implementation in form of several smaller files (packages) written separately that are compiled separately and linked together to build an executable – VCP Ada describes and respects visibility rules. In fact, from within a package, the procedures and variables declared in other package declarations are made visible by file inclusion, performed by **with statements** (see `<with-statement>` in section 3-I.B.4). This statement is similar to the *#include* statement in the C programming language or the *import* statement in Python.

In CompCert [Leroy et al., 2016], a formally verified C compiler, file inclusion is handled by an external preprocessor which is not part of the CompCert distribution. That preprocessor is software that is in charge of processing file inclusion, macro expansion and conditional compilation. This preprocessing leads to the generation of C code containing the declarations of variables and functions defined in other files, and has to be trusted because it could introduce errors during preprocessing. Without a similar available tool and to avoid using a preprocessor which can introduce errors, for VCP Ada, we achieved this task differently.

File inclusion in Ada and VCP Ada [Taft, 1997] can be illustrated in the example program structure shown in Figure 6.1 on page 66. In this example, the visibility that the VCP Ada file inclusion allows is summarized as follows:

- **a.ads:** cannot see anything from another package because it is a declaration package that contains no *with* statements.
- **a.adb:** can see `a1`, `a2` and `a3` from *a.ads* because *a.ads* is the declaration package that *a.adb* implements. Those elements (`a1`, `a2` and `a3`) are considered as global variables in *a.adb*.

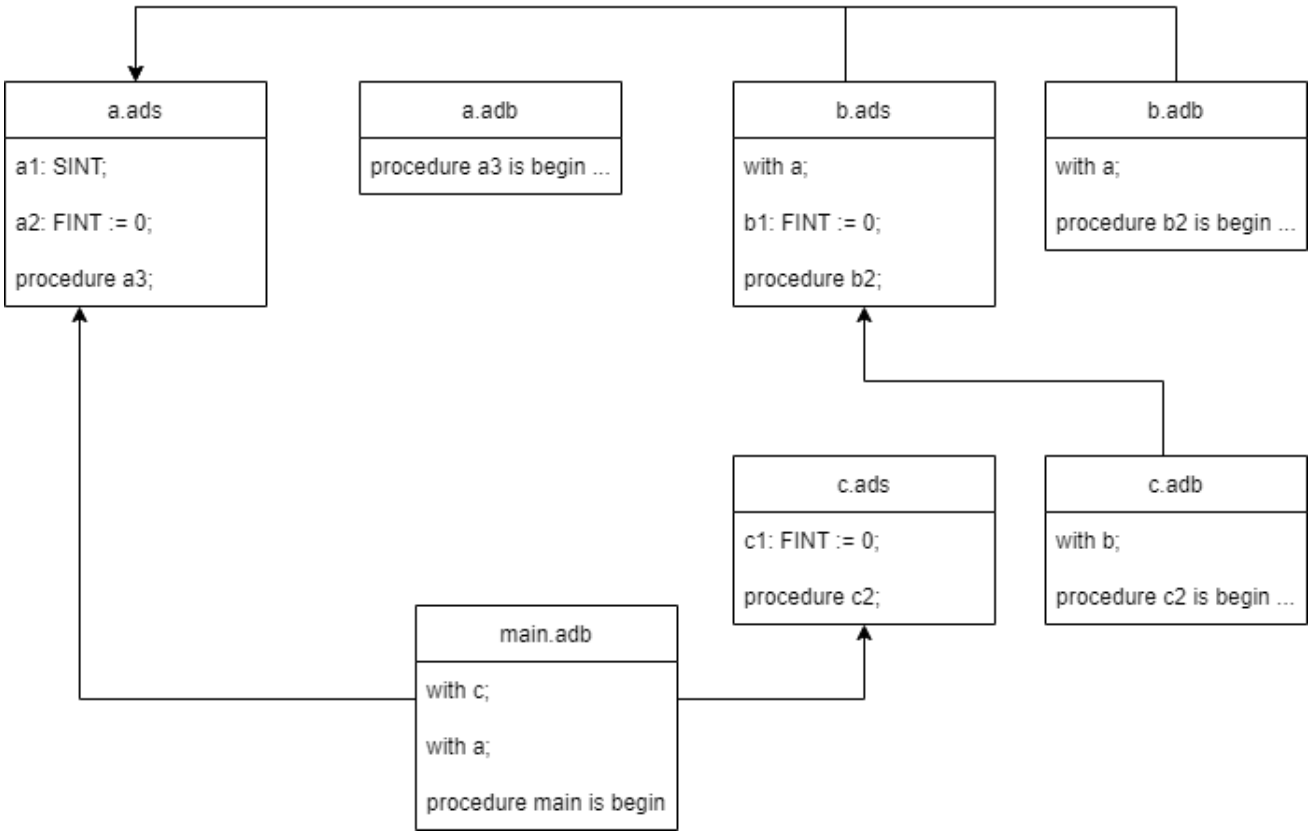


Figure 6.1 *– Example of a VCP Ada program structure*

- **b.ads:** can see `a1`, `a2` and `a3` from *a.ads* because of the *with a* statement. Those elements are considered as external elements.
- **b.adb:** can see `a1`, `a2` and `a3` from *a.ads* – as external elements – because of the *with a* . It also have access to elements defined in its declaration package: `b1`, `b2`.
- **c.ads:** cannot see anything from another package.
- **c.adb:** can see `b1` and `b2` from *b.ads* – as external elements – because of the *with b* . It also have access to elements defined in its declaration package: `c1` and `c2`.

  Remark: *b.ads* can see `a1`, `a2` and `a3` from *a.ads* but *c.adb*, which imports *b.ads* cannot see those elements.
- **main.adb:** can see `c1` and `c2` from *c.ads*, and `a1`, `a2` and `a3` from *a.ads* – as external elements.

The source code optimizer this thesis targets to build is software that analyses and optimizes programs, organized in the form of projects, in order to reduce their execution time. In practice, a typical VCP Ada project is a set of files that contains Ada, VCP Ada source code and in some cases, C source code located in one or many directories. With those files comes a text-file that locates the list of VCP Ada of this given project. As the software we are building is designed to process only VCP Ada files, there are projects where this source code optimizer cannot read the whole project source code. For example, when this project contains Ada files that uses a syntax that is not allowed in VCP Ada, or non accessible source code (files that cannot be accessed at compile time due access rights for example). When those kinds of files are imported by the VCP Ada files to optimize, this means that the analysis of those files will be possibly done without the variable or procedure declarations contained in the imported files.

In those cases, we expect the software we target to build will be able to perform correct optimizations as much as it can. This expectation implies implementation choices that we will describe in next sections. Later in this chapter and the next one, we will use the term *missing declarations* to refer to those cases.

6-II The abstract machine

The *big-step semantics*, like other operational semantics, describes the effect of programs on an abstract machine (more details in Chapter 2). This abstract machine is a data structure whose description depends on the theorem we want to prove. In this thesis, we want to transform the instructions of programs in order to reduce the execution time of those programs. As in other imperative languages, VCP Ada instructions can perform modifications of the values of variables and perform interactions with hardware (ex: printing a message on a screen) using function or procedure calls. To simulate both kinds of actions, we describe the abstract machine in form of a pair (σ, ϖ) where σ is the environment and ϖ is the program trace, the environment recording the modification of variable and the trace recording the interactions of the program with the outside (external calls).

6-II.A The abstract machine environment

The abstract machine environment (σ) is designed to store variables and procedures. According to the VCP Ada syntax, variables can be defined in four places: in package declarations, package definitions, in the definitions of procedure parameters and inside procedures. Depending on where a given variable has been defined, a given procedure can access this variable or not. Considering a procedure, variables are categorized depending on the place where they are declared. This category is called the scope of the variable. In VCP Ada, we identify three scopes for variables:

- **local variables:** Considering a procedure, local variables are variables that are declared either as procedure parameters or inside the procedure. Those variables are only visible and accessible by the instructions of this procedure.
- **global variables:** Considering a procedure, global variables are variables that are declared either in the same package declaration where this procedure is declared or in the same package definition where this procedure is defined. The set of global variables is the same for all the procedures defined in the same package definition.
- **external variables:** Considering a procedure, external variables are variables that are declared in a package declaration that is different from the package declaration where this procedure is declared. Those variables are made visible by the with statement. The set of external variables is the same for all the procedures defined in the same package definition.

VCP Ada provides built-in data structures such as *vital* (variables that are associated with signatures) and *non-vital* (variables that are not associated to signatures) *integers* (`SINT` and `FINT`), *vital* and *non-vital booleans* (`SBOOL` and `FBOOL`), *vital 1-D* and *2-D* arrays of integers or booleans, *strings* (only as function or procedure parameters). It also allows to create an alias for data-structures using subtype or define new types using record definition (see `<subtype-decl>` and `<record-decl>` in Section 3-I.B.3.3). Regarding the limitations that the VCP Ada syntax poses and the Ada semantics, subtype works as type renaming. Consequently, the data structure (`value`) describing the values of variables in the environment can be described using 6 categories:

1. **Integers:** Integer values represent (vital and non-vital) integers in VCP Ada.
2. **Booleans:** Boolean values represent (vital and non-vital) booleans in VCP Ada.
3. **Strings:** String values represent strings in VCP Ada.
4. **Arrays:** Array values represent arrays in VCP Ada.

5. **Records:** Record values represent records in VCP Ada.
6. **Undefined:** Undefined values represent the value of a non-initialized variable. This is also used to indicate the value of variables whose declarations are missing (see Section 6-I).

This value model is similar to work performed in *formal semantics research around Spark2014* (Sparkformal) [Aponte et al., 2014, Courtieu et al., 2013].

We give some further remarks regarding variables :

- **difference between vital and non-vital variables:** This modelling does not make difference between vital and non-vital variables (see more details on vital and non-vital variables in Section 2-III and Section 3-I). We explicitly made the choice of storing vital variables in the same data structures as non-vital ones. We did it because from the point of view of developers, the vital coding is transparent. The only difference between vital and non-vital variables resides in signatures whose exact values and evolutions will be decided at compile time by the *signature predetermination tool* (SPT).
- **constant and non-constant variables:** At declaration time, variables can be declared as constant (non modifiable) or non-constant (modifiable). The environment must also be able to say if a variable is constant or not. Constant variables are not assignable.

A similar distinction is made for procedure. We identify two scopes:

local procedures: Considering a procedure, local procedures are procedures that are declared either in the same package declaration where this procedure is declared or in the same package definition where this procedure is defined.

external procedures: Considering a procedure, external procedures are procedures that are declared in a package declaration that is different from the package declaration where this procedure is declared. Those procedures are made visible by the with statement. The set of external procedures is the same for all the procedures defined is the same package definition.

Remark: External procedures are limited to declarations because in Ada, a package can only access declaration from imported files. We decided to respect this visibility aspect in our implementation.

6-II.B The abstract machine trace

Instructions can call external functions or procedures – functions or procedures defined in another file – in order to perform interaction such as writing data on hardware. The environment, as defined above, cannot take into account this kind of instruction. For example, if we assume that there exists a procedure called `print` that prints the value of a variable on standard output, from the point of view of the environment, the two listings below are equivalent:

Code with interaction

```
1 v := assign(20);
2 print(v);
3 d := assign(30);
4 print(d);
```

Code without interaction

```
1 v := assign(20);
2 d := assign(30);
```

Listing 7 *– Example of 2 source code that are equivalent from the point of view of the environment*

In fact, as the `print` procedure does not modify the values of variables, the environment cannot reflect those calls on its data structure. Like in CompCert, as a solution to address this issue, we decided

to add additional information to the abstract machine: the program trace, a data structure that reports on external function or procedure calls.

In addition to calls to functions or procedures such as `print` that do no modify the values of variables, there exist other functions that can have impacts on variables' values but whose source code is either not accessible nor interpretable (see missing declaration problem in Section 6-I on page 65). Consequently, calls to the functions or procedures whose definitions are not stored in the environment of the abstract are also stored in the trace, as in CompCert.

We defined the program trace as an ordered list of events. An event models a call to an external function or procedure. We describe the event data structure to contain 4 components: the name – in form of selected component – of the function or procedure that is called, the signature of this function or procedure (optional, in case the function or procedure declaration is missing), the list of call parameters of this function or procedure call and the environment at the moment of the function or procedure call. The program trace will be described using the list notation: [] refers to empty list (in that case, a trace), $h :: \varpi$ refers to a list where h is the first element (in that case, an event) of the list, and ϖ is the tail of the list (in that case, the events that happened before).

In the previous figure, the abstract machines that result from the evaluation of those two codes are different because their traces are different.

6-III The semantics of VCP Ada

This section focuses on the description of the effect of VCP Ada instructions on the abstract machine. In this description, the notation $\frac{H_0 \quad ... \quad H_n}{R}$ C (inference rule notation) indicates that if the hypotheses $H_0, ..., H_n$ and the condition C are verified, then the conclusion R is verified. The organization and titles of this section follow the structure of the syntax description presented in Section 3-I.B.3.

6-III.A Item names and expressions

Objects such as variables and functions are identified by *selected components* or names – non-empty sequences of non-empty strings (identifiers) separated by a dot (see `<name>` described in Section 3-I.B.3). For example, when there is a package named p that contains a variable named b, the value of this variable is bound to $p.b$. In addition, when a variable is r is a record containing a field name f, the value of this field can be extracted using the syntax $r.f$.

The evaluation of a selected component is done by successive evaluations of the identifiers that compose it. For example, to evaluate the value of $a_0.a_1.a_2...a_n$, the system first evaluates a_0. This evaluation respects a hierarchy: the value of a_0 is first evaluated in the local scope. If no values are found, a_0 is then evaluated in the global scope and if no values is found, then the environment has to check if there exists a package whose name is a_0. If no values or packages are found, the environment can state that there is no variable associated with a_0 or $a_0.a_1.a_2...a_n$. Else, the evaluation of $a_0.a_1.a_2...a_n$ can be described as follows:

- **Case 1:** if a_0 refers to a record r (from local or global scope), the evaluation is equivalent to evaluating $a_1...a_n$ in r.
- **Case 2:** if a_0 refers to a procedure or type of value that is distinct from a record (from local or global scope), the evaluation fails or is undefined because none of those typed-elements contains accessible fields.
- **Case 3:** if a_0 refers to a package p, the system evaluates the value v of a_1 in the package. Then we go to case 2 or 3 in order to analyse the value of $a_2....a_n$ depending on whether of not v is a record.

When the selected component consists in a unique string, it is called an identifier. Two variables can be identified by the same selected component or identifier if and only if those variables are declared in different scopes. Except that case, considering an environment, a given selected component identifies an unique element (procedure, variable or package).

To define the semantics for expression evaluation, we will use the following notation: $\sigma \vdash_{var} id \mapsto v$ indicates that the name id identifies a variable in the environment σ and this variable contains the value v. The notation $\sigma \vdash_{var} id \mapsto \varnothing$ indicates that the name id identifies nothing in the environment σ (this can be a consequence of the missing declaration problem described in Section 6-I on page 65) and the notation $\varkappa$ refers to the value **Undefined** (used for non-initialised values and variables whose declarations are missing).

For this definition, we assume the functions `MathModule.binary_operation` and `MathModule.unary_operation` give semantics to VCP Ada binary and unary operators such as 'and', 'or', 'not', '-', *. We also assume that those functions can succeed (`MathModule.binary_operation` $op\ v_1\ v_2 = v$) or fail (`MathModule.binary_operation` $op\ v_1\ v_2 = \varnothing$) when an error such as type error (e.g. 1 + true) is discovered. The notation $(\sigma, \omega) \vdash e \mapsto v$ indicates that the evaluation of the expression e from an abstract machine (σ, ω) gives the value v.

$$\frac{\sigma \vdash_{var} id \mapsto v}{(\sigma,\omega) \vdash id \mapsto v} \quad \frac{\sigma \vdash_{var} id \mapsto \varnothing}{(\sigma,\omega) \vdash id \mapsto \varkappa} \quad \frac{}{(\sigma,\omega) \vdash v \mapsto v}\ [v \text{ is a value (integer, boolean, string, record or array)}]$$

$$\frac{(\sigma,\omega) \vdash e_1 \mapsto v_1 \qquad (\sigma,\omega) \vdash e_2 \mapsto v_2 \qquad (\texttt{MathModule.binary\_operation}\ op\ v_1\ v_2) = v}{(\sigma,\omega) \vdash e_1\ op\ e_2 \mapsto v}$$

$$\frac{(\sigma,\omega) \vdash e_1 \mapsto v_1 \qquad (\texttt{MathModule.unary\_operation}\ op\ v_1) = v}{(\sigma,\omega) \vdash op\ e_1 \mapsto v}$$

Listing 8 *– VCP Ada semantics - expression evaluation inference rules (part 1 of 3)*

We impose that the value $\varkappa$ is absorbent. This means that if no error is detected in the evaluation of an expression where the value $\varkappa$ appears, the result of this evaluation is $\varkappa$. For example, $\varkappa + 5 = \varkappa$.

As the environment uses the same data types to store VCP Ada vital and non-vital variables, the description above holds for most vital expression syntax. To describe the evaluation of the other elements, we will use the notation $t[i]$ to refer to the value at position i (starting from 0) in the array value t. The notation $t[i_0, i_1]$ refers to the value at position (i_0, i_1) in the 2D array value t. For this operation (selection of elements in an array), we impose that for all i_0 and i_1, $\varkappa[i_0] = \varkappa$ and $\varkappa[i_0, i_1] = \varkappa$. This decision is made to be coherent with the missing declaration problem. In fact, if the value of an array is not accessible, then the value of the element this array contains is not accessible. The semantics of expressions such as `div2`, `assign` and `read` are described in the figure below:

$$\frac{\sigma \vdash_{var} id \mapsto v_0 \quad (\sigma,\varpi) \vdash e \mapsto v_1 \quad (\texttt{MathModule.binary\_operation `=`}\ v_0\ v_1) = v}{(\sigma,\varpi) \vdash \texttt{equal}(id,e) \mapsto v}$$

$$\frac{\sigma \vdash_{var} id \mapsto v_0 \quad (\sigma,\varpi) \vdash e \mapsto v_1 \quad (\texttt{MathModule.binary\_operation `}\neq\texttt{`}\ v_0\ v_1) = v}{(\sigma,\varpi) \vdash \texttt{diff}(id,e) \mapsto v}$$

$$\frac{\sigma \vdash_{var} id \mapsto v_0 \quad (\sigma,\varpi) \vdash e \mapsto v_1 \quad (\texttt{MathModule.binary\_operation `/`}\ v_0\ 2^{v_1}) = v}{(\sigma,\varpi) \vdash \texttt{div2}(id,e) \mapsto v}$$

$$\frac{(\sigma,\varpi) \vdash e \mapsto v}{(\sigma,\varpi) \vdash \texttt{assign}(e) \mapsto v} \quad \frac{(\sigma,\varpi) \vdash e \mapsto v}{(\sigma,\varpi) \vdash \texttt{w}(e) \mapsto v} \quad \frac{(\sigma,\varpi) \vdash id \mapsto v}{(\sigma,\varpi) \vdash \texttt{t}(id) \mapsto v}$$

$$\frac{(\sigma,\varpi) \vdash id \mapsto t \quad (\sigma,\varpi) \vdash id' \mapsto i}{(\sigma,\varpi) \vdash \texttt{read}(id,id') \mapsto t[i]}\ [i \in \mathbb{N}]$$

$$\frac{(\sigma,\varpi) \vdash id \mapsto t \quad (\sigma,\varpi) \vdash id_0 \mapsto i_0 \quad (\sigma,\varpi) \vdash id_1 \mapsto i_1}{(\sigma,\varpi) \vdash \texttt{read}(id,id_0,id_1) \mapsto t[i_0,i_1]}\ [i_0,i_1 \in \mathbb{N}]$$

Listing 9 *– VCP Ada semantics - expression evaluation inference rules (part 2 of 3)*

The functions `assign`, `t` and `w` are special expression evaluation functions that allow to the SPT to set up the signatures correctly at compilation time and to verify those signature at execution time. For example, the function `t` indicates to the SPT the branch condition of an if/else instruction, allowing to update the tracer correctly (see Section 3-I.A). As we do not handle signature management in our semantics, the evaluation of those functions is equivalent to the evaluation of their arguments.

Functions such as `hard_input_valid` and `is_valid` aim to express a result of interactions with another device. For example, `is_valid` checks whether or not the co-processor, an additional processor dedicated to vital processing (signature computation), is still online. The interactions performed by the program can impact this result. To simulate this behaviour, we assume that there are two abstract functions – `ops_hard_input_valid` for `hard_input_valid` and `ops_is_valid` for `is_valid` – that take the current environment and return a boolean, that we assume equal to the result that the function they simulate. The evaluation rules for expression evaluation can be completed as follows:

$$\frac{\texttt{ops\_is\_input\_valid}\ \varpi = v}{(\sigma,\varpi) \vdash \texttt{hard\_input\_valid} \mapsto v} \quad \frac{\texttt{ops\_is\_valid}\ \varpi = v}{(\sigma,\varpi) \vdash \texttt{is\_valid} \mapsto v}$$

Listing 10 *– VCP Ada semantics - expression evaluation inference rules (part 3 of 3)*

6-III.B Procedure call and instructions

Instructions are described using six categories: the null instruction, the assign instruction, the if-else instruction, the case instruction, the while instruction and the procedure call instruction (see `<inst>` in Section 3-I.B.3.2 on page 35). To describe their semantics, we will use the following notations: σ refers to an environment and ϖ refers to a program trace. The notation (σ,ϖ) refers to the state of the abstract machine in which the evaluation must be done. The notation $(\sigma,\varpi) \models i \Rightarrow (\circ,\sigma',\varpi')$ means that the evaluation of the instruction i in the abstract machine (σ,ϖ) succeeds and the transforms the input abstract machine state into (σ',ϖ'). In case of failure, the symbol $\circ$ is replaced by the symbol $\bullet$. In that case, the environment and the trace describe the state of the machine at the moment of that failure. The notation $\sigma[var \leftarrow v]$ refers to σ environment where the variable var successfully assigned to the value v (this assignment can fail when the variable to assign is constant). The evaluation rules for the null instruction, the assign instruction, the if-else instruction, the case instruction, the while instruction can be written as follows:

$$\frac{}{(\sigma,\varpi) \models \texttt{null;} \Rightarrow (\circ,\sigma,\varpi)} \qquad \frac{\sigma \vdash val \mapsto v}{(\sigma,\varpi) \models var\texttt{:=}val\texttt{;} \Rightarrow (\circ,\sigma[var \leftarrow v],\varpi)}$$

$$\frac{(\sigma,\varpi) \models i_0\texttt{;} \Rightarrow (\bullet,\sigma',\varpi')}{(\sigma,\varpi) \models i_0\texttt{;}\ \ i_1\texttt{;}\ \ \ldots\texttt{;}\ \ i_n\texttt{;} \Rightarrow (\bullet,\sigma',\varpi')} \qquad \frac{(\sigma,\varpi) \models i_0\texttt{;} \Rightarrow (\circ,\sigma',\varpi') \qquad (\sigma',\varpi') \models i_1\texttt{;}\ \ \ldots\texttt{;}\ \ i_n\texttt{;} \Rightarrow r}{(\sigma,\varpi) \models i_0\texttt{;}\ \ i_1\texttt{;}\ \ \ldots\texttt{;}\ \ i_n\texttt{;} \Rightarrow r}$$

$$\frac{(\sigma,\varpi) \vdash c \mapsto false}{(\sigma,\varpi) \models \texttt{while}\ \ c\ \ \texttt{loop}\ \ i\ \ \texttt{end loop;} \Rightarrow (\circ,\sigma,\varpi)}$$

$$\frac{(\sigma,\varpi) \vdash c \mapsto true \qquad (\sigma,\varpi) \models \texttt{i;} \Rightarrow (\bullet,\sigma',\varpi')}{(\sigma,\varpi) \models \texttt{while}\ \ c\ \ \texttt{loop}\ \ i\ \ \texttt{end loop;} \Rightarrow (\bullet,\sigma',\varpi')}$$

$$\frac{(\sigma,\varpi) \vdash c \mapsto true \qquad (\sigma,\varpi) \models \texttt{i;} \Rightarrow (\circ,\sigma',\varpi') \qquad (\sigma',\varpi') \models \texttt{while}\ \ c\ \ \texttt{loop}\ \ i\ \ \texttt{end loop;} \Rightarrow r}{(\sigma,\varpi) \models \texttt{while}\ \ c\ \ \texttt{loop}\ \ i\ \ \texttt{end loop;} \Rightarrow r}$$

$$\frac{(\sigma,\varpi) \vdash c \mapsto true \qquad (\sigma,\varpi) \models \texttt{i;} \Rightarrow r}{(\sigma,\varpi) \models \texttt{if}\ \ c\ \ \texttt{then}\ \ i\ \ \texttt{elsif}\ \ c_2\ \ \texttt{then}\ \ i_2\ \ \ldots\ \ \ \texttt{else}\ \ i_n\ \ \texttt{end if;} \Rightarrow r}$$

$$\frac{(\sigma,\varpi) \vdash c \mapsto false \qquad (\sigma,\varpi) \models \texttt{if}\ \ c_2\ \ \texttt{then}\ \ i_2\ \ \ldots\ \ \ \texttt{else}\ \ i_n\ \ \texttt{end if;} \Rightarrow r}{(\sigma,\varpi) \models \texttt{if}\ \ c\ \ \texttt{then}\ \ i\ \ \texttt{elsif}\ \ c_2\ \ \texttt{then}\ \ i_2\ \ \ldots\ \ \ \texttt{else}\ \ i_n\ \ \texttt{end if;} \Rightarrow r}$$

$$\frac{(\sigma,\varpi) \vdash c \mapsto false \qquad (\sigma,\varpi) \models i_2 \Rightarrow r}{(\sigma,\varpi) \models \texttt{if}\ \ c\ \ \texttt{then}\ \ i\ \ \texttt{else}\ \ i_2\ \ \texttt{end if;} \Rightarrow r}$$

$$\frac{(\sigma,\varpi) \vdash (var\ \texttt{=}\ c) \mapsto true \qquad (\sigma,\varpi) \models \texttt{i;} \Rightarrow r}{(\sigma,\varpi) \models \texttt{case c(}var\texttt{)}\ \ \texttt{when}\ \ c\ \ \texttt{=>}\ \ i\ \ \texttt{when}\ \ c_2\ \ \texttt{=>}\ \ i_2\ \ \ldots\ \ \ \texttt{when others =>}\ \ i_n\ \ \texttt{end case;} \Rightarrow r}$$

$$\frac{(\sigma,\varpi) \vdash (var\ \texttt{=}\ c) \mapsto false \qquad (\sigma,\varpi) \models \texttt{case c(}var\texttt{)}\ \ \texttt{when}\ \ c_2\ \ \texttt{=>}\ \ i_2\ \ \ldots\ \ \ \texttt{when others =>}\ \ i_n\ \ \texttt{end case;} \Rightarrow r}{(\sigma,\varpi) \models \texttt{case c(}var\texttt{)}\ \ \texttt{when}\ \ c\ \ \texttt{=>}\ \ i\ \ \texttt{when}\ \ c_2\ \ \texttt{=>}\ \ i_2\ \ \ldots\ \ \ \texttt{when others =>}\ \ i_n\ \ \texttt{end case;} \Rightarrow r}$$

$$\frac{(\sigma,\varpi) \vdash (var\ \texttt{=}\ c) \mapsto false \qquad (\sigma,\varpi) \models i_2 \Rightarrow (\circ,\sigma',\varpi')}{(\sigma,\varpi) \models \texttt{case c(}var\texttt{)}\ \ \texttt{when}\ \ c\ \ \texttt{=>}\ \ i\ \ \texttt{when others =>}\ \ i_2\ \ \texttt{end case;} \Rightarrow (\bullet,\sigma',\varpi')}$$

$$\frac{(\sigma,\varpi) \vdash (var\ \texttt{=}\ c) \mapsto false \qquad (\sigma,\varpi) \models i_2 \Rightarrow (\bullet,\sigma',\varpi')}{(\sigma,\varpi) \models \texttt{case c(}var\texttt{)}\ \ \texttt{when}\ \ c\ \ \texttt{=>}\ \ i\ \ \texttt{when others =>}\ \ i_2\ \ \texttt{end case;} \Rightarrow (\bullet,\sigma',\varpi')}$$

Listing 11 *– VCP Ada semantics - instruction evaluation inference rules*

Remark: the default case of case instructions always fails. In fact, the VCP Ada case instruction is designed to cite all the possible values for the case parameter (the variable used in the function `c` after the keyword *case*), making the *default* branch of case instruction synonym of error when taken (see VCP Ada language instruction description in Section 3-I.B.3.2 for more details).

To deal with the missing declaration problem, when an instruction is trying to assign a value to an external variable that does not exist in the environment, the environment creates that variable and initializes it, with that assignment value.

VCP Ada defines two sorts of procedure calls: the built-in procedures and the user defined procedures. The evaluation of procedure calls relies on a function whose name is `make_event`. The `make_event` function is in charge of building an event to store in the trace. This function takes three parameters, the current environment, the name of the procedure that is called and the list of the parameters. Using that function, we can write the rules to evaluate calls to most of the VCP Ada built-in procedures (all the procedures except `read_hard_output`) in the environment as follows:

$$\frac{}{(\sigma,\varpi) \models \texttt{end\_branch}(n)\,; \Rightarrow (\circ,\sigma,\varpi)} \quad \frac{}{(\sigma,\varpi) \models \texttt{end\_iteration}; \Rightarrow (\circ,\sigma,\varpi)}$$

$$\frac{}{(\sigma,\varpi) \models \texttt{converge}(n)\,; \Rightarrow (\circ,\sigma,\varpi)} \quad \frac{}{(\sigma,\varpi) \models \texttt{begin\_loop}(n)\,; \Rightarrow (\circ,\sigma,\varpi)}$$

$$\frac{\texttt{make\_event}(\sigma,\texttt{init},[v]) = e}{(\sigma,\varpi) \models \texttt{init}(v)\,; \Rightarrow (\circ,\sigma,e :: \varpi)}$$

$$\frac{(\sigma,\varpi) \vdash exp \mapsto v \qquad \texttt{make\_event}(\sigma,\texttt{init},[var;\ exp]) = e}{(\sigma,\varpi) \models \texttt{init}(var,\ exp)\,; \Rightarrow (\circ,\sigma[var \leftarrow v], e :: \varpi)}\ [var \text{ is an integer or a boolean}]$$

$$\frac{(\sigma,\varpi) \vdash exp \mapsto v \qquad \texttt{make\_event}(\sigma,\texttt{init},[var;\ exp]) = e}{(\sigma,\varpi) \models \texttt{init}(var,\ exp)\,; \Rightarrow (\circ,\sigma[\forall i, var[i] \leftarrow v)], e :: \varpi)}\ [var \text{ is a 1D array}]$$

$$\frac{(\sigma,\varpi) \vdash exp \mapsto v \qquad \texttt{make\_event}(\sigma,\texttt{init},[var;\ exp]) = e}{(\sigma,\varpi) \models \texttt{init}(var,\ exp)\,; \Rightarrow (\circ,\sigma[\forall i_1, i_2, var[i_1][i_2] \leftarrow v], e :: \varpi)}\ [var \text{ is a 2D array}]$$

$$\frac{(\sigma,\varpi) \vdash r \mapsto v \qquad (\sigma,\varpi) \vdash idx \mapsto i_1 \qquad (\sigma,\varpi) \vdash idx' \mapsto i_2}{(\sigma,\varpi) \models \texttt{write}(var,\ idx,\ idx',\ r)\,; \Rightarrow (\circ,\sigma[var[i_1][i_2] \leftarrow v], \varpi)}\ [i_1, i_2 \in \mathbb{N}]$$

$$\frac{(\sigma,\varpi) \vdash r \mapsto v \qquad (\sigma,\varpi) \vdash idx \mapsto i}{(\sigma,\varpi) \models \texttt{write}(var,\ idx,\ r)\,; \Rightarrow (\circ,\sigma[var[i] \leftarrow v], \varpi)}\ [i \in \mathbb{N}]$$

$$\frac{\texttt{make\_event}(\sigma,\texttt{write\_hard\_output},[from; to; offset; table]) = e}{(\sigma,\varpi) \models \texttt{write\_hard\_output}(from; to; offset; table)\,; \Rightarrow (\circ,\sigma,e :: \varpi)} \quad \frac{\texttt{make\_event}(\sigma,\ \texttt{failure},\ []) = e}{(\sigma,\varpi) \models \texttt{failure}; \Rightarrow (\bullet,\sigma,e :: \varpi)}$$

Listing 12 *– VCP Ada semantics - instruction evaluation inference rules 2*

Those rules use the notation $\sigma[\texttt{var}[i] \leftarrow v]$ that refers to the environment σ where the element at index i of var, a 1D array variable, is assigned to the value v. The notation $\sigma[\forall i, \texttt{var}[i] \leftarrow v)]$ refers to the environment σ where all the elements of the var (a 1D array) are assigned to the value v. Similar notations ($\sigma[\texttt{var}[i_1][i_2] \leftarrow v]$ and $\sigma[\forall i_1, i_2, \texttt{var}[i_1][i_2] \leftarrow v]$) are used for 2D arrays.

Remarks:

- **the end_branch, end_iteration, converge and begin_loop procedure calls are ignored**. This means that those instructions have no effect either on the environment or on the trace. In fact, those instructions work like annotations that are only used by the signature predetermination tool (SPT) and have no impact on our abstract machine. Consequently, they can be seen as syntax elements with no meaning.
- **the init procedure is logged in the program trace**. We decided to do so in order to avoid confusion with the *assignment* instruction.

For the evaluation of the `read_hard_output` procedure, a procedure that instructs the system to read data from hardware, we assume the existence of an abstract function called `read_out` that takes the current trace, the list of events (interactions) that are registered since the beginning of the evaluation of the program, and the parameters used the `read_hard_output` procedure call. This abstract function (`read_out`) simulates reading from hardware and returns the read value. The evaluation rule for the `read_hard_output` procedure can be written as follows:

$$\frac{\texttt{make\_event}(\sigma,\texttt{read\_hard\_output},[where; from; offset; table]) = e \qquad \texttt{read\_out}\ \ e :: \varpi\ \ from\ offset = \texttt{v}}{(\sigma,\varpi) \models \texttt{read\_hard\_output}(where, from, offset, table)\,; \Rightarrow (\circ,\sigma[where \leftarrow v], e :: \varpi)}$$

Listing 13 *– VCP Ada semantics - instruction evaluation inference rules 3*

For the evaluation of user-defined procedure calls, we need three more functions inspired from sparkformal [Aponte et al., 2014, Courtieu et al., 2013]: the `copy_in`, the `init_locals` and the `copy_out` functions. Considering an abstract machine (σ, ϖ) on which a local procedure call has to be evaluated, the `copy_in` function is in charge of creating a new environment for this procedure (setting up the scope, the visibility, initializing parameter variables as local variables). Then this new environment is used by the `init_locals` function to contain the declarations of the local variables of this procedure. Then the instructions of this procedure are evaluated using the output environment of `init_locals`. Finally, `copy_out` is in charge of rebuilding the environment before the procedure call in order to take into account the effect of the called procedure.

The `copy_in` function is in charge of generating a suitable environment for the declaration of local variables and the execution of the instructions of a given local procedure. This function takes the abstract machine at the moment that local procedure has been called (σ, ϖ), the parameter description of that local procedure and the bind parameters of that procedure call. The environment it returns is similar to σ except that its local scope only contains the parameter variables that are bound to the parameters provided by the procedure call. Its description uses the notation $\sigma[\sigma_{local-v} \leftarrow \varepsilon]$ that refers to the environment σ in which local variables are removed. The notation $\sigma_{ld}[a : t \leftarrow v]$ indicates the successful declaration of a non constant local variable called a of type t initialized to the value v in the environment σ and the notation $\sigma_{ld}[a : t]$ indicates the successful declaration of a non-constant and non-initialized local variable a of type t in the environment σ. When t is replaced by t_c, this means that the variable declared is constant. The description of the `copy_in` function is written as follows:

$$\frac{}{\texttt{copy\_in}\ (\sigma,\varpi)\ ()\ ()\ =\ \sigma[\sigma_{local-v} \leftarrow \varepsilon]}$$

$$\frac{\texttt{copy\_in}\ (\sigma,\varpi)\ (name_1 : mode_1\ \ t_1,\ \ \ldots,\ \ name_n : mode_n\ \ t_n)\ \ (e_1,\ \ \ldots,\ \ e_n) = \sigma' \qquad (\sigma,\varpi) \vdash e \mapsto v}{\texttt{copy\_in}\ (\sigma,\varpi)\ (name : \texttt{in}\ \ t,\ \ name_1 : mode_1\ \ t_1,\ \ \ldots,\ \ name_n : mode_n\ \ t_n)\ \ (e,\ \ e_1,\ \ \ldots,\ \ e_n) = \sigma'_{ld}[name : t_c \leftarrow v]}$$

$$\frac{\texttt{copy\_in}\ (\sigma,\varpi)\ (name_1 : mode_1\ \ t_1,\ \ \ldots,\ \ name_n : mode_n\ \ t_n)\ \ (e_1,\ \ \ldots,\ \ e_n) = \sigma' \qquad (\sigma,\varpi) \vdash e \mapsto v}{\texttt{copy\_in}\ (\sigma,\varpi)\ (name : \texttt{in out}\ \ t,\ \ name_1 : mode_1\ \ t_1,\ \ \ldots,\ \ name_n : mode_n\ \ t_n)\ \ (e,\ \ e_1,\ \ \ldots,\ \ e_n) = \sigma'_{ld}[name : t \leftarrow v]}$$

$$\frac{\texttt{copy\_in}\ (\sigma,\varpi)\ (name_1 : mode_1\ \ t_1,\ \ \ldots,\ \ name_n : mode_n\ \ t_n)\ \ (e_1 \ldots,\ \ e_n) = \sigma'}{\texttt{copy\_in}\ (\sigma,\varpi)\ (name : \texttt{out}\ \ t,\ \ name_1 : mode_1\ \ t_1,\ \ \ldots,\ \ name_n : mode_n\ \ t_n)\ \ (e,\ \ e_2,\ \ \ldots,\ \ e_n) = \sigma'_{ld}[name : t]}$$

Listing 14 *– VCP Ada semantics - copy in inference rules*

The `init_locals` function is in charge of declaring the local variables of the called local procedure. To describe the evaluation rules for this function we use the following notations: The notation $\sigma_{ld}[a : t \leftarrow v]$ indicates the successful declaration of a constant local variable called a of type t initialized to the value v in the environment σ and the notation $\sigma_{ld}[a : t]$ indicates the successful declaration of a non constant non-initialized local variable a of type t in the environment σ. Those evaluation rules are written as follows:

$$\frac{(\sigma,\varpi)\vdash e\mapsto v}{\texttt{init\_locals}\ (\sigma,\varpi)\ \ vname\texttt{:constant}\ \ tname\texttt{:=}e\texttt{;} = \sigma_{ld}[vname:tname\leftarrow v]}$$

$$\frac{}{\texttt{init\_locals}\ (\sigma,\varpi)\ \ vname\texttt{:}tname\texttt{;}\ \ = \sigma_{ld}[vname:tname]}$$

$$\frac{(\sigma,\varpi)\vdash e_0\mapsto v \quad \texttt{init\_locals}\ (\sigma_{ld}[vn_0:tn_0\leftarrow v],\varpi)\ \ vn_1\texttt{:}tn_1\texttt{;}\ \ \ldots \quad vn_n\texttt{:}tn_n = \sigma'}{\texttt{init\_locals}\ (\sigma,\varpi)\ \ vn_0\texttt{:constant}\ \ tn_0\texttt{:=}e_0\texttt{;}\ \ vn_1\texttt{:}tn_1\texttt{;}\ \ \ldots\texttt{;}\ \ vn_n\texttt{:}tn_n = \sigma'}$$

$$\frac{\texttt{init\_locals}\ (\sigma_{ld}[vn_0:tn_0],\varpi)\ \ vn_1\texttt{:}tn_1\texttt{;}\ \ \ldots\texttt{;}\ \ vn_n\texttt{:}tn_n = \sigma'}{\texttt{init\_locals}\ (\sigma,\varpi)\ \ vn_0\texttt{:}tn_0\texttt{;}\ \ vn_1\texttt{:}tn_1\texttt{;}\ \ \ldots\texttt{;}\ \ vn_n\texttt{:}tn_n = \sigma'}$$

Listing 15 *– VCP Ada semantics - init-locals inference rules*

The evaluation rules of the `copy_out` function use the following notations: the notation $\sigma[\sigma_{glob-v}\leftarrow a_0,\sigma_{ext-v}\leftarrow a_1]$ refers to the environment σ where the set of global variables is replaced by a_0 and where the set of external variables is replaced by a_1. The notation $s_0\triangleright s_1$ refers to the set of variables s_1 where for all variable for which there is a variable in s_0 with the same name is modified to take the value of the variable in s_0.

$$\frac{}{\texttt{copy\_out}\ \sigma'\ ()\ ()\ \sigma = \sigma[\sigma_{glob-v}\leftarrow\sigma'_{glob-v}\triangleright\sigma_{glob-v},\ \sigma_{ext-variables}\leftarrow\sigma'_{ext-v}\triangleright\sigma_{ext-v}]}$$

$$\frac{\texttt{copy\_out}\ \sigma'\ (name_1\texttt{:}mode_1\ \ type_1\texttt{,}\ \ \ldots\texttt{,}\ \ name_n\texttt{:}mode_n\ \ type_n)\ (e_1\texttt{,}\ \ \ldots\texttt{,}\ \ e_n)\ \ \sigma = \sigma''}{\texttt{copy\_out}\ \sigma'\ (name\texttt{:in}\ type\texttt{,}\ \ name_1\texttt{:}mode_1\ \ type_1\texttt{,}\ \ \ldots\texttt{,}\ \ name_n\texttt{:}mode_n\ \ type_n)\ (e\texttt{,}\ \ e_1\texttt{,}\ \ \ldots\texttt{,}\ \ e_n)\ \ \sigma'' = \sigma'}$$

$$\frac{\texttt{copy\_out}\ \sigma'\ (name_1\texttt{:}mode_1\ \ type_1\texttt{,}\ \ \ldots\texttt{,}\ \ name_n\texttt{:}mode_n\ \ type_n)\ (e_1\ldots\texttt{,}\ \ e_n)\ \ \sigma = \sigma'' \quad \sigma'\vdash_{var} name\mapsto v}{\texttt{copy\_out}\ \sigma'\ (name\texttt{:in out}\ type\texttt{,}\ \ name_1\texttt{:}mode_1\ \ type_1\texttt{,}\ \ \ldots\texttt{,}\ \ name_n\texttt{:}mode_n\ \ type_n)\ (id\texttt{,}\ \ e_1\ldots\texttt{,}\ \ e_n)\ \ \sigma = \sigma''[id\leftarrow v]}$$

$$\frac{\texttt{copy\_out}\ \sigma'\ (name_1\texttt{:}mode_1\ \ type_1\texttt{,}\ \ \ldots\texttt{,}\ \ name_n\texttt{:}mode_n\ \ type_n)\ (e_1\ldots\texttt{,}\ \ e_n)\ \ \sigma = \sigma'' \quad \sigma'\vdash_{var} name\mapsto v}{\texttt{copy\_out}\ \sigma'\ (name\texttt{:out}\ type\texttt{,}\ \ name_1\texttt{:}mode_1\ \ type_1\texttt{,}\ \ \ldots\texttt{,}\ \ name_n\texttt{:}mode_n\ \ type_n)\ (id\texttt{,}\ \ e_1\ldots\texttt{,}\ \ e_n)\ \ \sigma = \sigma''[id\leftarrow v]}$$

Listing 16 *– VCP Ada semantics - copy out inference rules*

To define the semantics of user-defined procedure calls, we will use the following notations: the notation $\sigma\vdash_{pro} pname\mapsto(s,d,i)$ indicates that the environment σ contains a procedure whose name is $pname$, whose parameter definition (or signature) is s, whose local variable declarations are d and whose instructions are i.

Using the `copy_in` function, the `init_locals` function and the `copy_out` function and the semantics of instruction evaluation, we can write procedure evaluation as follows:

$$\frac{\sigma\vdash_{pro} proc\mapsto(s,d,i) \quad \texttt{copy\_in}\ (\sigma,\varpi)\ s\ p = \sigma_1 \quad \texttt{init\_locals}\ (\sigma_1,\varpi)\ d = \sigma_2 \quad (\sigma_2,\varpi)\models i\Rightarrow(\bullet,\sigma_3,\varpi')}{(\sigma,\varpi)\models proc(p)\texttt{;}\Rightarrow(\bullet,\sigma_3,\varpi')}$$

$$\frac{\sigma\vdash_{pro} proc\mapsto(s,d,i) \quad \texttt{copy\_in}\ (\sigma,\varpi)\ s\ p = \sigma_1 \quad \texttt{init\_locals}\ (\sigma_1,\varpi)\ d = \sigma_2 \quad (\sigma_2,\varpi)\models i\Rightarrow(\circ,\sigma_3,\varpi') \quad \texttt{copy\_out}\ \sigma_3\ s\ p\ \sigma = \sigma_4}{(\sigma,\varpi)\models proc(p)\texttt{;}\Rightarrow(\circ,\sigma_4,\varpi')}$$

Listing 17 *– VCP Ada semantics - user defined procedure call inference rules*

These rules explain the process of evaluation of a VCP Ada procedure call by generating a suitable environment for this function (`copy_in` and `init_locals`), then executing the instructions of this procedure in this environment, then in case of success, terminating the evaluation of this procedure call by applying the modification caused by this procedure call to the called environment (`copy_out`). In case of error, the evaluation stops immediately.

Those rules simulate correctly variable passing, but could lead to surprising results in case of aliasing citegellerich2001parameter. However, B0 to VCP Ada translators are designed to detect and fail for those cases.

The Ada visibility policy implies that a program cannot access the definition of an external procedure. To reflect this behaviour, all that data can be retrieved from an external procedure is its signature. For the semantics of those external functions or procedures, we will generalize the technique used in the semantics of the `read_hard_output`. We assume there exists an abstract function called `run_external`, that for any external procedure, is in charge of generating a suitable environment for that procedure, declaring the local variables of that procedure, interpreting the instructions of that procedure and rearranging the names of variables – for example, a variable named a defined in a package definition p is designated by $p.a$ outside p but by a inside p. Then we can use `copy_out` and obtain the simulation of the call of the called external procedure.

To describe the behaviour of external procedure calls, we can use the notation $\sigma \vdash_{sig} proc \mapsto s$ that indicates that the name $proc$ is associated with an external procedure whose signature is s.

The semantics of external procedure calls can be described using the following rules:

$$\frac{\sigma \vdash_{sig} proc \mapsto s \quad \texttt{make\_event}(\sigma, proc, \texttt{p}) = e \quad \texttt{run\_external}\ \ (\sigma, e :: \varpi)\ \ proc\ \ p = (\bullet, \sigma_1, \varpi') }{(\sigma, \varpi) \models proc(p)\,; \Rightarrow (\bullet, \sigma_1, \varpi')}$$

$$\frac{\sigma \vdash_{sig} proc \mapsto s \quad \texttt{make\_event}(\sigma, proc, p) = e \quad \texttt{runExternal}\ \ (\sigma, e :: \varpi)\ \ proc\ \ p = (\circ, \sigma_1, \varpi') \quad \texttt{copy\_out}\ \ \sigma_1\ s\ p\ \sigma = \sigma_2 }{(\sigma, \varpi) \models proc(p)\,; \Rightarrow (\circ, \sigma_2, \varpi')}$$

Listing 18 *– VCP Ada semantics - external procedure call inference rules*

The missing declaration problems can lead to a situation in which the name of a procedure is associated with no procedure signatures ($\sigma \vdash_{sig} proc \mapsto \varnothing$). In that case, we use a variant of `copy_out` called `copyOutExternalWithoutSig`. This variant performs the same action as `copy_out` except that all the call parameters that are made up of a unique variable name are considered as call-by-reference (in out) parameters. Similarly, we assume that the abstract function `run_external` that takes in account the difference between known and unknown functions.

6-III.C Variables and procedures

VCP Ada allows to declare variables in procedure definitions, package declarations and package definitions. In procedure definitions, variables are declared in the local scope. But in other cases, variables are declared using the notation $\sigma_{gd}[a : t \leftarrow v]$ to indicate the successful declaration of a constant global variable called a of type t initialized to the value v in the environment σ and the notation $\sigma_{gd}[a : t]$ indicates the successful declaration of a non constant non-initialized global variable a of type t in the environment σ.

Types are declared and defined in package declarations and package definitions. The notation $\sigma_{type}[name_0 \mapsto name_1]$ indicates that the type $name_0$ is renamed to the type $name_1$. This allow to describe the subtype declaration. The notation $\sigma_{type}[name \leftarrow def]$ indicates the creation of a type

whose name is $name$ and definition is def. The notation $\mathbb{A}_{(n)}(t)$ refers to the type of a 1D t array of size n. The notation $\mathbb{A}_{(n_0,n_1)}(t)$ refers to the type of a 2D t array of size $n_1 \times n_2$. The notation $\Re(d)$ refers to the type of a record whose definition (field names and types) are defined by d.

The semantics of variables declarations can be written as follows:

$$\frac{(\sigma,\varpi) \vdash e \mapsto v}{(\sigma,\varpi) \models vname \texttt{ :constant } tname \texttt{ := } e \Rightarrow (\sigma_{gd}[vname : tname \leftarrow v],\varpi)}$$

$$\frac{}{(\sigma,\varpi) \models vname \texttt{ : } tname; \Rightarrow (\sigma_{gd}[vname : tname],\varpi)}$$

$$\frac{}{(\sigma,\varpi) \models \texttt{type } rname \texttt{ is record } rdef \texttt{ end record;} \Rightarrow (\sigma_{type}[rname \leftarrow \Re(rdef)],\varpi)}$$

$$\frac{}{(\sigma,\varpi) \models \texttt{subtype } name \texttt{ is } tname\texttt{;} \Rightarrow (\sigma_{type}[name \mapsto tname],\varpi)}$$

$$\frac{(\sigma,\varpi) \vdash e \mapsto n}{(\sigma,\varpi) \models \texttt{subtype } name \texttt{ is } tname\texttt{(}e\texttt{) ;} \Rightarrow (\sigma_{type}[name \mapsto \mathbb{A}_{(n)}(tname)],\varpi)} \; [n \in \mathbb{N}]$$

$$\frac{(\sigma,\varpi) \vdash e \mapsto n \qquad (\sigma,\varpi) \vdash e' \mapsto n'}{(\sigma,\varpi) \models \texttt{subtype } name \texttt{ is } tname\texttt{(}e\texttt{, } e'\texttt{) ;} \Rightarrow (\sigma_{type}[name \mapsto \mathbb{A}_{(n,n')}(tname)],\varpi)} \; [n,n' \in \mathbb{N}]$$

Listing 19 *– VCP Ada semantics - variable and type declaration inference rules*

Procedures are declared in a package declaration and a package definition. Procedures can only be defined in package definition or procedure definition files. The notation $\sigma_{pdec}[n \leftarrow sig]$ is used to indicate the successful declaration of a local procedure whose name is n and parameter definition is sig in the environment σ. The notation $\sigma_{pdef}[n \leftarrow (s,d,i)]$ is used to indicate the successful definition of a local procedure whose name is n, parameter definition is s, procedure variable definition is d and instructions are i in the environment σ.

The semantics of procedure declarations and definitions can be written as follows:

$$\frac{}{(\sigma,\varpi) \models \texttt{procedure } fn(args); \Rightarrow (\sigma_{pdec}[fn \leftarrow args],\varpi)}$$

$$\frac{}{(\sigma,\varpi) \models \texttt{procedure } fn\texttt{(}s\texttt{) is } d \texttt{ begin } i \texttt{ end;} \Rightarrow (\circ,\sigma_{pdef}[fn \leftarrow (s,d,i)],\varpi)}$$

$$\frac{(\sigma,\varpi) \models \texttt{procedure } fn\texttt{(}p\texttt{:}m\ t\texttt{, } p_1\texttt{:}m_1\ t_1\texttt{, ..., } p_n\texttt{:}m_n\ t_n\texttt{) is begin } n\texttt{(}p\texttt{, } p_1\texttt{, ..., } p_n\texttt{) end;} \Rightarrow (\sigma',\varpi)}{(\sigma,\varpi) \models \texttt{procedure } fn\texttt{(}p\texttt{:}m\ t\texttt{, } p_1\texttt{:}m_1\ t_1\texttt{, ..., } p_n\texttt{:}m_n\ t_n\texttt{) renames } n\texttt{;} \Rightarrow (\sigma',\varpi)}$$

Listing 20 *– VCP Ada semantics - procedure declaration and definition inference rules*

6-III.D Compilation units

VCP Ada source code, also called compilation unit, is made up of two parts: the import part (`with` statements) and the unit (the package declaration, definition or procedure definition). The semantics of `with` statements allows to make external variables and procedures visible using the file inclusion process. The interpretation of the instruction `with` p can be described as follows:

1. **finding the included file:** the file inclusion process starts by finding the package declaration file f of the package p. The base-name of f is p and its extension is either `.ads` or `.1.adb`. The next step consists in parsing f and transforming it into an abstract syntax tree a. As f is a package declaration file, a contains only variable, type and procedure declarations.

2. **building a start-environment to analyse the included file:** To analyse the declaration contained in a, the file inclusion process uses the inference rules described in section 6-III.C. The use of those inference rules requires an abstract machine. The abstract machine that we will use is made up with an empty trace and an environment that is built as follows:
   - If a contains no `with` statements, then the used environment is the empty environment (an environment that contains no variables and no procedures)
   - Else, we perform recursively the interpretation of `with` statements. Those interpretations return sets of external variables and procedures. In that case, the environment of the abstract machine will be an environment that contains only those external variables and procedures.
3. **extracting the external elements:** Step 2 provides us with an abstract machine whose environment contains global variables and procedure declarations. Those global variables and procedure declarations are the elements that are made visible by that `with` p statement.

The semantics of file inclusion provides us with the set of external variables and procedures. From an environment that contains only those external variables and procedures, the interpretation of the unit of the source code is done as follows:

- package declaration are evaluated as a set of variable and procedure declarations. Their interpretation uses the inference rules described in Listings 19 on page 77 and 20 on page 77.
- package definitions are evaluated as a set of variable and procedure declarations and definition and their interpretation uses the inference rules described in Listings 19 on page 77 and 20 on page 77.
- procedure packages are evaluated as procedure definitions and their interpretation uses the inference rules described in Listing 20 on page 77.

We make some final remarks about the semantics of compilation units:

- Regarding the rules that are involved in the evaluation of a compilation unit and the syntax of VCP Ada, the trace is neither used nor modified. So, the evaluation of a compilation unit can be seen as a relation that links an input environment and an input compilation unit to an output environment.
- Due to the visibility rules described in Section 6-I, external variables and procedures are made visible either in a file by the file inclusion process or by the evaluation of the unit of that file. So, the input environment of the evaluation of a compilation unit is the empty environment (an environment that contains no variables and no procedures). Consequently, the evaluation of a compilation unit can be seen as a relation that links an input compilation unit to an output environment.

6-IV The Coq implementation of the VCP Ada semantics

This section focuses on the implementation in Coq of the abstract machine and the semantics of VCP Ada described the previous sections of this chapter.

6-IV.A The Coq implementation of the abstract machine

The abstract machine is made up of 2 elements: the environment and the trace. This pair is implemented using the Coq native implementation of pairs (see Coq.Init.Datatypes). The implementation of those data structures uses the notion of map and value.

6-IV.A.1 Map data structure

A *map* is a data structure that associates an item called *key*, to an other item called *value*. This data structure is one of the main components of the environment data structure. Its 3 main operators are:

- `set`: the *set* operation associates a key to a value in a map. This function takes key k, a value v and a map and returns another map in which k is associated with v.
- `update`: the *update* is similar to the *set* operation except that it returns an option type. This return value is `None` if there already exists a value associated with k in the input map.
- `get`: the *get* to retrieve the value associated with key in a map. This function takes a map and a key then returns an option value (None if no value is associated with the key)

We started the implementation of the map data structure by providing a specification (abstract signature) of the operations described above using a Coq module type. Then, we provided a concrete implementation and associated proof of this module type using a list of pairs similarly to *OCaml list assoc*. Each element of this list is a pair that associates a *key* (stored as the first member of the pair) with a *value* (stored as the second member of the pair). We used this implementation because it is simple to understand and easy to prove.

Later in this document, we will use the notation `<a, b> map` to indicate the type of a map which key type is `a` and value type is `b`.

6-IV.A.2 Value data structure

The 6 categories of values described in Section 6-II.A are implemented using the data structure (`value`) that is described as follows:

1. **Integers:** Integer values represent (vital and non-vital) integers in VCP Ada and are implemented using the $\mathbb{Z}$ set implemented in Coq (see Coq.ZArith).
2. **Booleans:** Boolean values represent (vital and non-vital) booleans in VCP Ada and are implemented using the native booleans implemented in Coq (see Coq.Bool).
3. **Strings:** String values represent strings in VCP Ada and are implemented using the native strings implemented in Coq.
4. **Arrays:** Array values represent arrays in VCP Ada and are implemented using the type `<nat, value> map * value` that represents a pair. The first member of this pair is a map that associates the indexes, implemented using the Coq implementation of natural numbers (see Coq.Init.Nat), of items to their values. The second member of this map represents the default value of the array. This element is useful to implement the `init` procedure.
5. **Records:** Record values represent records in VCP Ada and are implemented using `<string, value> map`. This map associates the identifiers of record fields to those record field values.
6. **Undefined:** Undefined values represent the value of a non-initialized variable. This is also used to indicate the value of variables which declarations are missing.

This description can be implemented using the following Coq code:

```
Inductive value: Type :=
  | Int : Z -> value
  | Bool : bool -> value
  | STR : string -> value
  | ArrayV : NatMapModule.t value (* equivalent to <nat, value> map *) -> value -> value
  | RecordV : StringMapModule.t value (* equivalent to <string, value> map *) -> value
  | Undefined : value.
```

***Listing 21** – VCP Ada semantics Coq implementation - Definition of values*

Example: A record containing three fields `a` containing the integer value `1`, `b` containing the boolean value `true` and `c` containing a string value `''hello''` is coded by `RecordV [(''a'', Int 1); (''b'', Bool 1); (''c'', STR ''hello'')]`. A 1D-array containing the integers 0, 2, 4 and 6 can be implemented using `ArrayV [(0, Int 0); (1, Int 2); (2, Int 4); (3, Int 6)] Undefined` and a 2D-array containing 0, 1, 2 and 3 at indexes (0, 0), (0, 1), (1, 0) and (1, 1) can be implemented using `ArrayV [(0, ArrayV [(0, Int 0); (1, Int 1)]); (1, ArrayV [(0, Int 2); (1, Int 3)])] Undefined`.

This value model is inspired from work performed in *formal semantics research around Spark2014* (Sparkformal) [Aponte et al., 2014, Courtieu et al., 2013]. This work comes with associated functions that simulate usual arithmetic and boolean operators. There exist two main differences between our model and the work presented in sparkformal:

variable names: Our model uses `string` to refer to variable names while in sparkformal, those variable names are indicated using `nat` (natural numbers).

simulating usual arithmetic and boolean operators: The functions that simulate usual arithmetic and boolean operators return option values. `None` means that a problem such as type error occurred during the evaluation of operation. In our model as in Compcert, `Undefined` is an *absorbing element*. This means for example that `Undefined + 5 = Some Undefined` while in sparkformal `Undefined + 5 = None`. This choice is a consequence of the missing-definition problem (defined in Section 6-I). In fact, when a declaration of a variable is missing, the value of this variable is evaluated to `undefined`. So, when this value is used in a expression, the value of this expression misses information to be completely computed. However, in our model, when a type error is detected, the result is `None`. For example `Undefined + True = None`.

Sparkformal work also provides functions to set and get fields in records – `record_select` and `recordUpdate` – and items in arrays – `array_select` and `arrayUpdate`. The equivalents of those functions in our source code is `ids_select_from_value_f`, `ids_set_from_value_f`, `idxs_select_from_value_f` and `idxs_set_from_value_f`. The two versions are similar, except that Sparkformal uses natural numbers as identifiers while our work uses strings.

Remark: When `set_id_record` or `set_idx_array` is executed in order to set the value of a non-existing field in a record or index in an array leads to the creation of the field or index in question. For example, setting the string value "yes" at index 15 in the array given in the previous example leads to the array `ArrayV [(0, Int 0); (1, Int 2); (2, Int 4); (3, Int 6); (15, STR ''yes'')]`. A similar processing is made for `Undefined` values. For example, `ids_set_from_value_f (Int 5) ["f"] Undefined = Some (RecordV [("f", Int 5)])`.

This implementation choice, shared with Sparkformal, is compatible with the missing declaration problem mentioned Section 6-I. In fact, as the processed source code comes from B source code translation, we assume that they are correct and must not contain source code errors such as trying to set or to get the value of a non-existing field.

To make the distinction between constant and non-constant values, we encapsulate values in a data structure that will indicate whether or not the encapsulated value is constant or not. The Coq code of this data structure is written as follows:

```
Inductive ConstOrVar :=
| ConstantValue : value -> ConstOrVar
| VariableValue : value -> ConstOrVar.
```

Listing 22 *– VCP Ada semantics Coq implementation - Definition of constant or non-constant values*

6-IV.A.3 Environment data structure

The environment is implemented in form of a record that represents the scopes. This environment implementation uses the data structures such as `element_declararif` and `(list (ident * mode * nom))` that are used to represent procedure definition and signature during the syntax analysis. The source code associated with this environment representation is written as follows:

```
Inductive External :=
| EVar : ConstOrVar -> External
| EFDecl : (list (String_Equality.t * mode * name)) -> External.

Record env := mkenv
{
  localVars : StringMapModule.t ConstOrVar
; globalVars : StringMapModule.t ConstOrVar
; funDefs : StringMapModule.t element_declararif
; localfunDecls : StringMapModule.t (list (ident * mode * name))
; externals : StringMapModule.t (StringMapModule.t External)
}.
```

Listing 23 *– VCP Ada semantics Coq implementation - Definition of environment*

This environment comes with functions to get and set values from identifiers (`getIdValue` and `setIdValue`) or selected components (`getIdsValue` and `setIdsValue`). All those functions have been defined by pattern-matching (see Section 2-I.B.2 on page 14), to simplify proofs and understandings. By construction, those definitions respect the hierarchy of scopes – names are first evaluated in local scope, then global and external scope – the rules for the evaluation of selected components described in Section 6-III.A and the fact that the modification of a constant variable fails. Those get and set functions return Coq options (`None` in case of failure). Those get and set functions come with formal proofs of their correctness. The most important of those proofs is the following: for all selected component s, environments σ and σ' and value v, if `setIdsValue` s v e = `Some` e', then `getIdsValue` s e' = `Some (Value (VariableValue` v`))`. Using the notations described in Section 6-III, we can rewrite it as follows: for all variable name s, environment σ, and value v, $\sigma[s \leftarrow v] \vdash_{var} s \mapsto v$.

We also implemented functions for variable declaration (`declareLocalVar`, `declareGlobalVar`), procedure definition (`defineFunction`) and declaration (`declareFunction`). The following table shows the correspondence between the Coq code and the formal semantics notations used to describe the semantics.

Description	Formal semantics notation	Coq notation
The value `undefined`	$\varkappa$	`Undefined`
The variable name id is associated to the value constant value v in the environment σ	$\sigma \vdash_{var} id \mapsto v$	`getIdsValue` id σ `= Some (Value (ConstantValue` v`))`
The variable name id is associated to the value variable value v in the environment σ	$\sigma \vdash_{var} id \mapsto v$	`getIdsValue` id σ `= Some (Value (VariableValue` v`))`
The procedure name id is associated to the signature s in the environment σ	$\sigma \vdash_{sig} id \mapsto s$	`getIdsValue` id σ `= Some (ExternalFunction` s`)`
The procedure name id is associated to the procedure definition f in the environment σ where s, d and i are the parameter definition the local variable declarations and the instructions of f	$\sigma \vdash_{sig} id \mapsto (s,d,i)$	`set_idxs_array` id σ `= Some (DefinedFunction` f`)`
Accessing arrays for all natural numbers i_1 and i_2	$t[i_1]$	`idxs_get_from_value_f` t `[`i_1`]`
	$t[i_1, i_2]$	`idxs_get_from_value_f` t `[`i_1`; `i_2`]`
Assigning value v to variable var in the environment σ leading to environment σ'	$\sigma[var \leftarrow v] = \sigma'$	`setIdsValue` var v σ `= Some` σ'
Declaring a local constant variable var of type t initialized with v in the environment σ leading to the environment σ'	$\sigma_{ld}[var : t \leftarrow v] = \sigma'$	`declareLocalVar` var `(ConstantValue` v`)` σ `= Some` σ'
Declaring a local non-constant variable var of type t in the environment σ leading to the environment σ'	$\sigma_{ld}[var : t] = \sigma'$	`declareLocalVar` var `(VariableValue` $\varkappa$`)` σ `= Some` σ'
Declaring a global constant variable var of type t initialized with v in the environment σ leading to the environment σ'	$\sigma_{gd}[var : t \leftarrow v] = \sigma'$	`declareGlobalVar` var `(ConstantValue` v`)` σ `= Some` σ'
Declaring a global non-constant variable var of type t in the environment σ leading to the environment σ'	$\sigma_{gd}[var : t] = \sigma'$	`declareGlobalVar` var `(VariableValue` $\varkappa$`)` σ `= Some` σ'
Declaring a local procedure f with signature s in the environment σ leading to the environment σ'	$\sigma_{pdec}[f \leftarrow s] = \sigma'$	`declareFunction` f s σ `= Some` σ'
Defining a local procedure f whose signature is s, local declarations are d and instructions are i, (contained in a the `element_declaratif` data-type e) in the environment σ leading to the environment σ'	$\sigma_{pdef}[f \leftarrow (s,d,i)] = \sigma'$	`defineFunction` e σ `= Some` σ'

Table 6.1 *– VCP Ada semantics Coq implementation - Correspondence between the Coq code and formal semantics notations*

Remark: In our implementation, types of variables and type declarations are ignored. Let us illustrate the impact of this decision using the following example:

a.ads

```
1  package a is
2  type rec is record
3          r0 : SINT;
4          r1 : SBOOL;
5  end record;
6  procedure p( r: in out rec);
7  end a;
```

a.adb

```
1  package body a is
2          procedure p( r: out rec) is
3                  t : SINT_ARRAY(2);
4                  i : SINT;
5                  v : SINT;
6          begin
7                  init(i, 0);
8                  init(v, 0);
9                  write(t, i, v);
10                 i := assign(1)
11                 v := assign(4);
12                 write(t, i, v);
13                 init(r.r0, v);
14                 init(r.r1, True);
15         end p;
16 end a;
```

Listing 24 *– Ignoring type of variable and type declaration example*

Now let us compare how the value of the variable a would be represented during the execution of the procedure p when the type of a is ignored or not.

Line number	Representation of a if typing is not ignored	Representation of a if typing is ignored
7-12	`RecordV [("r0", Undefined); ("r1", Undefined)]`	`Undefined`
13	`RecordV [("r0", Int 4); ("r1", Undefined)]`	`RecordV [("r0", Int 4)]`
14	`RecordV [("r0", Int 4); ("r1", Bool true)]`	`RecordV [("r0", Int 4); ("r1", Bool true)]`

Table 6.2 *– Observing the impact of variable type ignoring*

By construction, our semantics considers those representations are equivalent. This equivalence is a consequence of the missing declaration problem. So, ignoring the type declaration and the type of variables would have no impacts on the evaluation of expressions in our semantics. Another reason to handle types is type checking. In our case, we assume that the input source code contains no type error because of the way it is generated (see chapter 3). And in case there are, we just propagate those errors that will be detected at compile time.

6-IV.A.4 The abstract machine trace

The abstract machine trace is implemented as a list (see Coq.Init.Datatypes) of events. An event is a record that contains the name of the external procedure – a selected component stored in the field `fun_name` – the signature of the called procedure – an optional list of parameter descriptions stored in the field `fun_sig` – the list of call parameter (stored in the field `fun_call_params`), the list of the values of those parameters (stored in the field `fun_call_params_eval`) and the environment without the at the moment this procedure is called. The Coq description of the event data structure is written as follows:

```
Record event:= mkEvent {
  fun_name: selected_com
; fun_call_params: list expression_or_string_lit
; fun_call_params_eval: list value
; fun_sig: option (list (ident * mode * name))
; env_at_call_wo_locals: env
}.
```

Listing 25 *– VCP Ada semantics Coq implementation - Definition of events*

6-IV.B The Coq implementation of evaluation rules

The Coq inductive relations allow a simple translation of inference rules. Let us illustrate this translation process for the inference rules presented in Listing 8 that present the evaluation of a VCP Ada expression (see `<expression>` in Section 3-I.B.3.1) on an abstract machine.

This translation process starts with the typing of the inductive property. For this translation, we used the type `env -> expression -> value -> Prop`. This means that the inductive relation we are about to describe will put in relation an environment (`env`), a syntactic expression (`<expression>`) and a value (`value` defined in Listing 21). The definition of `<expression>` can be written as follows:

```
(* Definition of expressions *)
Inductive literal : Type :=
| Int_lit : Z -> literal
| Bool_lit : bool -> literal.

Inductive expression : Type :=
| EVar : nom -> expression
| ELit : literal -> expression
| Ebinary : binary_op -> expression -> expression -> expression
| Eunary : unary_op -> expression -> expression.
```

Listing 26 *– VCP Ada semantics Coq implementation - Coq representation of `<expression>`*

The translation of the inference is also intuitive. For example, the translation of the inference rules presented in Listing 8 on page 70 is described in Listing 27 on page 90. Table 6.3 gives the correspondence between the inference rules presented in this chapter and their name in the Coq implementation.

Remark: For the translation presented in Listing 27, we decided to use the environment (σ) instead of the abstract machine (σ, ϖ) to compute the evaluation of expressions because only the environment intervenes in those evaluation rules.

We implemented the abstract function such as `is_valid` presented in Listing 10 and the file inclusion presented in section 6-III.D using Coq Parameter functions. The `make_event` function is implemented using a Coq parameter function.

Reference in the semantics section	Reference in Coq code	Additional information
Listing 9 and 10	`evalSec_opel_expression_if` `evalSec_opel_expression_d_affectation` `evalSec_opel_w`	Describes how to evaluate <if-cond>, <assign-exp> and <while-cond>. The last two uses σ instead of (σ, ϖ).
Listing 11	`evalInstruction`, `evalInstruction_s`, `evalElif_instruction` and `evalCase_instruction`	Describes how to evaluate <inst>, <insts>, <else-if> and <case-v>.
Listing 12, 13, 17 and 18	`evalInstruction`	<proc-call>
Listing 19 and 20	`evalDeclaration_de_baseList`	Describes how to evaluate <var-decl>, <subtype-decl>, <record-decl>, <proc-decl> and <proc-def>

Table 6.3 *– VCP Ada semantics Coq implementation - Correspondence between the semantics definition and the Coq implementation*

6-V Semantic preservation

Semantic preservation defines the correctness of an optimization (source code transformation). It states that an optimizer is correct if the transformations it performs do not modify the behaviour of programs. The missing declaration problem described in Section 6-I limits transformation possibilities. For example, this problem prevents the removal and the renaming of objects (variable, procedure or type) declared in a declaration file. In fact, when an object is declared in a declaration, this object can be accessed by a file that is missing or that can not be processed and so that is not modifiable. As a consequence, the removal or the renaming of any object can lead to compilation errors, because those files would make reference to objects that do not exist anymore.

More generally, the modifications we aim to implement will have the following properties:

- No package object creation: Creating objects comes with the difficulty of creating unique strings that will be used to identify those objects in a package. In addition to that, the missing declaration problems could also lead to additional problems. A typical problem would be the creation of a procedure in a package definition associated with an inaccessible package declaration (due to the missing declaration problem) that contains a rename procedure (see <proc-renm> in Section 3-I.B.3.3 on page 36) with the same name. In fact, those kinds of procedures do not appear in the package definition. As this package definition can be inaccessible, we would not be able to ensure the absence of declaration conflicts (declaring two objects with the same name).
- No package object removal: This restriction is due to the missing declaration problem. In fact, removing an object could lead to compilation problems because that removed object could be used by another file that is not accessible.
- No package object renaming: This restriction is due to the same reasons as the object creation problem.

Remark: Those restrictions forbid the implementation of optimizations such as common sub-expression elimination (CSE) [Resler and Winter, 2009] that may imply an object (variable) creation. However, the syntax of VCP Ada and particularly the form of assign expression (see <assign-exp> in Section 3-I.B.3.1 on page 32) leads developers to perform that optimization during the development.

The transformation performed by the optimizers we aim to build will be focussed on the modification of instructions (see <insts> in Section 3-I.B.3.2 on page 35) and the simplification of expressions (see <cond-exp> and <assign-exp> in Section 3-I.B.3) and the addition of pragma declaration such as the inline pragma. Consequently, there is no modification of other elements such as function signatures or import statements. So, we can define semantic preservation as follows:

For a compilation unit a whose optimization leads to b, we say that a have the same behaviour as b, if all the condition cited below are verified:

- The type of b must be the same as a. This means that if a is a package declaration source code, then b should be too. Similarly, if a is a package definition or procedure definition source code, then b should be too.
- The package name in package declaration and package definition files must be the same. In procedure definition files, the name of the procedure must be the same.
- For any global variable that is declared or defined in a, there must be a variable in b with the same name, same type, same scope and same definition.
- For any procedure that is declared or defined in a, there must be an equivalent procedure (as defined hereafter) in b.
- Pragmas, type declarations, type definitions and the type of variables are conserved after opt. This means that for any pragma, type declaration, type definition and the type of variables present in a, are the same in b.

The equivalence of procedures defines when two given procedures have the same behaviour. To define that equivalence, we need to define two elements: the behaviour of a procedure execution and the runtime environment.

1. The inference rules used to evaluate the execution of a procedure (s,d,i) – where s, d and i are respectively the signature, the set of local variable declaration and the sequence of instructions of that procedure – with the set of parameters p on an abstract machine (σ,ϖ) noted $\texttt{run}_p^{(s,d,i)}(\sigma,\varpi)$ are described as follows:

$$\frac{\texttt{copy\_in}\ \ (\sigma,\varpi)\ s\ p = \sigma_1 \qquad \texttt{init\_locals}\ \ (\sigma_1,\varpi)\ d = \sigma_2 \qquad (\sigma_2,\varpi) \models i \Rightarrow (\bullet,\sigma_3,\varpi')}{\texttt{run}_p^{(s,d,i)}(\sigma,\varpi) = (\bullet,\sigma_3,\varpi')}$$

$$\frac{\texttt{copy\_in}\ \ (\sigma,\varpi)\ s\ p = \sigma_1 \qquad \texttt{init\_locals}\ \ (\sigma_1,\varpi)\ d = \sigma_2 \qquad (\sigma_2,\varpi) \models i \Rightarrow (\circ,\sigma_3,\varpi') \qquad \texttt{copy\_out}\ \ \sigma_3\ s\ p\ \sigma = \sigma_4}{\texttt{run}_p^{(s,d,i)}(\sigma,\varpi) = (\circ,\sigma_4,\varpi')}$$

Listing 28 *– VCP Ada semantics equivalence - procedure execution*

2. The runtime environment is the environment during the execution of the program. As the exact values of variables cannot be determined in advance, we will use an abstract relation (parameter relation) that we call `is_runtime_of` to put in relation an environment at initialization and its value at runtime. Chapter 7 proposes a more precise description of that relation.
3. As pragmas and types are ignored by our abstract machine model (see Section 6-IV.A.3 on page 81), we need to define inference rules to ensure that variable types, pramas, type declarations and definitions present are preserved. For that, we defined a type called `declaration` that is used to store pragmas, variable types, type declarations and definitions. Then we implemented functions (`extract_declarations`) to extract such declarations (pragmas, variable declaration types, type declarations and definitions) from package declarations and package body and store them in form

of a list of `declaration` type. Then, using those lists, we defined the inference rules to ensure conservation as follows:

$$\frac{}{\texttt{type\_conserve}\ []\ []} \qquad \frac{h = h' \quad \texttt{type\_conserve}\ t\ t'}{\texttt{type\_conserve}\ (h :: t)\ (h' :: t')}$$

$$\frac{\texttt{type\_conserve}\ (h :: t)\ t'}{\texttt{type\_conserve}\ (h :: t)\ ((\texttt{pragma}\ id\ (arg_1, arg_2, \ldots, arg_n)) :: t')}$$

Listing 29 *– VCP Ada semantics equivalence - pragma and type conservation*

The notation `type_conserve` $a\ b$ means that the declaration list a, extracted from the original code, is preserved regarding the declaration list b extracted from the optimized code. Those inference rules ensure that all the elements present in a are present in b in the same order. The list b can contain additional elements only if those elements are pragmas. This possibility allows optimizers to perform optimizations using Ada pragmas.

To define the semantic preservation of a source code optimizer (a function that takes a source code and returns a source code), we will use the notations σ_{loc-v}, σ_{glob-v} to refer to the sets of local variables and global variables of the environment σ. The notation $\sigma_{pack}[p]$ refers to set of external elements associated with the package p in the environment σ. For a given set of local or global variables or a set of external elements s, the notation $s \overset{id}{\longmapsto} v$ means that in s, there is an element whose name is id and is bound to the value v. For example, $\sigma_{loc-v} \overset{id}{\longmapsto} v$ means that there is a local variable whose name is id and whose value is v in the environment σ. The notation σ `= compile` a means that the evaluation of the source code a leads to the environment σ. The notation `is_runtime_of` $\sigma_0\ \sigma_1$ means that the environment σ_1 is a possible runtime environment of the environment σ_0. The notation `compile` a `=` σ to indicate that the evaluation of the compilation unit (described in the Section 6-III.D) a leads to the environment σ.

Knowing the limitations of the optimizers we aim to build, we define semantic preservation as follows:

Definition 1 (Semantic preservation)**.** *Let f be a source code optimizer, two source code Ast a and b such that $b = f\,a$, σ and σ' two environments such that $\sigma =$ `compile` a and σ' `= compile` b. f is* correct *if all the conditions cited below are verified:*

1. *the package name of a is the same as the one of b*
2. *the package type of a is the same as the one of b*
3. *the declarations of a are preserved. This means that* `type_conserve (extract_declarations a) (extract_declarations (f a))`
4. *for every variable name n and value v, if $\sigma_{loc-v} \overset{n}{\longmapsto} v$, then $\sigma'_{loc-v} \overset{n}{\longmapsto} v$*
5. *for every variable name n and value v, if $\sigma_{glob-v} \overset{n}{\longmapsto} v$, then $\sigma'_{glob-v} \overset{n}{\longmapsto} v$*
6. *for every package name p, external variable name n and variable value v, if $\sigma_{pack}[p] \overset{n}{\longmapsto} v$, then $\sigma'_{pack}[p] \overset{n}{\longmapsto} v$*
7. *for every declared but not defined procedure name p and procedure signature s, if $\sigma \vdash_{sig} p \mapsto s$ then $\sigma' \vdash_{sig} p \mapsto s$*

8. *for every defined procedure name p and procedure definitions* (s,d,i) *and* (s',d',i')*, if* $\sigma \vdash_{pro} p \mapsto (s,d,i)$ *then* $\sigma' \vdash_{pro} p \mapsto (s',d',i')$ *and for all parameters* ρ*, environment* σ_r*, trace* ϖ *and instruction execution result r, if* $(is\_runtime\_of\ \ \sigma\ \ \sigma_r)$ *and* $run_{\rho}^{(s,d,i)}(\sigma_r,\varpi) = r$ *then* $run_{\rho}^{(s',d',i')}(\sigma_r,\varpi) = r$

Let us illustrate the definition of semantic preservation with an example.

a.ads (original)

```
1 package a is
2   a1 : constant FINT := 5;
3   a2 : constant FINT := a1 + 2;
4   uselessvariable : SINT;
5   type unusedtype is record
6           f1: SINT;
7           f2: SINT;
8    end record
9 end package;
```

a.ads (optimized)

```
1 package a is
2   a1 : constant FINT := 5;
3   a2 : constant FINT := 7;
4 end package;
```

Listing 30 *– Semantic preservation non-example*

This example supposes that the variable whose name is `uselessvariable` is unused in the project. Consequently, it has been removed. Regarding our semantic preservation rules, this optimization is not valid. In fact, the environment of the original a.ads file has a variable that the environment of the optimized version of a.ads does not have. There is a conflict with rules 4 and 5 of the definition of semantic preservation. As a reminder, this rule is a direct consequence of the missing declaration problem. In fact, the missing declaration problem makes impossible to prove that this variable is not used in a file that is not accessible. Another reason is that our model performs the analysis on one file at the time. The analysis of a given file only give access to the package declaration that file depends on. So, from a.ads, we cannot have access to the files that depend on a.ads. This also makes impossible to prove that `uselessvariable` is unused.

Similarly, our definition forbids the removal of `unusedtype` declaration. The mechanism used to ensure that conservation is explained in rule 5 above.

The Coq implementation details of the semantic preservation between a (the original source code) and b (the optimized version of a) can be written as follows:

1. **Step 1: static element analysis.** From the abstract syntax of a code source, we can extract the package name of that source code, its package types and its declarations. We know that optimization must not change the package name and type (see rules 1 and 2 in the definition of semantic preservation). We also know that the declarations of a must be conserved in b.

   Step 1 consists of those verifications.

2. **Step 2: source code evaluation.**

   (a) starting from an empty abstract machine – an abstract machine that consists of an environment that contains no objects and an empty trace (a trace that contains no events), we run the preprocessor to obtain an abstract machine that contains all the external declarations (declarations made in other files) and if the source code that is analysed is a package definition, the output abstract machine also contains the global declarations made in the associated package declaration.

   (b) then from that output abstract machine, we can evaluate the declarations and definitions made in the file to analyse. The result machine coming from this evaluation contains an empty

trace because no function calls have been evaluated during this process. For all file f, we will use the notation σ^f to refer to the environment that results from this processing on file f.

Step 2 consists in the evaluations of a and b. This step is very similar to the semantics of the evaluation of compilation units (source code) described in Section 6-III.D and implemented using the Coq version of those inference rules.

3. **Step 3 : equivalence of a and b.** The equivalence between the source code a and its optimized version b consist in making sure that σ^a and σ^b, the environments that result from the evaluation of a and b, contain equivalent elements.

   To implement this equivalence in Coq, we first defined the notion of equivalence at map level (see Section 6-IV.A.1 on page 79). This definition says two maps m_1 and m_2 are equivalent if any k is associated with a value v in m_1 if and only if k is associated with the value v in m_2. Then this equivalence, noted `StringMapModule.map_eq` for maps whose key type is the Coq native `string`, is used to define the equivalence of environments. Two environments σ^a and σ^b are equivalent if the conditions below are verified:

   - the localVars, globalVars and localfunDecls maps (see the Coq definition of environments in Listing 23 on page 81) of σ^a are equivalent to the the localVars, globalVars and localfunDecls maps of σ^b
   - for all procedure definitions (see the funDefs field in Listing 23 on page 81), we used the inductive rule `execFunction`, the Coq version of `run`.
   - for all package name p, if p is associated with the map m_p^a in σ^a (see externals in Listing 23 on page 81), then p is associated with a map m_p^b in the σ^b and the maps m_p^a and m_p^b are equivalents.

Remark: We proved that this equivalence is transitive.

The Coq definition of this environment equivalence is written in Listing 31 on page 91 and the Coq formalization of the other notions can be found in the Coq source code of this thesis.

6-VI Conclusion

This chapter described the formal semantics of VCP Ada and particularly the definition of semantic preservation, that is the definition that ensures that the VCP Ada optimizations (the middle end), implemented by AST to AST function, are correct. The next chapter presents the VCP Ada optimizations we implemented and uses the semantic preservation definition to prove their correctness.

```
Inductive evalExpression : env -> expression -> value -> Prop :=
| evalELitInt: forall v env, evalExpression env (ELit (Int_lit v)) (Int v)

(* equivalent to ──────────── when v ∈ ℤ*)
                 σ ⊢ v ↦ v

| evalELitBool: forall v env, evalExpression env (ELit (Bool_lit v)) (Bool v)

(* equivalent to ──────────── when v is a boolean*)
                 σ ⊢ v ↦ v

| evalEVarConst : forall id env v, getIdsValue id env = Some (Value (ConstantValue v)) ->
evalExpression env (EVar id) v

(* equivalent to  σ ⊢var id ↦ v
                  ──────────── *)
                  σ ⊢ id ↦ v


| evalEVarVar : forall id env v, getIdsValue id env = Some (Value (VariableValue v)) ->
evalExpression env (EVar id) v

(* equivalent to  σ ⊢var id ↦ v
                  ──────────── *)
                  σ ⊢ id ↦ v

| evalEVarNone : forall id env, getIdsValue id env = None ->   evalExpression env (EVar id) Undefined

(* equivalent to  σ ⊢var id ↦ ∅
                  ──────────── *)
                  σ ⊢ id ↦ ↯

| evalEbinary :
forall e0 e1 v0 v1 op v env, evalExpression env e0 v0 ->
evalExpression env e1 v1 -> Math.binary_operation op v0 v1 = Some v -> evalExpression env (Ebinary op e0 e1) v

(* equivalent to  σ ⊢ e0 ↦ v0    σ ⊢ e1 ↦ v1    Math.binary_operation op v0 v1 = v
                  ──────────────────────────────────────────────────────────────── *)
                                    σ ⊢ e0 op e1 ↦ v


| evalEunary : forall e0 v0 op v env, evalExpression env e0 v0 ->
Math.unary_operation op v0 = Some v ->  evalExpression env (Eunary op e0) v.

(* equivalent to  σ ⊢ e0 ↦ v0    Math.unary_operation op v0 = v
                  ──────────────────────────────────────────── *)
                                σ ⊢ op e0 ↦ v
```

Listing 27 *– VCP Ada semantics Coq implementation - expression evaluation inductive relation*

```
Definition env_equiv e e' :=
  StringMapModule.map_eq e.(localVars) e'.(localVars) /\
  StringMapModule.map_eq e.(globalVars) e'.(globalVars) /\
  StringMapModule.map_eq e.(localfunDecls) e'.(localfunDecls) /\
  (
    forall i,
    match StringMapModule.get i e.(funDefs), StringMapModule.get i e'.(funDefs) with
    | Some f, Some f' => forall run_env res params,
        is_runtime_of e run_env ->
        execFunction (run_env, inevnt) f params res ->
        execFunction (run_env, inevnt) f' params res
    | None, None => True
    | _, _ => False
    end
  ) /\
  (
    forall i,
    match StringMapModule.get i e.(externals), StringMapModule.get i e'.(externals) with
    | Some v, Some v' => StringMapModule.map_eq v v'
    | None, None => True
    | _, _ => False
    end
  ).

```

Listing 31 – *VCP Ada semantics equivalence - environment equivalence*

Chapter 7

Verified optimizations for VCP Ada

Chapter 5 provides solutions to build formally verified front-ends that allow a formally verified transformation of the source code into an abstract syntax tree (AST). Chapter 6 provides the formal constructions allowing to evaluate an AST, to simulate the execution of source code and to validate optimizations. This chapter focuses on the study, the research, the implementation and the formal verification of VCP Ada source code optimizations.

7-I Vital Coded Processor related restrictions

Siemens' current implementation of the Vital Coded processor technique poses restrictions on optimization opportunities. Those restrictions, whose verification is under the responsibility of the Signature Predetermination Tool (SPT), can be grouped into three categories: the non-duplicable instructions, the code size limit and the non-compensable variable problem.

The non-duplicable instructions restriction is related to the identifier of *converge* or *begin_loop* instructions. In fact, those identifiers are unique, pre-determinable (can be evaluated at compile time) positive integers that must be different for each *converge* or *begin_loop* instruction. For example, the code in Listing 32 will not be accepted by the SPT because *begin_loop* and *converge* are called with 0 as parameter.

```
1  begin_loop(0);
2  while w(l < u) loop
3          write(tab, l, r);
4          l := l + 1;
5          end_iteration;
6  end loop;
7  if t(v) then
8          f(l);
9          end_branch(0);
10 else
11         f(u);
12         end_branch(1);
13 end if;
14 converge(0);
```

Listing 32 *– Example of code that is not accepted by the SPT*

This restriction is extended to procedure calls. For example, Figure 33 presents the `max` procedure

that stores in `r` the maximum value between `x` and `y`. As the `max` procedure instruction contains an `if/else` instruction, it also has to contain a converge instruction. The identifier for that converge instruction is the *table* parameter (`t`) of the `max` procedure. Consequently, the value of the parameter `t` must also be unique and pre-determinable for each `max` procedure call.

```
procedure max( x : SINT; y : SINT; r : out SINT; t : TABLE) is begin
        if (x <= y) then
                r := assign(y);
                end_branch(0);
        else
                r := assign(x);
                end_branch(1);
        end if;
        converge(t);
end max;
```

Listing 33 *– Definition of the function max*

Two of the optimizations we will implement will perform code duplication. Those duplications can lead to pieces of code that will be rejected by the SPT. For example, Listing 34 describes an example of the application while unrolling optimization that is rejected by the SPT. The solution to allow the duplication of procedure calls such as `max` is to change the `table` parameter (`t`).

Non-optimized code

```
l := assign(0);
u := assign(2);
begin_loop(0);
while w(l < u) loop
        read(rtab, l, rv);
        max(a, rv, wv, l);
        write(wtab, l, rv);
        l := l + 1;
        end_iteration;
end loop;
```

Optimized but rejected code

```
l := assign(0);
u := assign(2);
read(rtab, l, rv);
max(a, rv, wv, 0);
write(wtab, l, rv);
l := l + 1;
read(rtab, l, rv);
max(a, rv, wv, 0);
write(wtab, l, rv);
l := l + 1;
```

Optimized and accepted code

```
l := assign(0);
u := assign(2);
read(rtab, l, rv);
max(a, rv, wv, 0);
write(wtab, l, rv);
l := l + 1;
read(rtab, l, rv);
max(a, rv, wv, 1);
write(wtab, l, rv);
l := l + 1;
```

Listing 34 *– While unrolling optimization in VCP Ada*

However, the possible inability to access or process the complete list of the source code files of the program to optimize (see missing declaration problems described in 6-I) makes it difficult to modify correctly `table` parameters. To solve that problem, we introduced the concept of *non-duplicable* procedures.

Generally, a procedure call is said to be non-duplicable if at least one of the following conditions is true:

1. The list of instructions of the called procedure contains a call to a `converge` or `end_branch` instruction
2. The list of instructions of the called procedure contains a call to a procedure that is non-duplicable
3. If the list of instructions of the called procedure is not accessible (due to missing declaration problems), we suppose that a procedure call is non-duplicable if the signature of the called procedure contains at least one parameter whose type is `table`

```
1 if (x <= y) then
2         r := assign(y);
3         end_branch(0);
4 else
5         r := assign(x);
6         end_branch(1);
7 end if;
8 converge(t);
9 t := assign(r);
```

Listing 35 – *Compensable example.*

4. If neither the list of instructions nor the signature of the called procedures are accessible (due to missing declaration problems) we suppose that a procedure call is non-duplicable if the call parameters contain at least one variable whose name starts by `'t_'` or `'lt_'` followed by a number. In fact, after analysing a set of VCP Ada source code files, we noticed that the name `table` variables start by `'t_'` or `'lt_'` followed by a number.

We extended that definition to instructions: a sequence of instructions is non-duplicable if it contains calls to non-duplicable procedures. As a consequence, we forbid optimizations that imply the duplication of non-duplicable sequences of instructions. For example, we refuse to execute the while unrolling optimization if the sequence of instructions that will be duplicated is non-duplicable.

Due to hardware limits, the Signature Predetermination Tool (SPT) performs two other verifications:

1. Given an `if-else` or `case` instruction, the number of different branches of that instruction must be lower than 100. According to the rules related to the `end_branch` procedure, this condition can be stated as follows: the maximal parameter of an end_branch instruction must be lower than 99.

2. The maximal size of a source code file must be less than 2Mb.

If at least one of those conditions is not verified, the SPT stops the compilation process.

Another restriction is due to optimization reasons. During years, signature compensations have been time consuming and thus increasing the execution time of programs. As a consequence, Siemens engineers introduced a new concept: the non-compensable variables. Non-compensable variables are vital variables whose signatures cannot be compensated and so those signatures become invalid in case of compensation (see section2-III.C.1). Consequently, non-compensable variables that are modified in `if-else`, `case` or `while` instruction cannot be read outside the structure or must be reassigned – to reinitialize their signatures – before read. For example, if the variable `r` is non compensable, then the processing of the code in figure 35 by the SPT will fail.
By default, all the local variables are non-compensable variables and that allows to reduce considerably the compensation operation and thus programs' execution time. This optimization poses limitations to source code optimizations. In fact, we must make sure that source transformation will be accepted by the SPT.

To take those restrictions into account, we implemented two functions:

- The first one, called `duplicate`, takes a sequence of instructions and returns the boolean `true` if that sequence is *non-duplicable* and `false` otherwise.

- The second one, called `decider`, takes two sequences of instruction s_0, s_1 and returns `true` if the optimization that transforms s_1 into s_0 will be accepted by the SPT and `false` otherwise.

As an error in `duplicate` and `decider` functions will be detected by SPT, we decided to implement those functions in OCaml – without correctness proofs. During the Coq implementation of our source code optimizations, we declared those functions as abstract Coq functions.

7-II Optimization research protocol

The impact of an optimization is evaluated by its efficiency – difference of execution time before and after the application of the optimization – and its occurrence – the appearance rate of situations where this optimization can be applied. In fact, optimizations are to perform modifications from an initial source code. Those modifications start with an analysis of the input source code in order to detect pieces of source code that can be improved. If those detections are rare or if the modifications do not have an impact on the resources or the execution time of a program, those detections and modifications are worthless. One advantage of performing optimizations on source code is that less energy is needed to evaluate the efficiency of an optimization. In fact, one needs only apply source code changes by hand on the test source code, then compile the optimized code and compare the impact on the execution time to evaluate its efficiency. During the research protocol, we implemented unverified OCaml prototypes of all the optimizations we wanted to test on a set of VCP Ada source code used to evaluate the execution time of functions defined in VCP LIB. Those prototypes gave us an estimation of the efficiency and the occurrence of chosen optimizations.

Those chosen optimizations come from 2 sources:

1. **Taking advantage on VCP LIB function and SPT:** There exist situations where VCP LIB functions execute slowly and can be replaced by other ones, leading to opportunities;
2. **Common source code optimizations:** We adapted common Ada source code optimization techniques such as expression simplification, constant propagation and dead-code elimination to VCP Ada.

The optimizations we decided to implement can be separated into 2 categories – syntax or context-free optimizations, and partial execution or context-sensitive optimizations – that we describe in the next sections.

7-III Syntax optimizations

Syntax or context-free optimizations are source code transformations that consist in the recognition of a pattern in the source code and the replacement of the recognized pieces of code by other pieces of code that execute faster. We found and implemented one optimization that consists in procedure inlining using compilation pragmas and two simple but effective optimizations based on converge instructions. As a reminder, the `converge` procedure is used by the **signature predetermination tool** (SPT) to generate compensation tables and also perform the compensations (see Section 2-III.C on page 21 for more details). In VCP Ada, a call to the `converge` function must be the first instruction executed after a conditional branch – if/else or case – instruction and each branch of this structure must finish by a `end_branch` call. According to VCP Ada expert measurements, converge instructions are time consuming. In fact, this instruction includes data exchange with the co-processor (see Section 3-I.C.1 on page 37), signature computation for all the possibly modified variables and an update of the tracer. However, its syntax rules allow optimization possibilities.

Most syntax optimizations we implemented modify sequences of instructions. The implementation of this kind of optimizations starts by implementing an instruction syntax optimizer that is a function that takes a sequence of instructions (see `<insts>` in Section 3-I.B.3.2 on page 35) and returns a sequence

of instructions. Using the notations described in Section 6-III.B, we define the correctness of instruction syntax optimizer as follows:

Definition 2 (the correctness of instruction syntax optimizers at instruction level)**.** *Let f be an instruction syntax optimizer. The sentence 'f is **correct**' means that for every abstract machine (σ,ϖ), instruction evaluation result r and sequence of instructions i, if $(\sigma,\varpi) \models i \Rightarrow r$ then $(\sigma,\varpi) \models f(i) \Rightarrow r$.*

From that definition, we proved the following lemma:

Lemma 3 (transitivity of the correctness of instruction syntax optimizers)**.** *Let f_1 and f_2 be two instruction syntax optimizers. If f_1 and f_2 are correct, then the composition function of f_1 and f_2, that is $x \mapsto f_2(f_1(x))$, is correct.*

This lemma is proved as follows: let f_1 and f_2 be two correct instruction syntax optimizers. Let m, r and i be an abstract machine, an evaluation result and a sequence of instructions. As f_1 is correct, we know that if $m \models i \Rightarrow r$ then $m \models f_1(i) \Rightarrow r$. As f_2 is correct, we know that if $m \models f_1(i) \Rightarrow r$ then $m \models f_2(f_1(i)) \Rightarrow r$. So, if $m \models i \Rightarrow r$ then $m \models f_2(f_1(i)) \Rightarrow r$.

To facilitate the obtaining of syntax optimizers from instruction syntax optimizers, we implemented a function P that takes an instruction syntax optimizers and an AST (abstract syntax tree, used to represent source codes, see `<comp>` in section 3-I.B.4) and returns an AST. The formal description of P is presented in figure 7.1. Using that formal description, we proved that for every instruction syntax optimizer f, if f is correct, then $o \mapsto P(f,o)$ is a correct source code optimizer (see definition 1).

7-III.A Converge removal 1

This optimization aims to reduce the number of converge instructions by taking advantage of VCP Ada syntax rules. The listing 36 presents two equivalent pieces of source code that have two different variable signature managements. In fact, only calls to the `converge` and the `end_branch` instructions have been modified or removed. The **optimized code** runs quicker because the **un-optimized code** makes two compensation processes (line 10 and 17) instead of one (line 17). In the worst cases, the optimized code avoids one procedure call while in the best cases, the optimized code avoids compensations.

According to a performance benchmark, a call to a converge instruction takes 40 microseconds (time needed to locate the compensation table) plus a compensation time that is dependent on the number of variables involved in the compensation. So, each removal of a converge call in a procedure should reduce the execution time of that procedure by 40 microseconds.

7-III.A.1 Implementation and proof

Considering a sequence of instructions s such that $s = i_0 \;\; i_1 \;\; \ldots \;\; i_m$, the converge removal - 1 optimization performs transformations T as follows:

1. **pattern detection:** This step of the optimization consists in determining if i_m is an `end_branch` instruction and that there exists n such that $0 < n < m$ and i_n is a `converge` instruction. If it is not the case, then the sequence of instruction returned by T is s – no optimization is performed.
2. **duplication and insertion:** Regarding the syntax rules of VCP Ada, we can deduce that i_{n-1} is a `if/else` or `case` instruction. Those instructions are branch instruction and the last instruction of each branch is an `end_branch` instruction. We use the function I that allows us to obtain i'_{n-1} from i_{n-1} by replacing the last `end_branch` instruction of each of the branches of i_{n-1} by the sequence $s' = \;\; i_{n+1} \;\; i_{n+2} \ldots \;\; i_m$. Before those duplications and insertions, we ensure that s' is duplicable

Considering an instruction syntax optimizer f, P can be defined as follows:

$$P\left(f,\ \begin{array}{l}\texttt{with } p_0\texttt{;}\\ \texttt{with } p_1\texttt{;}\\ \texttt{...;}\\ \texttt{with } p_n\texttt{;}\\ u\end{array}\right) = \begin{array}{l}\texttt{with } p_0\texttt{;}\\ \texttt{with } p_1\texttt{;}\\ \texttt{...;}\\ \texttt{with } p_n\texttt{;}\\ P_1(f,u)\end{array}$$

$$P_1\left(f,\ \begin{array}{l}\texttt{package } x \texttt{ is}\\ c\\ \texttt{end } x\texttt{;}\end{array}\right) = \begin{array}{l}\texttt{package } x \texttt{ is}\\ c\\ \texttt{end } x\texttt{;}\end{array}$$

$$P_2\left(f,\ \texttt{procedure } x \texttt{ (}a_0\texttt{, }a_1\texttt{, ..., }a_n\texttt{);}\right) = \texttt{procedure } x \texttt{ (}a_0\texttt{, }a_1\texttt{, ..., }a_n\texttt{);}$$

$$P_2\left(f,\ \begin{array}{l}\texttt{procedure } x \texttt{ (}a_0\texttt{, }a_1\texttt{, ..., }a_n\texttt{)}\\ \texttt{renames } x'\texttt{;}\end{array}\right) = \begin{array}{l}\texttt{procedure } x \texttt{ (}a_0\texttt{, }a_1\texttt{, ..., }a_n\texttt{)}\\ \texttt{renames } x'\texttt{;}\end{array}$$

$$P_1\left(f,\ \begin{array}{l}\texttt{package body } x \texttt{ is}\\ v_1\\ v_2\\ \texttt{...}\\ v_n\\ g_1\\ g_2\\ \texttt{...}\\ g_m\\ \texttt{end } x\texttt{;}\end{array}\right) = \begin{array}{l}\texttt{package body } x \texttt{ is}\\ v_1\\ v_2\\ \texttt{...}\\ v_n\\ P_2(f,g_1)\\ P_2(f,g_2)\\ \texttt{...}\\ P_2(f,g_m)\\ \texttt{end } x\texttt{;}\end{array}$$

$$P_2\left(f,\ \begin{array}{l}\texttt{procedure } x \texttt{ (}a_0\texttt{, }a_1\texttt{, ..., }a_n\texttt{) is}\\ v_0\texttt{: } t_0\texttt{;}\\ v_1\texttt{: } t_1\texttt{;}\\ \texttt{...}\\ v_n\texttt{: } t_n\texttt{;}\\ \texttt{begin}\\ s\\ \texttt{end } x\texttt{;}\end{array}\right) = \begin{array}{l}\texttt{procedure } x \texttt{ (}a_0\texttt{, }a_1\texttt{, ..., }a_n\texttt{) is}\\ v_0\texttt{: } t_0\texttt{;}\\ v_1\texttt{: } t_1\texttt{;}\\ \texttt{...}\\ v_n\texttt{: } t_n\texttt{;}\\ \texttt{begin}\\ f(s)\\ \texttt{end } x\texttt{;}\end{array}$$

$$P_1\left(f,\ \begin{array}{l}\texttt{procedure } x \texttt{ (}a_0\texttt{, }a_1\texttt{, ..., }a_n\texttt{) is}\\ v_0\texttt{: } t_0\texttt{;}\\ v_1\texttt{: } t_1\texttt{;}\\ \texttt{...}\\ v_n\texttt{: } t_n\texttt{;}\\ \texttt{begin}\\ s\\ \texttt{end } x\texttt{;}\end{array}\right) = \begin{array}{l}\texttt{procedure } x \texttt{ (}a_0\texttt{, }a_1\texttt{, ..., }a_n\texttt{) is}\\ v_0\texttt{: } t_0\texttt{;}\\ v_1\texttt{: } t_1\texttt{;}\\ \texttt{...}\\ v_n\texttt{: } t_n\texttt{;}\\ \texttt{begin}\\ f(s)\\ \texttt{end } x\texttt{;}\end{array}$$

Figure 7.1 *– Transforming instruction syntax optimizers into syntax optimizers.*

Un-optimized code

```
if cond_1 then
        ins_s;
        if cond_2 then
                c_ins_s;
                end_branch (0);
        else
                e_ins_s;
                end_branch (1);
        end if;
        converge (t_0);
        ins_s2;
        end_branch(0);
else
        ee_ins_s;
        end_branch(1);
end if;
converge (t_1);
```

Optimized code

```
if cond_1 then
        ins_s;
        if cond_2 then
                c_ins_s;
                ins_s2;
                end_branch (0);
        else
                e_ins_s;
                ins_s2;
                end_branch (1);
        end if;
        -- Removal of converge instruction
else
        ee_ins_s;
        end_branch(2); -- end_branch recomputing
end if;
converge (t_1);
```

Listing 36 *– Converge removal 1 example. In this notation `cond_1` and `cond_2` are conditions (boolean expressions), `ins_s`, `c_ins_s`, `e_ins_s`, `ee_ins_s` are arbitrary sequences of instructions and `ins_s2` is a duplicable sequence of instructions.*

(using the `duplicate` function) and that this transformation will be accepted by the SPT (using the `decider` function). If s' is not duplicable or if I fails (typically when i_n is not a a `if/else` or `case` instruction) or if the replacement of i_{n-1} by i'_{n-1} is not accepted by the SPT, then T returns s – no optimization is performed. Else, T returns $i_0 \;\; i_1 \;\; \ldots \;\; i_{n-2} \;\; i'_{n-1}$.

To propagate this transformation to imbricated `if/else` or `case` instructions we defined a function called T' whose formal description is presented in figure 7.2.

Using T' we obtain a transformation in which the parameters of `end_branch` instructions are not correct. To solve that problem, we implemented a function F_e that takes a sequence of instructions and returns a sequence of instructions in which the parameters of `end_branch` instructions are correctly modified (see Section 3-I.A).

The function I is implemented by the Coq function `insert_at_the_end_instruction` located in `Converge2.v`, T' is implemented by `optimize_instruction_s` located in `Converge3.v` and F_e is implemented by the `recompute_end_branch_param` function located in `OptimUtils.v`.

The *converge removal 1* instruction syntax optimizer consists into applying T' and then F_e. We proved the correctness of that instruction syntax optimizer in Coq as follows:

- **the correctness of I:** Let i be an `if/else` or `case` and s a sequence of instructions. If we can obtain i' by duplicating and inserting s into i using I, we proved that for every abstract machine (σ, ϖ), environment σ' and trace $\bar{\omega}'$,
  - if $(\sigma,\varpi) \models i \Rightarrow (\bullet, \sigma', \varpi')$ then $(\sigma,\varpi) \models i' \Rightarrow (\bullet, \sigma', \varpi')$
  - for every evaluation result r if $(\sigma,\varpi) \models i \Rightarrow (\circ, \sigma', \varpi')$ and $(\sigma,\varpi) \models i\,;\;\; s \Rightarrow r$ then $(\sigma,\varpi) \models i' \Rightarrow r$
- **The correctness of T' and F_e:** This proof takes advantage on the fact that `converge` and `end_branch` instructions are not logged in the trace (see listing 12). Using case analysis and the correctness of I, we proved that T' and F_e are correct optimizations.

$$T'(i_0;\ i_1;\ \ldots;\ i_n) \quad = \quad T(T'(i_0)\ \ T'(i_1)\ \ \ldots\ \ \ T'(i_n))$$

$$T'\left(\begin{array}{l}\texttt{if } c_0 \texttt{ then}\\ s_0\\ \texttt{elsif } c_1 \texttt{ then}\\ s_1\\ \ldots\\ \texttt{else}\\ s_m\end{array}\right) = \begin{array}{l}\texttt{if } c_0 \texttt{ then}\\ T'(s_0)\\ \texttt{elsif } c_1 \texttt{ then}\\ T'(s_1)\\ \ldots\\ \texttt{else}\\ T'(s_m)\end{array}$$

$$T'\left(\begin{array}{l}\texttt{case } v\\ \texttt{when } e_0 \texttt{ =>}\\ s_0\\ \texttt{when } e_1 \texttt{ =>}\\ s_1\\ \ldots\\ \texttt{when others}\\ s_m\end{array}\right) = \begin{array}{l}\texttt{case } v\\ \texttt{when } e_0 \texttt{ =>}\\ T'(s_0)\\ \texttt{when } e_1 \texttt{ =>}\\ T'(s_1)\\ \ldots\\ \texttt{when others}\\ s_m\end{array}$$

$$T'\left(\begin{array}{l}\texttt{while } c \texttt{ loop}\\ s\\ \texttt{end loop}\end{array}\right) = \begin{array}{l}\texttt{while } c \texttt{ loop}\\ T'(s)\\ \texttt{end loop}\end{array}$$

For every instruction i that are neither an `if/else` nor `case` nor `while` instruction, $T'(i) = i$.

Figure 7.2 *– Propagating the* converge removal 1 *transformation to imbricated* `if/else` *or* `case`

- **The correctness of the *converge removal 1* instruction syntax optimizer:** We used the transitivity of the correctness of instruction syntax optimizers (see lemma 3) to prove that the *converge removal 1* instruction syntax optimizer is correct.

Using the correctness of the instruction syntax optimizer of *converge removal 1* and the function P defined in figure 7.1 to obtain the *converge removal 1* syntax optimizer and prove its correctness.

7-III.B Converge removal 2

The result of the first technique that removes converge instructions leads us to the conclusion that the number of converge instructions in the source code has an impact on the execution time. The optimization described in this section aims to perform more removals of converge instructions as in the example described in Listing 37. In this example, the optimized code has to perform additional compensations for variables that are modified in `ins` but reduces the number `converge` instructions. The prototype version of this optimization showed its high occurrence and its efficiency on a test program. This prototype also showed the importance the implementation of the `duplicate` and `decider` functions.

7-III.B.1 Implementation and proof

Considering a sequence of instructions s such that $s = i_0\ \ i_1\ \ \ldots\ \ \ i_m$, the converge removal - 2 optimization performs transformations T as follows:

1. **pattern detection:** This step consists into verifying if i_1 is a `converge` instruction and if there exists a number n such that $0 < n \leq m$ and i_n is a `converge`. If it is not the case then T returns s – no optimization is performed.
2. **sequence splitting:** After the pattern detection step, we defined the function S that takes s and returns two sequences: $i_2\ \ i_3\ \ \ldots\ \ \ i_{n-1}$ and $i'_n\ \ i_{n+1}\ \ \ldots\ \ \ i_m$ where n is the minimal natural

Un-optimized code

```
if cond then
  t1_ins;
  end_branch (0);
else
  f1_ins;
  end_branch (1);
end if;
converge(t0);
ins;
if cond2 then
  t2_ins;
  end_branch (0);
else
  f2_ins;
  end_branch (1);
end if;
converge(t1);
```

Optimized code

```
if cond then
  t1_ins;
  ins;
  if cond2 then
    t2_ins;
    end_branch (0);
  else
    f2_ins;
    end_branch (1);
  end if;
else
  f1_ins;
  ins;
  if cond2 then
    t2_ins;
    end_branch (2);
  else
    f2_ins;
    end_branch (3);
  end if;
end if;
-- Code duplication and end_branch
converge(t0);
```

Listing 37 *– Converge removal 2 example. In this notation* `cond` *and* `cond2` *are conditions (boolean expressions),* `t1_ins`*,* `f1_ins` *are arbitrary sequences of instructions and* `t2_ins`*,* `f2_ins`*,* `ins` *are duplicable sequences of instructions.*

number such that $0 < n \leq m$ and i_n is a `converge` and where i'_n is a `converge` instruction except that its parameter is replaced by the parameter of i_1. If i_1 is not a converge instruction or if S fails (typically when there is no number n that meets the required conditions), then the sequence of instruction returned by T is s – no optimization is performed.

3. **duplication and insertion:** Similarly to the converge removal - level 1, we use the function I to obtain i'_0 from i_0 by replacing the last `end_branch` instruction of each of the branches of i_0 by the sequence $s' = i_2 \ i_3 \ \ldots \ i_{n-1}$. Before those duplications and insertions, we ensure that s' is duplicable (using the `duplicate` function) and that this transformation will be accepted by the SPT (using the `decider` function). If s' is not duplicable or if I fails (typically when i_n is not a a `if/else` or `case` instruction) or if the replacement of i_0 by i'_0 is not accepted by the SPT, then this transformation returns s – no optimization is performed. In case of success, T returns $i'_0 \ i'_n \ i_{n+1} \ \ldots \ i_m$

Similarly to the converge removal-1 optimization, we propagated T to imbricated `if/else` or `case` instructions using the transformation T' described in figure 7.3. We also used the F_e function to modify correctly the parameters of `end_branch` instructions.

The functions S and T' are implemented by the Coq functions named `split_instrs` and `optimize_instruction_s` located in `Converge2.v`.

The instruction syntax optimizer of *converge removal 2* consists into applying T' and then F_e. We proved the correctness of that instruction syntax optimizer in Coq as follows:

- **the correctness of S:** Let s be a sequence of instructions. We proved that for every abstract machine

$$T'(s) \quad = \quad T(T'(i_0)\ \ T'(i_1\ \ \ldots\ \ \ i_m)$$

$$T'\left(\begin{array}{l}\texttt{if } c_0 \texttt{ then}\\ s_0\\ \texttt{elsif } c_1 \texttt{ then}\\ s_1\\ \ldots\\ \texttt{else}\\ s_m\end{array}\right) = \begin{array}{l}\texttt{if } c_0 \texttt{ then}\\ T'(s_0)\\ \texttt{elsif } c_1 \texttt{ then}\\ T'(s_1)\\ \ldots\\ \texttt{else}\\ T'(s_m)\end{array}$$

$$T'\left(\begin{array}{l}\texttt{case } v\\ \texttt{when } e_0 \texttt{ =>}\\ s_0\\ \texttt{when } e_1 \texttt{ =>}\\ s_1\\ \ldots\\ \texttt{when others}\\ s_m\end{array}\right) = \begin{array}{l}\texttt{case } v\\ \texttt{when } e_0 \texttt{ =>}\\ T'(s_0)\\ \texttt{when } e_1 \texttt{ =>}\\ T'(s_1)\\ \ldots\\ \texttt{when others}\\ s_m\end{array}$$

$$T'\left(\begin{array}{l}\texttt{while } c \texttt{ loop}\\ s\\ \texttt{end loop}\end{array}\right) = \begin{array}{l}\texttt{while } c \texttt{ loop}\\ T'(s)\\ \texttt{end loop}\end{array}$$

For every instruction i that are neither an `if/else` nor `case` nor `while` instruction, $T'(i) = i$.

Figure 7.3 *– Propagating the* converge removal 2 *transformation to imbricated* `if/else` *and* `case` *instructions*

(σ, ϖ) and interpretation result r, if $(\sigma, \varpi) \models s \Rightarrow r$ and we can obtain two sequences of s_1 and s_2 from s using the function S, then,

  - either there exists an environment σ' and a trace ϖ' such that $(\sigma, \varpi) \models s_1 \Rightarrow (\circ, \sigma', \varpi')$ and $(\sigma', \varpi') \models s_2 \Rightarrow r$
  - or $(\sigma, \varpi) \models s_1 \Rightarrow r$

- **The correctness of T':** Using case analysis and the correctness proofs of I and S, we proved that T' is a correct optimization.
- **The correctness of the instruction syntax optimizer of *converge removal 2*:** We used the transitivity of the correctness of syntax optimizers (see lemma 3) to prove that the instruction syntax optimizer of *converge removal 2* is correct.

Using the correctness of the instruction syntax optimizer of *converge removal 2* and the function P defined in figure 7.1 to obtain the *converge removal 2* syntax optimizer and prove its correctness.

7-III.C Inlining functions using pragmas

Inlining [Cooper et al., 1991] is the process of replacing a procedure call with the body of the called procedure. This eliminates call-linkage overhead and can expose significant optimization opportunities. In Ada and in VCP Ada, developers can provide directives to compilers using *pragmas*. One of those pragmas allows inlining. Consequently, we used that pragma to give guidance to the compiler on whether or not a procedure should be inlined. For that optimization, we implemented a Coq function that takes a `string->bool` function, an AST and returns an AST. The `string->bool` function takes the name of a procedure and returns true if and only if an inline pragma should be added for this procedure. The correctness of this optimization is based on the fact that pragmas have no effect on abstract machines and that the definition of semantic preservation allows adding pragmas.

The implementation and correctness proof of this optimization can be found in `InlinePragmaOptimizer.v`.

7-IV Context sensitive optimizations

The previous section offers solutions to write and prove the correctness of optimizations. Those optimizations have the particularity that they are based on the detection of patterns of pieces of source code and the replacement of those pieces of code by other pieces of code that run faster than the initial ones. However, this way of performing optimizations is limited. In fact, let us consider the following optimization, well-known as *(partial) dead code elimination*:

Un-optimized code

```
1  v := assign(True);
2  if t(v) then
3          pTrue(v);
4          end_branch(0);
5  else
6          pFalse(v);
7          end_branch(1);
8  end if;
9  converge(0);
```

Optimized code

```
1  v := assign(True);
2  pTrue(v);
```

Listing 38 *– Partially dead code elimination (example)*

In this example, the idea of the non-optimized code is to call the procedure `pTrue` if `v` contains the value `true` and `pFalse` otherwise. But, as the variable `v` has been assigned to `true` (**line 1**), the procedure `pTrue` is the one that is called.

In the simplest cases, the dead code elimination optimization can be implemented by syntax optimizations: the pattern to detect would be an assignment to a variable, say *a*, followed by an if/else instruction which condition is related to *a*. But, this detection becomes difficult when there are instructions that do not modify the value of *a* between the assignment and the if/else instruction. What happens if *a* is a constant variable defined in another file? All those questions present the limits of the syntax or context-free optimization presented in Section 7-III and motivates the implementation of the context sensitive optimizations.

Generally, the exact values of the variables of a program can only be known during the program execution. However, developers can state properties about those values – also known as runtime environment – when inspecting the source code. To store those properties, we define a data structure, named *context*. *Context-sensitive* optimizers exploit and maintain that data structure in order to perform correct optimizations by taking advantage of the information stored in it.

In this chapter, we present how we design and implement the concept of **context**, then the optimization related to it.

7-IV.A Partial evaluation

The implementation of *context sensitive optimizations* uses the concepts and materials around partial evaluation [Consel and Danvy, 1993, Meyer, 1991]. The following text provides materials to understand this background.

Let us consider **Program 1** and **Program 2** whose source code is written in Listing 39. **Program 1** describes a function that takes 4 parameters `x`, `y`, `r`, `t` – a table number used to identify the compensation table for the while loop (see section 3-I.A for more details) – then computes and stores x^y when `y` is positive and 1 otherwise into `r`. **Program 2** takes 2 parameters, `x` and `r`, and stores x^3 into `r`.

Program 1

```
procedure pow( x : SINT, y : SINT,
        r : out SINT, t : TABLE) is
        i : SINT;
        begin
        i := init(0);
        r := assign(1);
        begin_loop(t);
        while w(i < y) loop
                r := r * x;
                i := i + 1;
                end_branch;
        end loop;
end pow;
```

Program 2

```
procedure pow_3( x : SINT, r : out SINT) is
        begin
        r := assign(x);
        r := r * x;
        r := r * x;
end pow_3;
```

Listing 39 *– Program specialization example*

Regarding the semantics of those programs, one can state that **Program 2** is **Program 1** where y is equal to 3. In that case, **Program 2** is said to be a *specialization* or a *residual program* of **Program 1**.

Partial evaluation is a source-to-source program transformation technique for optimizing programs by specializing them depending on annotations on some of their inputs. Those optimizations are performed by a partial evaluator that takes advantage of those annotations that can be either static or dynamic.

Static annotations refer to assumptions that remain correct at any time of the execution of the program, that is for example the set of constant variables whose values can be computed at compilation time. Those annotations can be given manually by developers or automatically using a tool. According to Meyer [Meyer, 1991] the static annotation is appropriate for pure functional programming languages where variable values do not change.

Dynamic annotations refer to assumptions that the partial evaluator can call into question. Before executing an instruction in the program, those annotations contain assumptions about the runtime values of some of the program variables. Dynamic annotations describe a relation between each variable, its *state* (known or unknown) and its possible value. The state of a variable can change from known to unknown and vice versa, – typically after assign instructions. Consequently, dynamic annotations have to be updated instruction after instruction. To do that Meyer, formally described a set of rules that can be summarized as follows:

- **State of expressions:** The state of literals such as constant integers is *known*. The state of expressions that consist only of literals is *known*. If an expression contains at least one variable whose state is *unknown*, then the state of that expression is *unknown*. Otherwise, the state of that expression is *known*.

- **Taking account an instruction:** Given an input set of dynamic annotations α that are true before an instruction i, the set of assumptions that are true after the execution of i is noted $\Delta(\alpha, i)$ is computed as follows:

  – if i is a `null` instruction, the dynamic annotations do not change. This means that $\Delta(\alpha,$ `null`$) = \alpha$.

  – if i is an `assignment` instruction, say x `:=` v, only the annotation related to x change: the state of x – the assigned variable – is set to *known* if and only if the state of v – the assigned expression – is *known* and its possible value is the value of the evaluation of v. If the state of v is *unknown*, then the state of x set to *unknown*.

- if i is a `sequence` of instruction, say i_0; i_1; ...; i_n, the modification of a consists of the successive modifications from i_0 to i_n. This means that $\Delta(\alpha, \text{`}i_0; i_1; \ldots; i_n\text{`}) = \Delta(\Delta(\alpha, \text{`}i_1; \ldots; i_n\text{`}), \text{`}i_0\text{`})$
- if i is a `if/else` instructions, say `if` c `then` s_t `else` s_f `end if`, then, $\Delta(\alpha, \text{`}$`if` $c$ `then` $s_t$ `else` $s_f$ `end if`$\text{`})$ depends on the state of c:
  - if the state of c is *known* and evaluated to `true`, then $\Delta(\alpha,$ `if` c `then` s_t `else` s_f `end if`$) = \Delta(\alpha, s_t)$.
  - if the state of c is *known* and evaluated to `false`, then $\Delta(\alpha,$ `if` c `then` s_t `else` s_f `end if`$) = \Delta(\alpha, s_f)$.
  - Let α_t and α_f two sets of annotations such that $\alpha_t = \Delta(\alpha, s_t)$ and $\alpha_f = \Delta(\alpha, s_f)$. If the state of c is *unknown*, $\Delta(\alpha,$ `if` c `then` s_t `else` s_f `end if`$)$ is built as follows: for every variable x in α, if the state of x is *known* in α_t and α_f and associated with the same possible value v in both of them, then the state of x is set to *known* and associated with the value v. Otherwise, the state of x is set to *unknown*.
- if i is a `while` instruction, say `while` c `loop` s `end loop`, $\Delta(\alpha,$ `while` c `loop` s `end loop`$)$ depends on the state of c:
  - if the state of c is *known* and evaluated to `false`, then, $\Delta(\alpha,$ `while` c `loop` s `end loop`$) = \alpha$
  - if the state of c is *known* and evaluated to `true`, then, $\Delta(\alpha, \text{`}$`while` $c$ `loop` $s$ `end loop`$\text{`}) = \Delta(\Delta(\alpha, s),$ `while` c `loop` s `end loop`$)$
  - if the state of c is *unknown*, then, $\Delta(\alpha,$ `while` c `loop` s `end loop`$)$ is α where the state of all the possibly modified variables are set to *unknown*.

With that annotation management, Meyer also suggests instruction changes (optimizations) to speed up the execution time. For example, when the condition of an `if/else` instruction can be evaluated using the annotations, the instruction is replaced by the taken branch (like in partially dead code elimination example, see Listing 38 on page 103).

Specialization is the main task of partial evaluation. To perform it, Meyer proposes to analyse the program calls. If all the parameters of a program call can be evaluated totally, the system does not generate a residual function for this call. In that case, the specialization is equivalent to inlining [Cooper et al., 1991] the input function. If some parameters cannot be evaluated (unknown), the system generates a specialization for this call. This specialization is generated using the optimizations that can be done using the annotations. In the same paper, he also warned about eventual problems with global variable management and proposed solutions for that. In another paper [Meyer, 1999], Meyer provided directions to prove the correctness of those optimizations.

7-IV.B The context implementation and management

Our implementation uses constructions that are similar to what Meyer describes in his papers [Meyer, 1991, Meyer, 1999].

7-IV.B.1 Basic constructions and definitions

In our construction, the set of annotations or *context* is implemented using the environment data-structure described in section 6-II.A and implemented in section 6-IV.A.3. In that construction, the value `undefined`, presented in Listing 21, is used to represent the state *unknown*. When the value v of a variable is not `undefined`, it means that the state of this variable is *known* and associated with that value v. This choice has been motivated by the opportunities it offers in term of scope management – the management of global, local and external variables and functions – the existence of implementations, such as expression evaluation, that is compatible with the theoretical construction of Meyer and the simplification all this offers for correctness proofs.

Definition 4 (partial view)**.** *An environment σ_0 is said to be **a partial view** of another environment σ_1 when the following rules are respected:*

- *at any scope level (see Section 6-II.A), if a variable is defined in σ_1, then the same variable is defined in σ_0 and its value in σ_0 is either equal to the one in σ_1 or equal to* `undefined`.
- *at any scope level, if a variable is not defined in σ_1, then either there is no variable with the same name in σ_0 or there is a variable whose value is* `undefined` *that is defined in σ_0.*
- *σ_1 and σ_0 contain exactly the same function declarations and definitions.*

A context is said to be *valid* if this context is a partial view of the runtime environment. In other terms, a valid context has the same structure as the runtime environment and associates the variables either to their concrete value (value at runtime) or to `undefined`.

Definition 5 (precision)**.** *Let σ_0 and σ_1 be two valid environments. The environment σ_0 is said to be more precise or greater than σ_1 if σ_1 is a partial view of σ_0. In that case, we can also say that σ_1 is lower than σ_0.*

We define the *reasonably lowest environment* of an environment σ as the environment σ in which the state of all the non constant variables is *unknown*. The analysis of compilation units described in section 6-III.D provides us with the structure of the runtime environment. It also provides the context with data such as the value of constant variables and the structure of the runtime environment – the set of elements with associated scope. From the structure of the runtime environment, we can define the relation `is_runtime_of` defined in Section 6-V as follows: let σ and σ_l be two environments such that σ has the same structure as the runtime environment and σ_l is the reasonably lowest environment of σ. The environment σ_r is a possible runtime environment if σ_r has the same structure as σ and all the non constant variable can have arbitrary values: This means that σ_l is a partial view of σ_r.

Definition 6 (being a possible runtime environment)**.** *Considering a source code s and σ such that σ is obtained using the process described in section 6-III.D, an environment σ_r is said to be a possible runtime environment if the reasonably lowest environment of σ is a partial view of σ_r.*

The context-based optimizers refer to functions that will exploit a given valid context to suggest source code modification. Context-based instruction optimizers take a sequence of instructions and a context as inputs and return a sequence of instructions and a context. The efficiency of those instruction optimizers depends on the precision of the input context. For example, to perform the optimization described in Listing 38 above, a context-based optimization requires that the input context associates the variable `v` to value `true`. The context-based optimizers work like interpreters. In fact, when optimizing a list of instructions, they need to keep the context valid instruction after instruction. The evolution of the runtime environment is described by the formal semantics and the evolution of the input context is given by Meyer techniques [Meyer, 1991]. There are two differences between the rules suggested by Meyer and the ones we implemented.

The first difference resides in the management of `if/else` and `while` instructions. In the rules we used, the state of all the variables that are possibly modified in `if/else` or `while` instructions are set to *unknown*. This choice is made to ease the implementation and the Coq correctness proofs. There is a risk that this choice leads to contexts that are less precise than the one proposed by those suggested by the existing rules. For example, let us consider the following code:

```
1  if t(v) then
2          -- arbitrary code ...
3          a := assign(10);
4          end_branch(0);
5  else
6          -- another arbitrary code ...
7          a := assign(10);
8          end_branch(1);
9  end if;
10 converge(0);
```

Listing 40 *– Example for context/annotation management*

Using our rules, at the end of the execution of this code, the state of the variable a is set to *unknown*, whereas using Meyer's rules, the state of a is *known* and associated with the value 10. But we trust that developers would put the assignment instruction outside of the *if/else* instruction.

The second difference resides in the management of procedure calls. Meyer described an algorithm to compute the list of global variables that are modified by each procedure call and use it to set the state of those variables to *unknown*. This technique requires having access to all the source code files of the program and can lead to less precise context.

Considering an environment σ, our rules for procedure calls are works as follows:

- **known procedures:** Known procedures are procedures whose signature or/and definition can be provided by σ. If possible, our rules simulate those procedure calls. Otherwise, our rules set the state of all the global and external variables to unknown. The state of call-by-result and call-by-reference parameters are also set to *unknown*.
- **unknown procedures:** Unknown procedures are procedures which neither signature, nor definition can be provided by σ. This can happen with import problems described in Section 6-I. In those cases, our rules also set the state of all the global and external variables to unknown. Our rules guess the mode of parameters, but our rules know that call-by-result and call-by-reference parameters must be bound to variable names. Consequently, our rules analyse the procedure call instruction and the state of all the parameters that are made up with a variable name only are set to *unknown*.

Using all those definitions above, we define a context-based instruction optimizer as a function that takes a context and a sequence of instructions and then returns a sequence of instructions. Then, we can define the correctness of a context-based instruction optimizer as follows:

Definition 7 (context-based instruction optimizer correctness)**.** *Let f be a context-based instruction optimizer. The sentence 'f is **correct**' means that for every trace ϖ, evaluation result r, sequence of instructions i_0, runtime environment σ_r and context σ_c such that σ_c is a partial view of σ_r, if $(\sigma_r, \varpi) \models i_0 \Rightarrow r$ then $(\sigma_r, \varpi) \models f(\sigma_c, i_0) \Rightarrow r$.*

This definition is very similar to the definition of the correctness of instruction syntax optimizer (see definition 2). We proved a lemma that is similar to lemma 3 for context-based instruction optimizers and implemented an adaptation of the function P (described in figure 7.1) to facilitate the obtention of context-based optimizers (that optimize source codes in the form of AST) from context-based instruction optimizers. We also proved that if a context-based instruction optimizer is correct, then the context-based optimizer implemented using the modified P function is a correct optimization (see the semantic preservation section).

7-IV.B.2 Coq implementation and correctness proof

The context data structure implementation reuses the implementation of environment data structure presented in Listing 23. The Coq implementation of the concept of partial view (presented in definition 4) is implemented using an inductive relation . Using this Coq implementation, we have proven that **being a partial view** is a *reflexive* and *transitive* relation. Setting the state of a variable *unknown* is equivalent to assigning the value `undefined` to this variable. This is implemented by the same function used for assignment (see `setIdsValue` in table 6.1). Using that Coq definition, we have proven the following theorem:

Theorem 8 (partial view and evaluation)**.** *For all environments σ and σ', for every trace ϖ and for every expression e, if σ is a partial view of σ' and $(\sigma', \varpi) \vdash e \mapsto v$ (see Listing 8 on page 70 for more details on the notation) then $(\sigma, \varpi) \vdash e \mapsto v$ or $(\sigma, \varpi) \vdash e \mapsto \varkappa$*

The rules used to update contexts are implemented in the form of a context rule that is a function that takes a context and an instruction and returns a context. A context rule λ is said to be correct if and only if for every instruction i, context σ_c, environments σ_r and σ'_r and traces ϖ_r and ϖ'_r, if $(\sigma_r, \varpi_r) \models i \Rightarrow (\circ, \sigma'_r, \varpi'_r)$ and σ_c is a partial view of σ_r, then $\lambda(\sigma_c, i)$ is a partial view of σ'_r.

Remarks:

- The definition of the expression evaluation relation (see Section 6-III.A on page 69) propagates the notion of *unknown* state. In fact, variables which state is *unknown* are variables that have the value `undefined` and the definition of this expression evaluation relation is written such that any operation that implies the value `undefined` returns `undefined` (if there is no type error detected). So, the use of this relation on a context implements the semantics of *unknown* expression as defined in Meyer's rules.
- This Coq implementation applies the all or nothing law: this means for example that for a given record or array variable, either the context knows the values of all its fields or items, or the state of this record or array is *unknown*.

The first version of context rules we implemented is noted λ_0 and consists into setting the state of all the possible modified variables to *unknown*. Its implementation starts by the implementation of a function that takes a context, an instruction and returns the list of the variables that can be modified by that instruction. The formal definition of that function is presented in Listing 41 and uses the notation `possibly_modified`$(\sigma,\ i) = l$ to indicate that the variables in the list l are susceptible to be modified by the execution of the instruction i in the context σ. The notation used to describe the list of variables is described as follow: [] refers to empty list, $h :: l$ refers to a list where h is the first element of the list and l is the tail of the list and the notation $l_0 + l_1$ refers to the concatenation of the list l_0 and l_1. The notation σ_{glob-v} and σ_{ext-v} refer to the list of global and external variables of the context σ. The function `top` is in charge of getting the variable name of a selected component. For example if a variable a is a record that contains a field named b which also refers to a record that contains a field named c, `top`$(a.b.c) = a$.

We proved two properties on λ_0:

- λ_0 is idempotent: We used case analysis to prove that for every instruction i and environment σ, $\lambda_0(\sigma, i) = \lambda_0(\lambda_0(\sigma, i), i)$.
- the correctness of λ_0: We used the idempotence of λ_0, the transitivity of the partial view relation and case analysis to prove that λ_0 is a correct context rule.

Using λ_0 we implemented the second version of context rule noted λ_1. Given a context σ, λ_1 can be described as follows:

$$\frac{}{\texttt{possibly\_modified}(\sigma,\ \texttt{null}) = []} \qquad \frac{}{\texttt{possibly\_modified}(\sigma,\ x\ :=\ v) = [\texttt{top}(x)]}$$

$$\frac{\texttt{possibly\_modified}(\sigma,\ i_0) = l_0 \qquad \texttt{possibly\_modified}(\sigma,\ i_1) = l_1}{\texttt{possibly\_modified}(\sigma,\ i_0; i_1) = l_0 + l_1}$$

$$\frac{\texttt{possibly\_modified}(\sigma,\ i_0) = l_0 \quad \texttt{possibly\_modified}(\sigma,\ i_1) = l_1 \quad \texttt{possibly\_modified}(\sigma,\ i_n) = l_n}{\texttt{possibly\_modified}(\sigma,\ \texttt{if}\ c\ \texttt{then}\ i_0\ \texttt{elsif then}\ i_1\ \ldots\ \ \texttt{elsif then}\ i_{n-1}\ \texttt{else}\ i_n\ \texttt{end if}) = l_0 + l_1 \ldots + l_n}$$

$$\frac{\texttt{possibly\_modified}(\sigma,\ i_0) = l_0 \quad \texttt{possibly\_modified}(\sigma,\ i_{n-1}) = l_{n-1s} \quad \texttt{possibly\_modified}(\sigma,\ i_n) = l_n}{\texttt{possibly\_modified}(\sigma,\ \texttt{case c}(v)\ \texttt{when}\ e_0\ \texttt{=>}\ i_0\ \ldots\ \ \texttt{when}\ e_{n-1}\ \texttt{=>}\ s_{n-1}\ \texttt{when other =>}\ i_n\ \texttt{end case}) = l_0 + \ldots + l_{n-1} + l_n}$$

$$\frac{\texttt{possibly\_modified}(\sigma,\ s) = l}{\texttt{possibly\_modified}(\sigma,\ \texttt{while}\ c\ \texttt{loop}\ s\ \texttt{end loop}) = l}$$

$$\frac{}{\texttt{possibly\_modified}(\sigma,\ \texttt{begin\_loop}(e)) = []}$$

$$\frac{}{\texttt{possibly\_modified}(\sigma,\ \texttt{converge}(e)) = []}$$

$$\frac{}{\texttt{possibly\_modified}(\sigma,\ \texttt{init}(x)) = []} \qquad \frac{}{\texttt{possibly\_modified}(\sigma,\ \texttt{init}(x,\ e)) = [\texttt{top}(x)]}$$

$$\frac{}{\texttt{possibly\_modified}(\sigma,\ \texttt{failure}) = []} \qquad \frac{}{\texttt{possibly\_modified}(\sigma,\ \texttt{end\_iteration}) = []}$$

$$\frac{}{\texttt{possibly\_modified}(\sigma,\ \texttt{end\_branch}) = []}$$

$$\frac{}{\texttt{possibly\_modified}(\sigma,\ \texttt{write}(var,\ e_0,\ e_1)) = [\texttt{top}(var)]}$$

$$\frac{}{\texttt{possibly\_modified}(\sigma,\ \texttt{write}(var,\ e_0,\ e_1,\ e_3)) = [\texttt{top}(var)]}$$

$$\frac{}{\texttt{possibly\_modified}(\sigma,\ \texttt{write\_hard\_input}(var,\ e_0,\ e_1,\ e_2)) = [\texttt{top}(var)]}$$

$$\frac{}{\texttt{possibly\_modified}(\sigma,\ \texttt{read\_hard\_input}(var,\ e_0,\ e_1,\ e_3)) = []}$$

$$\frac{}{\texttt{list\_from\_sig}\ ()\ ()\ =\ []}$$

$$\frac{\texttt{list\_from\_sig}\ (p_1\texttt{:}\ \ m_1\ t_1,\ \ldots,\ p_n\texttt{:}\ \ m_n\ t_n)\ (e_1,\ \ldots,\ e_n)\ =\ \texttt{l}}{\texttt{list\_from\_sig}\ (p\texttt{:}\ \ \texttt{out}\ t,\ p_1\texttt{:}\ \ m_1\ t_1,\ \ldots,\ p_n\texttt{:}\ \ m_n\ t_n)\ (x\ e_1,\ \ldots,\ e_n)\ =\ (\texttt{top}(x))\texttt{::l}}$$

$$\frac{}{\texttt{list\_from\_sig'}\ ()\ =\ []} \qquad \frac{\texttt{list\_from\_sig'}\ (e_1,\ \ldots.\ \ e_n)\ =\ \texttt{l}}{\texttt{list\_from\_sig'}\ (e_0,\ e_1,\ \ldots.\ \ e_n)\ =\ (\texttt{top}(e_0))\texttt{::l}}\ [e_0 \text{ is a variable name}]$$

$$\frac{\texttt{list\_from\_sig'}\ (e_1,\ \ldots.\ \ e_n)\ =\ \texttt{l}}{\texttt{list\_from\_sig'}\ (e_0,\ e_1,\ \ldots.\ \ e_n)\ =\ \texttt{l}}\ [e_0 \text{ is } \textbf{not} \text{ a variable name}]$$

$$\frac{\sigma \vdash_{pro} proc \mapsto (s, d, i)}{\texttt{possibly\_modified}(\sigma,\ \texttt{proc}(e_0,\ e_1,\ \ldots,\ e_n)) = \texttt{list\_from\_sig}\ s\ (e_0,\ e_1,\ \ldots,\ e_n) + \sigma_{glob-v} + \sigma_{ext-v}}$$

$$\frac{\sigma \vdash_{sig} proc \mapsto s}{\texttt{possibly\_modified}(\sigma,\ \texttt{proc}(e_0,\ e_1,\ \ldots,\ e_n)) = \texttt{list\_from\_sig}\ s\ (e_0,\ e_1,\ \ldots,\ e_n) + \sigma_{glob-v} + \sigma_{ext-v}}$$

$$\frac{\sigma \vdash_{sig} proc \mapsto \varnothing}{\texttt{possibly\_modified}(\sigma,\ \texttt{proc}(e_0,\ e_1,\ \ldots,\ e_n)) = \texttt{list\_from\_sig'}\ (e_0,\ e_1,\ \ldots,\ e_n) + \sigma_{glob-v} + \sigma_{ext-v}}$$

Listing 41 *– Context sensitive optimization - possibly_modified inference rules*

- $\lambda_1(\sigma, \texttt{null}) = \sigma$
- Let x be a variable name, e be an expression and v be a value such that if there exists a value v' so that for every trace ϖ, $(\sigma, \varpi) \vdash e \mapsto v'$ then $v = v'$. Otherwise, $v = \varkappa$.

$$\lambda_1(\sigma,\ x\texttt{:=}e) = \begin{cases} \sigma[x \leftarrow v] & \text{if } \texttt{top}(x)\texttt{=}x \\ \sigma[\texttt{top}(x) \leftarrow v] & \text{Otherwise} \end{cases}$$

- $\lambda_1(\sigma,\ i_0\texttt{;}\ \ i_1\texttt{;}\ \ \ldots\texttt{;}\ \ i_n) = \lambda_1(\lambda_1(\sigma,\ i_0),\ i_1\texttt{;}\ \ \ldots\texttt{;}\ \ i_n)$
- Let c be an expression and v be a value such that if there exists a value v' so that for every trace ϖ, $(\sigma, \varpi) \vdash c \mapsto v'$ then $v = v'$. Otherwise, $v = \varkappa$.

$$\lambda_1(\sigma, \texttt{if}\ \ c\ \ \texttt{then}\ \ s_t\ \ \texttt{else}\ \ s_f\ \ \texttt{end if}) = \begin{cases} \lambda_1(\sigma,\ s_t) & \text{if } v = \texttt{true} \\ \lambda_1(\sigma,\ s_f) & \text{if } v = \texttt{false} \\ \lambda_0(\sigma, \texttt{if}\ \ c\ \ \texttt{then}\ \ s_t\ \ \texttt{else}\ \ s_f\ \ \texttt{end if}) & \text{Otherwise} \end{cases}$$

- Let c be an expression. We define a value v as follows: if there exists a value v' such that for every trace ϖ, $(\sigma, \varpi) \vdash c \mapsto v'$ then $v = v'$. Otherwise, $v = \varkappa$.

$$\lambda_1(\sigma, \texttt{while}\ \ c\ \ \texttt{loop}\ \ s\ \ \texttt{end loop}) = \begin{cases} \sigma & \text{if } v = \texttt{false} \\ \lambda_0(\sigma, \texttt{while}\ \ c\ \ \texttt{loop}\ \ s\ \ \texttt{end loop}) & \text{Otherwise} \end{cases}$$

- $\lambda_1(\sigma,\ p(a_0,\ \ a_1,\ \ \ldots,\ \ a_n)) = \lambda_0(\sigma,\ p(a_0,\ \ a_1,\ \ \ldots,\ \ a_n))$

Using the correctness of λ_0, the definition and the properties of the *partial view* relation, we proved that λ_0 is a correct.

The third context rule we implemented aims to provide more precise contexts, particularly for procedure calls and while loops. Processing those cases requires interpreting those instructions and also a termination proof for that interpretation as it can loop. As a consequence, we implemented a third version of our context rule, λ_2. To avoid providing a termination proof, λ_2 takes an additional natural number as parameter – the fuel technique. The main differences between λ_1 and λ_2 can be summarized as follows:

- for every instruction i and context σ, $\lambda_2(0, \sigma, i) = \lambda_1(\sigma, i)$
- Let c be an expression. We define a value v as follows: if there exists a value v' such that for every trace ϖ, $(\sigma, \varpi) \vdash c \mapsto v'$ then $v = v'$. Otherwise, $v = \varkappa$. For every strictly positive natural number n,

$$\lambda_2(n, \sigma, \texttt{while}\ \ c\ \ \texttt{loop}\ \ s\ \ \texttt{end loop}) = \begin{cases} \sigma & \text{if } v = \texttt{false} \\ \lambda_2(n-1, \lambda_2(n, \sigma, s), \texttt{while}\ \ c\ \ \texttt{loop}\ \ s\ \ \texttt{end loop}) & \text{if } v = \texttt{true} \\ \lambda_1(\sigma, \texttt{while}\ \ c\ \ \texttt{loop}\ \ s\ \ \texttt{end loop}) & \text{Otherwise} \end{cases}$$

- Let σ, p, s, d and i be respectively a context, a procedure name, a procedure signature, a set of procedure local variable declaration and a sequence of procedure instructions such that $\sigma \vdash_{pro} p \mapsto (s, d, i)$. We implemented functional versions of the `copy_in` (`copy_in_f`), the `init_locals` (`init_locals_f`) and the `copy_out` (`copy_out_f`) and proved that they are equivalent to their inductive definition presented in Listings 14, 15 and 16. We adapted the `copy_out_f` (`copy_out_f2`) functions to use contexts (`copy_out_f2`) and then we described λ_2 processing for procedure calls as follows: for every strictly positive natural number n,

$$\frac{\texttt{copy\_in\_f}\ \ (\sigma, s, p) = \sigma_1 \quad \texttt{init\_locals\_f}\ \ (\sigma_1, d) = \sigma_2 \quad \lambda_2(n-1, \sigma, s) = \sigma_3 \quad \texttt{copy\_out\_f2}\ \ (\sigma_3, s, (a_0,\ \ a_1,\ \ \ldots,\ \ a_n), \sigma) = \sigma_4}{\lambda_2(n, \sigma,\ p(a_0,\ \ a_1,\ \ \ldots,\ \ a_n)) = \sigma_4}$$

Using the correctness of λ_1, we proved by recurrence on the fuel parameter that λ_2 is correct.

7-IV.C The implemented context-sensitive optimizations

Using the λ_2 context rule, we implemented three context-based optimizations: dead code elimination, loop unrolling and expression simplification (associated with constant propagation). Those well-known optimizations come with adaptations related to the VCP Ada syntax. The context-based instruction optimizers of those optimizations are implemented in the form of Coq functions that take a sequence of instructions associated with a context and return a context associated with a sequence of instructions.

Given a context-based transformation T (a function that takes a context, an instruction and returns an instruction that is optimized), a context σ, the formal description of the implementation of the context-based instruction optimizer T' of T is presented in figure 7.4.

$$T'(\sigma, i_0;\ i_1;\ \ldots;\ i_n) \quad = \quad T(\sigma, i_0);\ T'(\lambda_2(n_d, \sigma, i_0), i_1;\ \ldots;\ i_n)$$

$$T'_0\left(\sigma,\ \begin{array}{l}\texttt{if } c_0 \texttt{ then}\\ s_0\\ \texttt{elsif } c_1 \texttt{ then}\\ s_1\\ \ldots\\ \texttt{else}\\ s_m\end{array}\right) \quad = \quad \begin{array}{l}\texttt{if } c_0 \texttt{ then}\\ T'(\sigma, s_0)\\ \texttt{elsif } c_1 \texttt{ then}\\ T'(\sigma, s_1)\\ \ldots\\ \texttt{else}\\ T'(\sigma, s_m)\end{array}$$

$$T'_0\left(\sigma,\ \begin{array}{l}\texttt{case } v\\ \texttt{when } e_0 \texttt{ =>}\\ s_0\\ \texttt{when } e_1 \texttt{ =>}\\ s_1\\ \ldots\\ \texttt{when others}\\ s_m\end{array}\right) \quad = \quad \begin{array}{l}\texttt{case } v\\ \texttt{when } e_0 \texttt{ =>}\\ T'(\sigma, s_0)\\ \texttt{when } e_1 \texttt{ =>}\\ T'(\sigma, s_1)\\ \ldots\\ \texttt{when others}\\ T'(\sigma, s_m)\end{array}$$

$$T'_0\left(\sigma,\ \begin{array}{l}\texttt{while } c \texttt{ loop}\\ s\\ \texttt{end loop}\end{array}\right) \quad = \quad \begin{array}{l}\texttt{while } c \texttt{ loop}\\ T'(\lambda_0(\sigma, s), s)\\ \texttt{end loop}\end{array}$$

The T'_0 function is used to propagate T to branch instructions. For every instruction i that are neither an `if/else` nor `case` nor `while` instruction, $T'_0(i) = T(i)$. The number n_d is the default fuel number (500).

Figure 7.4 *– From context-based transformation to context-based instruction optimizer*

We define the correctness of T as follows: for every abstract machine (σ, ϖ), instruction i and interpretation result r, if $(\sigma, \varpi) \models i \Rightarrow r$ then $(\sigma, \varpi) \models T(i) \Rightarrow r$. Using that construction of T', we proved that if T is correct, then T' is a correct context-based instruction optimizer.

7-IV.C.1 Partial dead code elimination

Partial dead code elimination is a well-known optimization that consists in removing code that is never executed. This optimization helps to reduce the size of the code and to avoid executing irrelevant operations (see example in Figure 7.5). The implementation of this optimizer is based on the information given by the context. The transformation T for this optimization is described in Figure 7.6 The correctness proof of T is based on the definition of partial view and on the correctness of λ_2. Intuitively, we use the fact that if the context is correct and the evaluation of an expression e returns a value v such that is different of `undefined`, it means that the evaluation of e in the runtime environment is v.

Using T and T', we obtain the dead code elimination optimization but the result code can contain misplaced `end_branch` and `converge` instructions. We implemented a function that takes a sequence

Non-optimized code

```
a := assign(true);
if t(a) then
        u := assign(0);
        end_branch(0);
else
        u := assign(2);
        end_branch(1);
end if;
converge(0);
l := assign(0);
begin_loop(1);
while w(l < u) loop
        write(tab, l, a);
        end_iteration;
end loop;
```

Optimized code

```
a := assign(true);
u := assign(0);
l := assign(0);
```

Figure 7.5 *– Dead code elimination example*

of instruction s and returns a sequence of instructions where misplaced `end_branch` and `converge` instructions are removed. We obtained the dead code elimination optimizer using that function, T and T' λ_2 and the adaptation of propagation function P described in figure 7.1.

The dead code optimization is implemented in the Coq file `FlowOptim.v` and the correctness proof of its correctness is proved in the Coq file `PartialViewOptimizers.v`.

7-IV.C.2 Loop unrolling

The unrolling optimization consists in duplicating the body of a loop in order to increase the speed of the program by reducing or eliminating the control instruction of the loop. This technique can be partial when the control instruction is not eliminated or full when the whole loop instruction is replaced by the duplicated body as in the following example:

Non-optimized program

```
v := assign(20);
l := assign(0);
u := assign(5);
begin_loop(0);
while w(l <= u) loop
        write(tab, l, v);
        l := l + 1;
        end_iteration;
end loop;
```

Partial loop unrolling

```
v := assign(20);
l := assign(0);
u := assign(5);
write(tab, l, v);
l := l + 1;
write(tab, l, v);
l := l + 1;
begin_loop(0);
while w(l <= u) loop
        write(tab, l, v);
        l := l + 1;
        end_iteration;
end loop;
```

Full loop unrool

```
v := assign(20);
l := assign(0);
u := assign(5);
write(tab, l, v);
l := l + 1;
write(tab, l, v);
l := l + 1;
write(tab, l, v);
l := l + 1;
write(tab, l, v);
l := l + 1;
write(tab, l, v);
```

Listing 42 *– Loop optimization example*

We implemented a prototype that implements those 2 types of loop unrolling. The strategy of this prototype is to first try to perform full unrolling and if it is not possible, to try to perform partial unrolling. This is due to the fact that optimizations performed by hand proved that full unrolling is efficient but the partial unrolling seemed not to reduce the execution time. As this optimization can loop, we added

$$T\left(\sigma,\ \begin{array}{l}\texttt{if } c_0 \texttt{ then}\\ s_0\\ \texttt{elsif } c_1 \texttt{ then}\\ s_1\\ \ldots\\ \texttt{else}\\ s_m\end{array}\right) = \begin{cases} s_0 & \text{if for every } \varpi,\ (\sigma,\varpi)\vdash c_0 \mapsto \texttt{true}\\ T(\sigma, \texttt{if } c_1 \texttt{ then } s_1 \ \ldots\texttt{else } s_m) & \text{if for every } \varpi,\ (\sigma,\varpi)\vdash c_0 \mapsto \texttt{false}\\ T'_0(\sigma, \texttt{if } c_0 \texttt{ then } s_0 \texttt{ elsif } c_1 \texttt{ then } s_1 \ \ldots\texttt{else } s_m) & \text{Otherwise}\end{cases}$$

$$T\left(\sigma,\ \begin{array}{l}\texttt{case } v\\ \texttt{when } e_0 \texttt{ =>}\\ s_0\\ \texttt{when } e_1 \texttt{ =>}\\ s_1\\ \ldots\\ \texttt{when others}\\ s_m\end{array}\right) = \begin{cases} s_0 & \text{if for every } \varpi,\ (\sigma,\varpi)\vdash v \texttt{ = } e_0 \mapsto \texttt{true}\\ T\left(\sigma,\ \begin{array}{l}\texttt{case } v\\ \texttt{when } e_1 \texttt{ =>}\\ s_1\\ \ldots\\ \texttt{when others}\\ s_m\end{array}\right) & \text{if for every } \varpi,\ (\sigma,\varpi)\vdash v \texttt{ = } e_0 \mapsto \texttt{false}\\ T'_0\left(\sigma,\ \begin{array}{l}\texttt{case } v\\ \texttt{when } e_0 \texttt{ =>}\\ s_0\\ \texttt{when } e_1 \texttt{ =>}\\ s_1\\ \ldots\\ \texttt{when others}\\ s_m\end{array}\right) & \text{Otherwise}\end{cases}$$

$$T\left(\sigma,\ \begin{array}{l}\texttt{while } c \texttt{ loop}\\ s\\ \texttt{end loop}\end{array}\right) = \begin{cases} \texttt{null} & \text{if for every } \varpi,\ (\sigma,\varpi)\vdash c \mapsto \texttt{false}\\ T'_0(\sigma, \texttt{while } c \texttt{ loop } s \texttt{ end loop}) & \text{Otherwise}\end{cases}$$

For every instruction i that are neither an `if/else` nor `case` nor `while` instruction, $T(i) = i$.

Figure 7.6 *– The description of transformation function of the dead code optimization*

a fuel parameter to the transformation function T of this optimization. The formal definition of this transformation can be described as follows:

$$F_u(n,\sigma,c,s) = \begin{cases} (\texttt{null},\sigma) & \text{if for every } \varpi,\ (\sigma,\varpi)\vdash c \mapsto \texttt{false}\\ (s;\ s',\sigma') & \text{if for every } \varpi,\ (\sigma,\varpi)\vdash c \mapsto \texttt{true} \text{ and } F_u(n-1,\lambda_2(n_d,\sigma,s),c,s) = (s',\sigma')\\ \bot & \text{if } n = 0 \text{ or if it exists } \varpi \text{ and } v \text{ such that } v \neq \texttt{true},\ v \neq \texttt{false} \text{ and } (\sigma,\varpi)\vdash c \mapsto v\\ \bot & \text{Otherwise}\end{cases}$$

$$T\left(\sigma,\ \begin{array}{l}\texttt{while } c \texttt{ loop}\\ s\\ \texttt{end loop}\end{array}\right) = \begin{cases} s' & \text{if } s \text{ is duplicable and } F_u(n_d,\sigma,c,s) = (s',\sigma')\\ T'_0(\texttt{while } c \texttt{ loop } s \texttt{ end loop}) & \text{if } F_u(n_d,\sigma,c,s) = \bot\end{cases}$$

For every instruction i that are neither an `if/else` nor `case` nor `while` instruction, $T(i) = i$. The number n_d is the default fuel number (500).

Figure 7.7 *– The description of transformation function of loop full unrolling*

Using T and T', we obtain the dead code elimination optimization but the result code can contain misplaced `end_iteration` and `begin_loop` instructions. We implemented a function that takes a

sequence of instruction *s* and returns a sequence of instructions where misplaced `end_iteration` and `begin_loop` instructions are removed. We obtained the dead code elimination optimizer using that function, T and T' λ_2 and the adaptation of propagation function P described in figure 7.1.

The loop unrolling optimization is implemented in the Coq file `While_unroll.v` and the correctness proof of its correctness is proved in the Coq file `PartialViewOptimizers.v`.

7-IV.C.3 Constant propagation and expression simplification

Constant propagation is a well-known optimization that consists into substituting the values of known constants in expressions at compile time. In our implementation, we associated this optimization with the expression simplification optimization that consists into replacing arithmetic expressions and boolean expressions by expressions which computation executes quicker. Due to VCP Ada syntax rules, we need to pay attention when using this optimization in order to avoid syntax errors.

Initial code

```
1 a := assign(14);
2 b := a + 5;
3 c := a <= z;
```

Optimization with syntax error

```
1 a := assign(14);
2 b := 14 + 5; -- Syntax error
3 c := 14 <= z; --Syntax error
```

Optimization without syntax error

```
1 a := assign(14);
2 b := assign(19);
3 c := z >= 14;
```

Listing 43 *– Constant propagation example*

This optimization uses the context to compute access the values of variables and to suggest expression simplifications. It only modifies *assign* instructions and constant variable declaration. The loop unrolling optimization is implemented in the Coq file `ExpressionSimplifier.v` and the correctness proof of its correctness is proved in the Coq file `PartialViewOptimizers.v`.

7-V Conclusion

This chapter presented the source code (in the form of abstract syntax tree) transformations we implemented and how we proved their correctness regarding the semantic preservation definition (see definition 6-V). The next chapter will present how we combined those transformations with other tools to obtain software that will optimize VCP Ada source code files.

Chapter 8

Formally verified source-to-source optimizer

Chapter 2 presented compilers as software made of three parts: a front-end that is in charge of transforming the source code written in a source language into abstract syntax trees (ASTs), the middle-end that is in charge of performing analysis and optimizations on those ASTs and the back-end that is in charge of the generation of the compiled code that is written in the target language. Chapter 5 presented solutions to implement formally verified compiler front-ends. Chapter 6 and 7 described the formally verified AST optimization we implemented. This chapter focusses on the techniques we used to assemble those components into a compiler with safety guarantees.

8-I Filling the gaps

To obtain an executable program, Coq files must be translated into another programming language, typically OCaml, so that those translated files can be compiled. This translates only Coq functions – Coq propositions are ignored. During our implementation, we declared certain abstract functions and used them to implement other functions. In such situations, we need to implement those functions. This section describes the implementation details of the VCP Ada preprocessor used for context-based optimization, the `duplicate` and the `decider` functions presented in section 7-I.

8-I.A The implementation of a preprocessor

A typical input for the compiler this thesis aims to build would be a directory that contains a set of source code of a program. As Ada, VCP Ada is modular. This means that there are dependencies – materialized by import statements whose syntax is described by `<with-statement>` in Section 3-I.B.4 – between source code files. Those dependencies are processed and used by the preprocessor to generate *contexts* for context-based optimizations.
The task of the preprocessor can be illustrated as follows: given the program structure in Figure 8.1 on page 116, the preprocessing leads to the following content:

Variable	Preprocessed value	Elements it depends on
`a.a1`	1	
`b.b1`	2	
`c.c1`	2	(`a.a1`) a.ads
`c.c2`	5	
`d.d1`	7	(`c.c1`) c.ads; (`c.c2`) c.ads
`e.e1`	6	(`a.a1`) a.ads; (`c.c2`) c.ads
`e.e2`	4	(`b.b1`) b.ads; (`c.c1`) c.ads

Table 8.1 *– Preprocessing the variable values*

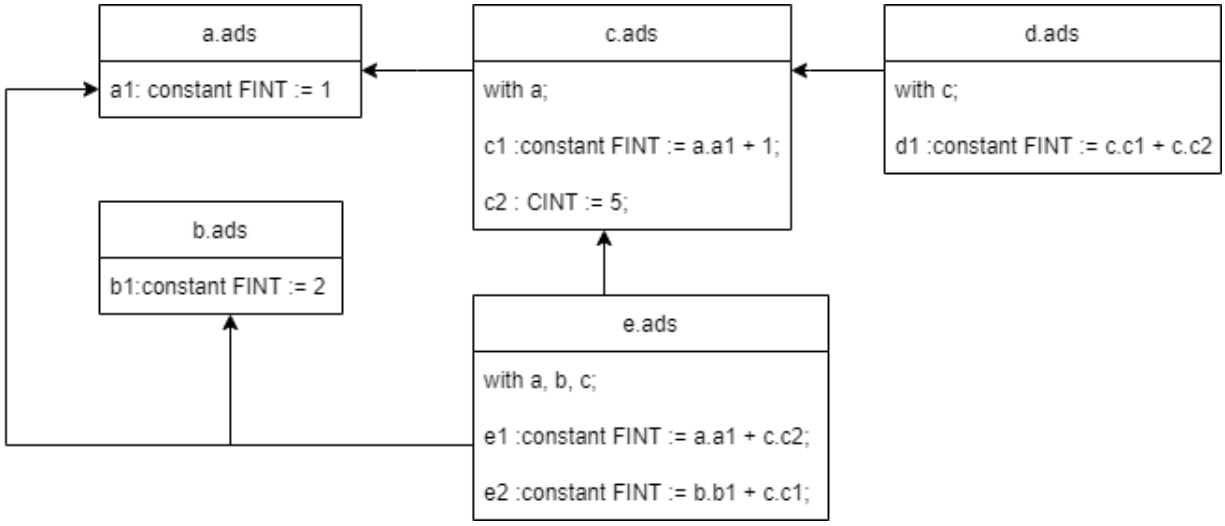


Figure 8.1 *– Example of a VCP Ada program structure*

To execute this task, we implemented two functions:

- **the evaluation of a source code:** This function is implemented in Coq and is named `eval_unit`. It takes an environment in which all the external objects are initialized, a source code – in the form of abstract syntax tree (AST) – and returns an environment that contains the global variables and functions defined in that source code. This function also performs the computation of constant variables values.
- **the exportable variables:** We implemented a Coq function named `extract` that takes a source code and an environment as input. This function returns the set of global variables and function declaration if the input source is a package declaration, and the empty set otherwise.

Using those functions, we used OCaml hashtables to associate filenames with their AST (`<string, AST> table`) and filenames with environments (`<string, environment> table`) in which external variables and procedures are the global variables and procedures declared in the imported packages. Using those constructions, we perform preprocessing in OCaml as follows:

1. we create and fill the `<string, AST> table` by parsing (using the Coq lexer and parser) one by one, the files that are located in the input directory
2. we used topological sorting [Kahn, 1962], based on depth-first search [Tarjan, 1972], to order the files that are located in the input directory in the way that for all files f_1 and f_2, if f_2 imports (or depends on) f_1, then f_1 comes before f_2 in the result list. The dependency is evaluated by analysing the import statement that can be obtained by analysing the AST that can be found in the `<string, AST> table`. This implementation is done in Ocaml.
3. following that order, we fill the `<string, environment> table` as follows:
   - if the file to analyse imports no file, then we use the `eval_unit` function (implemented in Coq) with the empty environment (an environment that contains no variables or procedure) and its associated AST to compute the environment this file will be associated with in the `<string, environment> table`.
   - if the file to analyse f imports files, then we can deduct that those files have already been analysed. Consequently, their environment is already computed and can be accessed looking in the `<string, environment> table`. For each file that f imports, we use the `extract` function

(implemented in Coq) and the environment of the imported file to extract the variables and procedures that are made visible by the importation. Those variables and procedures are added to the empty environment and the result environment is used with the `eval_unit` function to obtain the environment of f.

To illustrate this processing on the structure described in figure 8.1, we will use the notation $\{|\, a_1 \mapsto v_1;\ \ldots\ a_n \mapsto v_n;\ p_1.a_1 \mapsto v_{p_1.a_1};\ \ldots\ p_1.a_m \mapsto v_{p_1.a_m};\ p_2.a_1 \mapsto v_{p_2.a_1};\ \ldots\ p_2.a_o \mapsto v_{p_2.a_o};\ \ldots |\}$ to refer to an environment that contains the global variables a_1, ... a_n associated with the values v_1, ... v_n and the external variables $p_1.a_1$ ... $p_1.a_m$ declared in the package p_1 and associated with the values $v_{p_1.a_1}$ ... $v_{p_1.a_1}$, $p_2.a_1$ ... $p_2.a_o$ declared in the package p_2 and associated with the values $v_{p_2.a_1}$ ... $v_{p_2.a_o}$ and so on. The notation $\{||\}$ refers to the empty environment.
That processing can be executed as follows:

1. The topological order: the dependencies can be defined as follows: `a.ads` and `b.ads` have no dependencies, `c.ads` depends on `a.ads`, `d.ads` depends on `c.ads` and `e.ads` depends on `a.ads`, `b.ads` and `c.ads`. One possible topological order would be `a.ads`, `c.ads`, `d.ads`, `b.ads` and `e.ads`.

2. Computation of the preprocessed values:

   (a) **a.ads**: The file a.ads has no dependencies. The preprocessed environment that will be stored uses the result of `eval_unit` called with $\{||\}$ and the AST of a.ads stored in `<string, AST> table`. This result leads to the result environment $\{|\, a1 \mapsto 1;|\}$ that is stored in the `<string, environment> map`.

   (b) **c.ads**: The file c.ads only depends on a.ads. For that kind of file, we get the external variables that are made visible by a.ads and stored as global variables in the environment stored in the `<string, environment> map` using the `extract` function. Then we use `eval_unit` with the environment $\{|\, a.a1 \mapsto 1|\}$ and the AST of c.ads. This call computes the `c1` that is `a.a1 + 1 = 1 + 1 = 2` and the value of `c2` that is `5`. The environment that will be stored in the `<string, environment> map` is $\{|\, c1 \mapsto 2;\ c2 \mapsto 5;\ a.a1 \mapsto 1;|\}$.

   (c) **d.ads**: Using the same method as before, calling `extract` with $\{|\, c.c1 \mapsto 2;\ c.c2 \mapsto 5;|\}$ and the AST of d.ads as parameters will contribute to compute the value of `d1` is `c.c1 + c.c2 = 2 + 5 = 7` and will lead to store the environment $\{|\, d1 \mapsto 7;\ c.c1 \mapsto 2;\ c.c2 \mapsto 5;|\}$ in the `<string, environment> table`.

   (d) **b.ads**: Using the same technique as in (a), the environment that will be stored in the `<string, environment> table` is $\{|\, b1 \mapsto 2;|\}$

   (e) **e.ads**: Using the same technique as in (b) or (c), `eval_unit` will be called with $\{|\, a.a1 \mapsto 1;\ b.b1 \mapsto 2;\ c.c1 \mapsto 2;\ c.c2 \mapsto 5;|\}$ and the AST of e.ads as parameter and the value that will be stored is $\{|\, e1 \mapsto 6;\ e2 \mapsto 4;\ a.a1 \mapsto 1;\ b.b1 \mapsto 2;\ c.c1 \mapsto 2;\ c.c2 \mapsto 5;|\}$.

Remarks and notes:

- This preprocessing is an implementation of the process described in section 6-III.D.
- In case of file access problem such as file not found error or missing declaration problems (described in Section 6-I) the files in question are associated with an empty environment.
- In Compcert, this preprocessing is performed by an external tool that takes C source code and produces C code that contains the declaration of the external functions and variables used in the code to process. The absence of a similar tool for VCP Ada lead us to build this preprocessor.
- This preprocessing is critical because it can potentially introduce errors with context based optimizations. For instance, when a constant variable is not read or evaluated correctly. However, only two of the four components/functions used for the implementation of the preprocessor are not verified: the OCaml hashtables, used to store AST and environments, and the topological order, used to reduce the execution time of the preprocessing.

The Coq functions presented this section are implemented in the Coq file named `Preprocessor.v` and the OCaml ones are implemented in the OCaml file named `main.ml`.

8-I.B Implementing the abstract function related to VCP restrictions

Section 7-I presented restriction that are posed by the current implementation of Vital Coded Processor technique. During the implementation of optimizations, we used abstract functions to take into account those restrictions.

The first function is the `duplicate` function that takes a sequence of instructions and returns `true` if that sequence of instructions is *non-duplicable* and `false` otherwise. As a reminder, a non-duplicable sequence of instruction is a sequence of instructions that contains the `converge` instruction or a procedure call whose body contains a non-duplicable sequence of instruction. We analyse the source code in the input directory following the order of the sorted list described in section 8-I.A and used a `<string, bool> table` to associate procedure names with booleans (`true` if the body of that procedure contains a non-duplicable sequence of instructions, `false` otherwise). Then, we implemented a function that uses that table to analyse sequences of instructions and say whether or not input sequences of instructions are non-duplicable or not.

The second function is the `decider` function that takes two sequences of instructions (the optimized one and the original one) and returns `true` if the optimization is accepted and `false` otherwise. This function must take into account three restrictions:

1. **the end-branch limit:** Given an `if/else` or `case` instruction, the number of different branches must be lower than 100. To take that restriction in account, we implemented a function that counts the number of branches and returns true if the number of branches of each `if/else` or `case` instruction of the optimized sequence of instructions is lower than 100.

2. **the non-compensable variable limit:** If a local variable is modified inside a block (a branch of a `if/else` or `case` instruction or the body of a `while`) then that variable cannot be read without reassigning after that bloc. To detect those cases, we implemented a function to detect when a variable is modified inside a bloc then we ensure that this variable is not used after that block. We implemented that function using the native `set` data type of OCaml and a `<string, signature> table` to detect when variables are modified inside functions.

3. **the code size limit:** The maximal size of a source code file must be less than 2Mb. We used a `<string, int> table` to associate filenames to their size. To take that restriction in account, we implemented a function that updates that table. That function computes the size of the source code if that optimization is accepted and check if that size is less than 2Mb. If it is the case, it updates the `<string, int> table`.

The `decider` function is implemented as follows: first, we call the function that implements the *end-branch limit*, then the one that implements the *non-compensable variable limit* and then the one that implements the *code size limit*. If one of those functions returns `false`, the optimization is refused.

We also defined an abstract function for the inline optimization. This function takes a procedure name and returns `true` if an inline pragma should be added for this procedure and `false` otherwise. This function uses the `<string, AST> table` to get the definition of that function and verifies that its body contains less than three (this number is read from a configuration file) instructions. If it is the case, then the defined function returns `true` and `false` otherwise.

All the functions presented in this section are implemented in the OCaml file named `deciders.ml`.

8-II The code generation

The code generation aims to generate the VCP Ada code from abstract syntax trees (AST). In other terms, this task is the reciprocal function of the front-end (lexer + parser). Due to the complexity of the parser

(generated using Menhir) and the use of non formally verified modules for code generation, our work does not provide formal proofs for this task. However, we can take advantage on the fact that the input language of the compiler is the same as the output one. Considering the `print` function that are in charge of the code generation and the `ast_of_code` function that is the formally verified front-end of our compiler, the correctness of the code generation can be stated as follows: `print` is correct means that for all AST a, the result of `ast_of_code(print(`a`))` must be syntactically equivalent to a. Instead of proving that in Coq, we check this property at every code generation.

This verification does not prove that the code generator is fault-free but it ensures that for every code we process, the generated output code is correct regarding the verified front-end. The advantage of that verification also known as translation validation [Siegel et al., 1998] is that it can be done in a OCaml program with a simple if/else instruction.

The code generation function is implemented in the OCaml file named `prettyPrinter.ml` and the verification is performed in `main.ml`.

8-III Putting components all together

The compilation process is orchestrated by an OCaml program that works as follows:

1. **Getting the directories ready**: the program starts by reading the input directory to get the filename of the VCP Ada source files that are located in that directory. It also creates the output directory if necessary.

2. **Reading the VCP Ada source files**: Then the program parses those source files and uses a `<string, AST> table` associates those filenames to their abstract syntax tree (AST). Files whose parsing failed due to the missing declaration problem (see section 6-I) are copied in the output directory and removed from the list of files. During that process, the `<string, signature> table` and the `<string, int> table` are initialized and updated.

3. **Computing the topological order**: Then we compute the topological order presented in section 8-I.A.

4. **Computing the environment**: Using the topological sort, we initialize the `<string, environment> table` and `<string, bool> table` that are important for context-based optimizations.

5. **Applying optimizations**: Using the topological sort and all the tables we initialized, we apply optimization in that order: *expression-simplification*, *dead-code elimination*, *loop unrolling*, *converge-2*, *converge-1* and *inlining*. Each optimization updates the `<string, AST> table` by replacing ASTs by their optimized versions.

6. **Code generation**: We generate VCP Ada source files from the `<string, AST> table` and store them into the output directory.

Remarks:

- The composition of correct optimizations is correct (see transitivity).

- The order of the optimization are decided arbitrarily. They come from a test program we used for testing non-verified prototypes.

8-IV Conclusion

The components described in Chapter 5 allow us to build a front-end that allows to convert source code text into an AST. Chapter 7 describes solutions to modify this AST and all those components are used all together and complemented by the components described in this chapter, leading to a VCP Ada to VCP Ada compiler. The next chapter will focus on the evaluation of the performance of that compiler and will conclude our research work.

Chapter 9

Evaluation and conclusion

The railway field imposes that software, whose actions can cause serious damage, provide guarantees of their safety. As a railway control system provider, Siemens Mobility successfully designed and implemented a development process leading to SIL-4 (the highest level of security certification) software. The implementation of this process (presented in Chapter 3) lead Siemens engineers to the development of a restrictive programming language, a subset of Ada called VCP Ada, that is compiled by a Siemens internal tool called the Signature Predetermination Tool (SPT), in addition to the standard compilation using a qualified Ada compiler. This programming language is associated with a redundant compilation and execution process that ends up producing programs whose worst case execution time can reach in the order of hundreds of milliseconds, a value that can be interpreted as the reaction time of the auto pilot software in a driverless train system and that has a great impact on the quality of service of the whole system.

The aim of this research work is to design and implement solutions to reduce that value, in the context of a real-world (i.e. complex) system, while at the same time not loosing the safety guarantees needed at SIL-4 level.

Our research started by analysing the development process and looking for optimization possibilities that conserve the same safety guarantees. Chapter 4 focused on those possibilities and suggested to optimize VCP Ada source code, leading us to the implementation of a VCP Ada to VCP Ada compiler equipped with formal proofs of its correctness. This compiler is made of three parts:

1. **The front-end**: The front-end is made of a lexer and a parser written in Coq and equipped with Coq proof of their correctness (see Chapter 5). Each of those components is generated using tools:
   - The lexer is generated using Coqlex, a library and tool implemented during this thesis, used to generate a formally verified lexer in Coq. We compared Coqlex with two other lexer tools and libraries: OCamllex, an Ocaml (not formally verified) lexer generator, and Verbatim++, a Coq library for formally verified lexer development. Generally, the lexer generated using Coqlex executes 15 times faster than those written using Verbatim++ and between 10 and 100 times slower than those generated with Ocamllex, which is surprisingly good and does not pose limitations to its usefulness in real-world settings. The main advantages of Coqlex are that Coqlex provides protections against infinite loops (see Section 5-I.F.1), Coq proofs of the correctness of the generated lexers, most of the features implemented by not formally verified lexer generator and the possibility to write additional Coq proofs on the generated lexers.
   - The parser is generated using Menhir, a parser generator that produces parsers equipped with a formal proof of their correctness and completeness.

   Associated with a lexer, a parser aims to transform a source code text into an abstract syntax tree (AST).
2. **The optimization**: We used Coq inductive properties to axiomatize the execution of VCP Ada programs and the semantic preservation property that is the property that ensures that optimizations,

implemented in the form of AST to AST functions, are correct (see Chapter 6). We implemented five optimizations and proved that they are correct. For those correctness proofs, we wrote a big step operational semantics for VCP Ada and a semantic preservation theorem for that semantics. The data structure used to represent the values in that semantics is inspired from the sparkformal project that aims at developing semantics of the SPARK language in Coq. By comparison, the semantics we implemented takes into account the semantics of the function defined in the VCP Lib (see Section 3-I.A) and the modularity of VCP Ada while the semantics of the sparkformal project is focused on the type-check of SPARK, the detection of runtime errors, allows sub-procedure definition (procedures defined inside procedures) but cannot handle modular programs –that semantics supposes that all the code of the program it analyses is written in a unique file. Compcert uses more sophisticated structures to define the small step semantics for each of the 8 languages it processes.

3. **Code generation**: We extracted the Coq code of the two first components into OCaml and used an OCaml library to implement the code generation function that converts ASTs into VCP Ada source code. This function does not provide a formal proof of its correctness. However, such a proof is not necessary, since we verify, at every code generation, that the generated code, when lexed/parsed with our proven lexer/parser leads to an AST identical to the AST used for code generation.

Performing optimizations using a source-to-source compiler may have advantages compared to using a source-to-binary compiler:

- semantics preservation relies on unique formal semantics instead of using one semantics for the source and another semantics for the target language (and for all the intermediate languages). For instance, the Compcert project implemented 8 formal semantics for each of the 10 intermediary languages it manipulates.
- another advantage is that using a source-to-source compiler allows performing optimizations without significant impact on the development/validation toolchain, particularly in the case of the Siemens development process that combines the use of a source-to-binary compiler and a signature predetermination tool (SPT) to obtain the executable. This combination restricts the optimizations that can be performed by the source-to-binary because it can cause incompatibility with the output generated by the SPT. Performing optimization using a source-to-source provides more optimization options while preserving compatibility with the SPT.
- optimizations are human-readable and human-verifiable, which is very important for safety critical applications.

We also implemented OCaml functions that aim to authorize or forbid optimizations regarding the restrictions of the VCP Ada language. We decided to implement those functions without formal verification because an error in those functions will be detected by the SPT. There are three particular restrictions that we took in account for the implementation of those functions: the *size* limit, the *end-branch* limit, the *non-compensable variable* and the *code duplication* restriction (See section 7). Until now, we did not see an example in which those functions forbid an optimization that would be accepted or authorize an optimization that would be rejected.

The implementation of this VCP Ada to VCP Ada compiler used 17722 lines of Coq code and 2856 lines of OCaml code.

Executing realistic programs generated using the Siemens development and execution process requires specific hardware that is less and less available for testing due to industrial reasons. This limited the testing opportunities. However, we used the VCP Ada to VCP Ada compiler we implemented on a test

program. The use of that test program on accurate (but incomplete) hardware allowed us to choose the optimizations we will implement. For example, this test program showed us the importance of removing converge instructions and implementing expression simplification. It also showed us that partial loop unrolling was not efficient.

We also tested the VCP Ada to VCP Ada compiler on a realistic program and noticed the modification of 458 pieces of code for 1136 optimization opportunities that are related to the removal of *converge* instructions. Of the 678 opportunities that have been rejected, 111 were rejected because of the *non-compensable variable* restriction, 17 were rejected for *end-branch* limit and 550 were rejected because of the *code duplication* restriction. This allowed removing 12% of the converge instruction, to fully unrolling 2 while instructions, to simplify 14% of expressions. The generated code is accepted by the SPT, which generates a report showing a reduction of the number of compensation tables by 10% (1123) and a reduction of 1% (1019) in the number of variables compensated by those tables.

The generated code passes all of the numerous non-regression tests. The industrial reasons causing the unavailability of specific hardware needed to run and measure the execution time of the auto-pilot program we optimized make it difficult to measure the impact of the optimizations we implemented on the worst-case execution time. This measurement remains future work but the execution time should directly depend on the percentage of removed `converge` and parts of the compensation tables.

The VCP Ada to VCP Ada compiler we implemented can be improved in various ways:

- **Making the compiler more restrictive**: Our implementation allows to optimize programs without accessing all its source code files. This decision is a consequence of the missing declaration problem (see Section 6-I). Extending the parser to analyse non-VCP Ada files and obliging users to provide all the source code of the program to optimize would allow to obtain more precise contexts (see Section 7-IV.A) and to ignore most VCP restrictions.
- **The management of the VCP Ada modularity**: In our implementation, files are analysed one by one and as in Ada specification, only declarations of the imported files are visible (see Section 8-I.A). However, allowing to access function definitions can lead to more precise contexts and then to optimization opportunities.
- **Allowing to the optimizer to create/remove variables**: Our implementation does not allow optimizers to create or remove variables. This is linked to the difficulty to generate unique strings and the missing declaration problem. Solving this problem with associated Coq proofs would allow to ignore the problem related to the *non-compensable variable* restriction and to implement other optimizations such that *common sub-expression elimination* or *dead-store elimination*.
- **Redefining the partial view relation**: The current implementation of the partial view relation applies the all-or-nothing law: this means for example that for a given record or array variable, either the context contains the exact values of all its fields or items, or the context contains no value of this record or array. Changing this relation to allow context to contain partial values would lead to more precise contexts and thus to more optimization opportunities.

In spite of this, our work already provides a way to securely optimize CBTC Ada VCP code on a realistic industrial scale. It can also serve as a platform for future optimizations of CBTC, or beyond, by demonstrating that proven source-to-source compiling is an effective method for safe program optimization.

Bibliography

[Abate et al., 2015] Abate, P., Di Cosmo, R., Gesbert, L., Le Fessant, F., Treinen, R., and Zacchiroli, S. (2015). Mining component repositories for installability issues. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 24–33. IEEE.

[Abrial, 1996] Abrial, J.-R. (1996). *The B-book: assigning programs to meanings*, volume 1. Cambridge university press Cambridge.

[Abrial, 2010] Abrial, J.-R. (2010). *Modeling in Event-B: system and software engineering*. Cambridge University Press.

[Adams et al., 2016] Adams, M. D., Hollenbeck, C., and Might, M. (2016). On the Complexity and Performance of Parsing with Derivatives. *SIGPLAN Not.*, 51(6):224–236.

[Adewumi, 2018] Adewumi, T. P. (2018). Inner loop program construct: A faster way for program execution. *Open Computer Science*, 8(1):115–122.

[Aponte et al., 2014] Aponte, V., Courtieu, P., and Zhang, Z. (2014). Formal semantics and other research around Spark2014.

[Armstrong et al., 2013] Armstrong, A., Struth, G., and Weber, T. (2013). Kleene algebra. *Archive of Formal Proofs*, 324.

[Ayatolahi et al., 2013] Ayatolahi, F., Sangchoolie, B., Johansson, R., and Karlsson, J. (2013). A study of the impact of single bit-flip and double bit-flip errors on program execution. In *International Conference on Computer Safety, Reliability, and Security*, pages 265–276. Springer.

[Barnes, 1984] Barnes, J. (1984). *Programming in ADA*. Addison-Wesley Longman Publishing Co., Inc.

[Barnes, 2012] Barnes, J. (2012). *SPARK: The Proven Approach to High Integrity Software*. Altran Praxis.

[Barras et al., 1997] Barras, B., Boutin, S., Cornes, C., Courant, J., Filliâtre, J.-C., Giménez, E., Herbelin, H., Huet, G., Muñoz, C., Murthy, C., Parent, C., Paulin-Mohring, C., Saïbi, A., and Werner, B. (1997). The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA. Projet COQ.

[Barth, 1978] Barth, J. M. (1978). A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736.

[Becchi and Crowley, 2013] Becchi, M. and Crowley, P. (2013). A-DFA: A Time- and Space-Efficient DFA Compression Algorithm for Fast Regular Expression Evaluation. *ACM Trans. Archit. Code Optim.*, 10(1).

[Behm et al., 1999] Behm, P., Benoit, P., Faivre, A., and Meynadier, J.-M. (1999). METEOR: A successful application of B in a large project. In *International Symposium on Formal Methods*, pages 369–387. Springer.

[Bergerand, 1986] Bergerand, J.-L. (1986). *LUSTRE: un langage déclaratif pour le temps réel*. PhD thesis, Institut National Polytechnique de Grenoble-INPG.

[Bertot, 2008] Bertot, Y. (2008). A short presentation of Coq. In *International Conference on Theorem Proving in Higher Order Logics*, pages 12–16. Springer.

[Blazy, 2008] Blazy, S. (2008). *Sémantiques formelles*. Habilitation à diriger des recherches, Université d'Evry-Val d'Essonne.

[Boulanger, 2015] Boulanger, J.-L. (2015). *CENELEC 50128 and IEC 62279 standards*. John Wiley & Sons.

[Bourke, 2021] Bourke, T. (2021). *Specification and End-to-End Proof of a Reactive Language and Its Compiler (Invited Talk)*, page 1. Association for Computing Machinery, New York, NY, USA.

[Brandt and Henglein, 1997] Brandt, M. and Henglein, F. (1997). Coinductive axiomatization of recursive type equality and subtyping. In *International Conference on Typed Lambda Calculi and Applications*, pages 63–81. Springer.

[Brosgol and Comar, 2010] Brosgol, B. and Comar, C. (2010). DO-178C: A new standard for software safety certification. Technical report, ADA CORE TECHNOLOGIES NEW YORK NY.

[Brown, 1988] Brown, R. F. (1988). *Fixed Point Theory and Its Applications: Proceedings of a Conference Held at the International Congress of Mathematicians, August 4-6, 1986*, volume 72. American Mathematical Soc.

[Brzozowski, 1964] Brzozowski, J. A. (1964). Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494.

[Cansell and Méry, 2003] Cansell, D. and Méry, D. (2003). Foundations of the B method. *Computing and Informatics*, 22:31 p. Article dans revue scientifique avec comité de lecture.

[Chapront, 1992] Chapront, P. (1992). Vital coded processor and safety related software design. In *Safety of Computer Control Systems 1992 (Safecomp'92)*, pages 141–145. Elsevier.

[Chlipala, 2013] Chlipala, A. (2013). *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press.

[Clark, 2002] Clark, C. (2002). Conflicts. *ACM SIGPLAN Notices*, 37(8):9–14.

[Clearsy, 2003] Clearsy (2003). Atelier B, Manuel Utilisateur.

[Clément et al., 1985] Clément, D., Despeyroux, J., Despeyroux, T., Hascoet, L., and Kahn, G. (1985). *Natural semantics on the computer*. PhD thesis, INRIA.

[Coco et al., 2018] Coco, E. J., Osman, H. A., and Osman, N. I. (2018). JPT: ASimple JAVA-PYTHON TRANSLATOR. *Computer Applications: An International Journal (CAIJ)*, 5(2).

[Consel and Danvy, 1993] Consel, C. and Danvy, O. (1993). Tutorial Notes on Partial Evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, page 493–501, New York, NY, USA. Association for Computing Machinery.

[Cooper et al., 1991] Cooper, K. D., Hall, M. W., and Torczon, L. (1991). An experiment with inline substitution. *Software: Practice and Experience*, 21(6):581–601.

[Courtieu et al., 2013] Courtieu, P., Aponte, M. V., Crolard, T., Roby, J. B., Hatcliff, J., Zhang, Z., Guitton, J., and Jennings, T. (2013). Towards the formalization of SPARK 2014 semantics with explicit run-time checks using coq. In *High Integrity Language Technology ACM SIGAda's Annual International Conference (HILT 2013)*. ACM.

[Cremers and Ginsburg, 1975] Cremers, A. and Ginsburg, S. (1975). Context-free grammar forms. *Journal of Computer and System Sciences*, 11(1):86–117.

[Dargaye, 2009] Dargaye, Z. (2009). *Vérification formelle d'un compilateur optimisant pour langages fonctionnels*. Theses, Université Paris-Diderot - Paris VII.

[Dollé, 2006] Dollé, D. (2006). Vital software: Formal method and coded processor. In *Conference ERTS'06*, Toulouse, France.

[Egolf et al., 2021] Egolf, D., Lasser, S., and Fisher, K. (2021). Verbatim: A verified lexer generator. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 92–100.

[Egolf et al., 2022] Egolf, D., Lasser, S., and Fisher, K. (2022). Verbatim++: Verified, Optimized, and Semantically Rich Lexing with Derivatives. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 27–39, New York, NY, USA. Association for Computing Machinery.

[Falk and Marwedel, 2004] Falk, H. and Marwedel, P. (2004). *Source code optimization techniques for data flow dominated embedded software*. Springer Science & Business Media.

[Gall, 2004] Gall, H. (2004). The TUV approach to functional safety assessment and certification.

[Gellerich and Plödereder, 2001] Gellerich, W. and Plödereder, E. (2001). Parameter-induced aliasing in Ada. In *Reliable SoftwareTechnologies—Ada-Europe 2001: 6th Ada-Europe International Conference on Reliable Software Technologies Leuven, Belgium, May 14–18, 2001 Proceedings 6*, pages 88–99. Springer.

[Geuvers, 2009] Geuvers, H. (2009). Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25.

[Gilbert et al., 2019] Gilbert, G., Cockx, J., Sozeau, M., and Tabareau, N. (2019). Definitional Proof-Irrelevance without K. *Proc. ACM Program. Lang.*, 3(POPL).

[Grove and Torczon, 1993] Grove, D. and Torczon, L. (1993). Interprocedural constant propagation: A study of jump function implementation. *ACM SIGPLAN Notices*, 28(6):90–99.

[HaskellWiki, 2020] HaskellWiki (2020). Smart constructors — haskellwiki,. [Online; accessed 10-March-2022].

[Haughton and Lano, 1996] Haughton, H. and Lano, K. (1996). *Specification in B: An introduction using the B toolkit*. World Scientific.

[Hoare, 1969] Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580.

[Jansen and Schäbe, 2004] Jansen, H. and Schäbe, H. (2004). Computer architectures and safety integrity level apportionment. *WIT Transactions on The Built Environment*, 74.

[Johnson et al., 1975] Johnson, S. C. et al. (1975). *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ.

[Jones, 2005] Jones, M. T. (2005). Optimization in GCC. *Linux journal*, 2005(131):11.

[Jourdan et al., 2012] Jourdan, J.-H., Pottier, F., and Leroy, X. (2012). Validating LR(1) parsers. In *ESOP 2012: Programming Languages and Systems, 21st European Symposium on Programming*, number 7211 in LNCS, pages 397–416. Springer.

[Kahn, 1962] Kahn, A. B. (1962). Topological Sorting of Large Networks. *Commun. ACM*, 5(11):558–562.

[Kahn, 1987] Kahn, G. (1987). Natural semantics. In *Annual symposium on theoretical aspects of computer science*, pages 22–39. Springer.

[Kane, 1988] Kane, G. (1988). *mips RISC Architecture*. Prentice-Hall, Inc.

[Karttunen et al., 1996] Karttunen, L., Chanod, J.-P., Grefenstette, G., and Schille, A. (1996). Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.

[Kleene et al., 1956] Kleene, S. C. et al. (1956). Representation of events in nerve nets and finite automata. *Automata studies*, 34:3–41.

[Knight, 2002] Knight, J. (2002). Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 547–550.

[Kozen, 1997] Kozen, D. (1997). Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):427–443.

[Krebbers et al., 2014] Krebbers, R., Leroy, X., and Wiedijk, F. (2014). Formal C semantics: CompCert and the C standard. In *International Conference on Interactive Theorem Proving*, pages 543–548. Springer.

[Kumar et al., 2014] Kumar, R., Myreen, M. O., Norrish, M., and Owens, S. (2014). CakeML: a verified implementation of ML. *ACM SIGPLAN Notices*, 49(1):179–191.

[Ladkin, 2008] Ladkin, P. B. (2008). An overview of IEC 61508 on E/E/PE functional safety. *Bielefeld, Germany*.

[Lano, 2012] Lano, K. (2012). *The B language and method: a guide to practical formal development*. Springer Science & Business Media.

[Lebras, 2019] Lebras, Y. (2019). *Code optimization based on source to source transformations using profile guided metrics*. Theses, Université Paris-Saclay.

[Lejeune et al., 2004] Lejeune, P., Dufour, F., and Chenu, E. (2004). DIGISAFE® XME: High Availability Vital Computer. In *2nd Embedded Real Time Software Congress (ERTS'04)*.

[Leroy, 2011] Leroy, X. (2011). Formally verifying a compiler: Why? How? How far? In *CGO 2011-9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page xxxi. IEEE.

[Leroy et al., 2016] Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., and Ferdinand, C. (2016). Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

[Letouzey, 2002] Letouzey, P. (2002). A new extraction for Coq. In *International Workshop on Types for Proofs and Programs*, pages 200–219. Springer.

[Leuschel and Butler, 2008] Leuschel, M. and Butler, M. (2008). ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203.

[Levine et al., 1992] Levine, J. R., Mason, J., Levine, J. R., Mason, T., Brown, D., Levine, J. R., and Levine, P. (1992). *Lex & yacc*. " O'Reilly Media, Inc.".

[McCracken and Reilly, 2003] McCracken, D. D. and Reilly, E. D. (2003). *Backus-Naur Form (BNF)*, page 129–131. John Wiley and Sons Ltd., GBR.

[McKeeman, 1965] McKeeman, W. M. (1965). Peephole optimization. *Communications of the ACM*, 8(7):443–444.

[Meyer, 1991] Meyer, U. (1991). Techniques for partial evaluation of imperative languages. In *Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 94–105.

[Meyer, 1999] Meyer, U. (1999). Correctness of on-line partial evaluation for a Pascal-like language. *Science of computer programming*, 34(1):55–73.

[Miyamoto, 2011] Miyamoto, T. (2011). Coq Regular Expression git page. https://github.com/coq-contribs/regexp. Accessed: 2020-10-14.

[Myklebust et al., 2015] Myklebust, T., Stålhane, T., and Lyngby, N. (2015). Application of an Agile Development Process for EN50128/railway con-formant Software.

[Nipkow, 1998] Nipkow, T. (1998). Verified lexical analysis. In *International Conference on Theorem Proving in Higher Order Logics*, pages 1–15. Springer.

[Ouedraogo et al., 2021] Ouedraogo, W., Ilik, D., and Straßburger, L. (2021). Coqlex, an approach to generate verified lexers. In *ML 2021-ACM SIGPLAN Workshop on ML*.

[Owens et al., 2009] Owens, S., Sarkar, S., and Sewell, P. (2009). A better x86 memory model: x86-TSO. In *International Conference on Theorem Proving in Higher Order Logics*, pages 391–407. Springer.

[Parikh, 1966] Parikh, R. J. (1966). On Context-Free Languages. *J. ACM*, 13(4):570–581.

[Pezoa et al., 2016] Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., and Vrgoč, D. (2016). Foundations of JSON schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273. International World Wide Web Conferences Steering Committee.

[Potii et al., 2015] Potii, O., Illiashenko, O., and Komin, D. (2015). Advanced security assurance case based on ISO/IEC 15408. In *International Conference on Dependability and Complex Systems*, pages 391–401. Springer.

[Régis-Gianas, 2016] Régis-Gianas, F. P. Y. (2016). Menhir Reference Manual.

[Resler and Winter, 2009] Resler, R. D. and Winter, V. (2009). A Higher-Order Strategy for Eliminating Common Subexpressions. *Comput. Lang. Syst. Struct.*, 35(4):341–364.

[Rinderknecht, 2018] Rinderknecht, C. (2018). A Mini-ML programming language. https://github.com/rinderknecht/Mini-ML/blob/master/Lang0/Lexer.mll.

[Ringer, 2021] Ringer, T. (2021). *Proof Repair*. PhD thesis, University of Washington.

[Ringer et al., 2018] Ringer, T., Yazdani, N., Leo, J., and Grossman, D. (2018). Adapting Proof Automation to Adapt Proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 115–129, New York, NY, USA. Association for Computing Machinery.

[Ritchie et al., 1988] Ritchie, D. M., Kernighan, B. W., and Lesk, M. E. (1988). *The C programming language*. Prentice Hall Englewood Cliffs.

[Rothwell and Youngman, 2007] Rothwell, T. and Youngman, J. (2007). The GNU C reference manual. *Free Software Foundation, Inc*, page 86.

[Ryzhyk, 2006] Ryzhyk, L. (2006). The ARM architecture. *Chicago University, Illinois, EUA*.

[Salomon, 1998] Salomon, D. (1998). The ASCII code. In *Data Compression*, pages 301–303. Springer.

[Schonberg and Banner, 1994] Schonberg, E. and Banner, B. (1994). The GNAT project: a GNU-Ada 9X compiler. In *Proceedings of the conference on TRI-Ada'94*, pages 48–57.

[Scott, 1982] Scott, D. S. (1982). Domains for denotational semantics. In *International Colloquium on Automata, Languages, and Programming*, pages 577–610. Springer.

[Siegel et al., 1998] Siegel, M., Pnueli, A., and Singerman, E. (1998). Translation validation. In *TACAS*, pages 151–166.

[Sikora, 2017] Sikora, A. (2017). Tutorial: parsing JSON with OCaml.

[Smith, 2007] Smith, J. B. (2007). Ocamllex and Ocamlyacc. *Practical OCaml*, pages 193–211.

[Song et al., 1994] Song, S. P., Denman, M., and Chang, J. (1994). The PowerPC 604 RISC microprocessor. *IEEE Micro*, 14(5):8.

[Stroustrup, 1986] Stroustrup, B. (1986). An overview of C++. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, pages 7–18.

[Taft, 1997] Taft, T. S. (1997). *Ada 95 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652: 1995 (E)*, volume 8652. Springer Science & Business Media.

[Tarjan, 1972] Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160.

[Thery, 2020] Thery, L. (2020). Coq String Module documentation. https://coq.inria.fr/library/Coq.Strings.String.html. Accessed: 2020-10-14.

[Thompson, 1968] Thompson, K. (1968). Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422.

[Weng, 2016] Weng, S.-C. (2016). *DeepSpec: Modular Certified Programming with Deep Specifications*. Yale University.

[Wilhelm and Maurer, 1995] Wilhelm, R. and Maurer, D. (1995). *Compiler design*. Springer.

[Yu, 1997] Yu, S. (1997). Regular languages. In *Handbook of formal languages*, pages 41–110. Springer.

DOCTORAL SCHOOL

Titre: Optimisation de code source pour les systèmes critiques de sécurité

Mots clés: Méthodes formelles, Langages de programmation, Compilation, Assistants de preuve, Sémantique formelle

Résumé: Un système informatique est dit critique de sécurité lorsqu'il peut causer de graves dommages matériels, environnementaux ou humains. Ce type de système est nécessairement complexe, car les interactions entre logiciel, matériel et environnement, nécessitent l'implémentation de mesures de sécurité aussi bien au niveau logiciel que matériel pendant leur conception, leur développement, leur compilation et leur exécution. Ces mesures de sécurité qui varient d'un système à un autre conduisent très souvent à des dégradations de performances, en particulier l'augmentation du temps d'exécution.

Ce travail de recherche se situe dans le contexte du système CBTC (Communication-Based Train Control) de Siemens Mobility France, un système de contrôle de train certifié EN-50128 et SIL-4 (le plus haut niveau de sûreté) dans le développement de nombreux systèmes de pilote automatique à travers le monde, en particulier sur les lignes 1, 4 et 14 du métro parisien. Dans ce contexte, le but de cette thèse est de trouver des solutions pour réduire le temps d'exécution de ces systèmes tout en conservant les garanties de sécurité déjà acquises.

La réponse apportée par cette thèse est une optimisation formellement vérifiée du code source. Une première contribution est un compilateur VCP Ada (un sous-ensemble d'Ada) vers VCP Ada, qui optimise le code source tout en préservant la sémantique des programmes. Ce compilateur a été implémenté avec l'assistant de preuve Coq et fourni des preuves en Coq qui garantissent l'équivalence entre le programme original et le programme optimisé. Ce compilateur tient aussi compte des complexités liées aux mesures de sécurité matérielle qui sont potentiellement incompatibles avec l'utilisation des compilateurs formellement vérifiés existants. Par ailleurs, le choix de l'application des optimisations sur le code source présente des avantages méthodologiques par rapport aux optimisations utilisant de nombreux langages intermédiaires, car ils permettent de simplifier et de réduire l'effort de preuve nécessaire.

Une deuxième contribution concerne la vérification formelle des analyseurs lexicaux des compilateurs, qui n'a jusqu'à présent, pas reçu beaucoup l'attention de la part des chercheurs, créant donc un maillon faible dans la chaîne de compilation. Nous avons développé CoqLex, le premier générateur d'analyseur lexical formellement vérifié en Coq, basé sur une modification d'une implémentation existante en Coq des expressions régulières via les dérivés de Brzozowski.

La théorie et les outils développés ont été utilisés pour optimiser les programmes VCP Ada des systèmes CBTC, composés de milliers de fichiers sources, avec des résultats prometteurs.

Institut Polytechnique de Paris
91120 Palaiseau, France

DOCTORAL SCHOOL

Title: Source code optimization for safety-critical systems

Keywords: Formal methods, Programming languages, Compilation, Proof assistants, Formal semantic

Abstract: A computer system is safety-critical when it can cause serious damage to property, the environment, human life, or to society as a whole. Real-world safety-critical systems are also necessarily complex, because, to take into account the interactions between software, hardware, the physical environment, and sometimes their distributed nature (systems of systems), they need to implement a variety of safety measures, in software, hardware, in the system design, at development time, at compile time, and at run-time. Those safety measures which vary from one safety-critical system to another very often lead to a decrease in performance, for an increase in the execution time of software.

This research work is situated in the context of one such system, the communication-based train control (CBTC) system of Siemens Mobility France which operates a number of driverless subway systems around the World, including Paris lines 1, 4, and 14. That system is certified according to the industrial norm EN-50128 and up to the highest Safety Integrity Level 4, required for safety-critical systems with potentially catastrophic consequences. In this context, the thesis looks for an answer to the question of how to automatically optimize the execution time performance of such systems without losing the previously obtained safety guarantees.

The answer provided by this thesis is provably correct optimization of source code. A first contribution is a source-to-source compiler for VCP Ada (a subset of Ada) programs, that optimizes source code while preserving the formal semantics of the programs. The compiler has been proven correct in the Coq proof assistant and guarantees the equivalence of execution between the original and the optimized program. The compiler copes with additional/hardware safety measures addressing random faults and compiler bugs, which is important, since real-world safety-critical systems are often susceptible to having such measures, potentially conflicting with existing formally proven optimizing compilers. Moreover, choosing the approach of a source-to-source compilation shows to have methodological advantages over an approach to proven optimizations having a number of intermediate languages, allowing to simplify and reduce the required proof effort.

A second contribution is to the problem of provably correct lexical analysis of compilers, which has previously not received a lot of research attention, a weak link in any compilation chain using a proven or qualified compiler. We develop CoqLex, the first Coq-formalized lexer generator, based on a modification of an existing Coq implementation of regular expression matching via Brzozowski derivatives.

The developed theory and tools have been applied to optimize real-world VCP Ada programs of CBTC systems, consisting of thousands of source files, with promising results.

Institut Polytechnique de Paris
91120 Palaiseau, France