Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE
ANSSI Agence nationale de la sécurité des systèmes d'information
General Secretariat for Defence and National Security
French Network and Information Security Agency

UPMC PARIS UNIVERSITAS
LIP6
Pierre et Marie Curie University
Paris 6 Computer Sciences Laboratory

Study of the Benefits of Using Deductive Formal Methods for Secure Developments

by *Éric Jaeger*

Thesis submitted for the degree of
Doctor of Philosophy in Computer Sciences

Approved by the Examining Committee:

(Advisors)	*C. Dubois*	–	CEDRIC, ENSIIE
	T. Hardin	–	LIP6, UPMC
(Reviewers)	*G. Dowek*	–	LIX, Ecole Polytechnique
	C. A. Muñoz	–	LaRC, NASA
(Chair)	*B. Bérard*	–	LIP6, UPMC
(Examiners)	*S. Boulmé*	–	VERIMAG
	L. Duflot	–	ANSSI, SGDSN
	F. Kamareddine	–	ULTRA, Heriot-Watt University

EDITE Doctoral School

Defended on 8 March 2010

at LIP6, 104 boulevard Kennedy, 75016 Paris, France

Abstract

The use of formal methods in general, and of deductive formal methods in particular, for the development of systems aims at providing mathematical guarantees, for example about their correctness. For this reason, the use of formal methods is recommended or required by safety or security standards, such as the *IEC 61508* or the *Common Criteria*.

Whereas formal approaches indeed induce important benefits, one may wonder about the exact extent of those. For example, some aspects of a system can be left out of the scope of formalisation, but it may not be easy to identify such restrictions or their consequences. If the validity of mechanically checked proofs is generally accepted, their applicability for justifying actual confidence in the physical systems is often questionned.

These question are especially relevant in the field of security, when the considered systems are targetted by intelligent agents; compared with the field of safety, there is a paradigm shift, preventing the application of well known facts or practices.

This memoir addresses the question of the benefits and the confidence resulting of the application of deductive formal methods for the development or the verification of systems having security requirements. But it also illustrates how it is possible to abuse inadequate specifications or inconsistencies in the theories or the tools, for example allowing a malicious developer to trap a system while providing a formal certificate, or a malicious user to identify possible ways around proofs of compliance. Based on these observations, various strategies are considered to mitigate the risks and improve the overall security of a formally developed system.

More specifically, a detailed study of the concept of refinement is presented, providing a generic vision which is applied to several deductive formal methods. It is used to highlight interesting points, and to understand the essence of possible abuses of the so-called Refinement Paradox. We identify for example the influence of parameters of the refinement relation which are generally left implicit in formal developments.

This memoir also discusses at length the question of the validation of the theories and tools supporting formal methods, through a deep embedding of the B logic in Coq. Several shortcomings of the B theory are identified and corrected – illustrating the interest of such a validation – and a proven prover is developed. New results for the B logic are also presented, dealing with the replacement of subterms whose free variables are bound by the context.

This embedding further leads to technical considerations about the mechanisation of languages, comparing for example various forms of *de Bruijn* representations. We study approaches with finer context management, enriched *de Bruijn* indexes, parallel substitutions, before deriving from our work a representation of the simply typed λ-calculus without typing context.

Remerciements

Je tiens tout d'abord à remercier *Florent Chabaud*, qui en 2004 a considéré avec bienveillance ma candidature d'ingénieur enthousiaste mais non initié et m'a mis au défi de développer une expertise personnelle dans le domaine des méthodes formelles. Je remercie également *Patrick Pailloux*, qui a permis que cette thèse soit menée dans le cadre de mes activités professionnelles à la DCSSI puis à l'ANSSI.

Je remercie *Thérèse Hardin* d'avoir accepté de prendre en thèse un étudiant atypique, aux idées parfois bien arrêtées et ne consacrant qu'une partie de son temps à la recherche. Sa disponibilité, son écoute et ses questions tout autant que ses suggestions ont permis à ces travaux d'aboutir. Je remercie également *Catherine Dubois*, qui a accepté de co-encadrer cette thèse, pour ses nombreux conseils.

Je remercie *Gilles Dowek* et *César Muñoz*, qui ont accepté de rapporter cette thèse et ont contribué par leurs commentaires à son amélioration. Je remercie également *Béatrice Bérard*, *Sylvain Boulmé*, *Loïc Duflot* et *Fairouz Kamareddine* pour avoir accepté d'en être les examinateurs.

Je remercie l'ensemble des personnes que j'ai pu rencontrer au travers des projets FOCALIZE et SSURF, pour de nombreuses discussions – parfois animées – sur la théorie comme sur la pratique de l'informatique et d'autres sciences plus ou moins exactes. Je pense notamment, sans que cette liste ne soit exhaustive, à *Philippe Ayrault*, *Julien Blond*, *Richard Bonichon*, *Mathieu Carlier*, *David Delahaye*, *Véronique Delebarre*, *Damien Doligez*, *Lionel Habib*, *Mathieu Jaume*, *Charles Morisset*, *Ivan Noyer*, *François Pessaux*, *Renaud Rioboo*, *Pierre Weis*. Ces discussions ont souvent contribué, de manière directe ou indirecte, au contenu de cette thèse.

Je remercie également mes collègues de la DCSSI puis l'ANSSI, qui ont su tolérer mes périodes d'introversion comme de prolixité (ces dernières étant sans doute les plus pénibles pour eux). Je remercie notamment *Mathieu Baudet*, *Dominique Chandesris*, *Julie Chuzel*, *Olivier Grumelard*, *Olivier Levillain*, *Louis Mussat* pour des discussions sur les méthodes formelles, ainsi que *Philippe Blot*, *Pierre-Alain Drucbert*, *Loïc Duflot*, *Frédéric Gauche*, *Fabien Germain*, *Sébastien Héon*, *Véronique Joubert*, *Julien Kaci*, *Philippe Le Moigne*, *Benjamin Morin*, *Yves-Alexis Perez*, *Pascale Pouliquen*, *Pierre Durieux*, *Vincent Strubel*, *Guillaume Valadon* pour d'autres discussions toutes aussi instructives sur les différents aspects de la sécurité. Ici encore, les contributions furent nombreuses, trop pour être citées exhaustivement ici.

Enfin, je remercie mon épouse et mes enfants, qui ont accepté les contraintes que représente une telle thèse.

Contents

Chapter 1

Introduction

1.1 Formal methods and high assurance systems

The development of software, equipments or systems has been for long a source of engineering challenges, and various forms of flaws, either in the specification, the design or the implementation, are frequent. Such flaws are at best nuisances, but for systems having strong safety or security requirements, the consequences can be inacceptable: huge financial losses, injuries or even deaths.

Numerous approaches have been proposed to tackle the development of such critical systems, aiming at providing high assurance about their reliability, dependability, safety or security. These approaches include for example recommendations regarding the design (e.g. modularity through *Object-Oriented Programming*), the existence of appropriate documentation, the amount of testing, the setup of reviews by independent peers. The multiplication of practices and controls is expected to reduce as much as possible the likelihood of flaws in the final system.

Formal methods represent a very distinct approach in this domain: they aim at erradicating errors – at least of some sorts – on the basis of mathematically justified analyses. For this reason, their use is encouraged, or sometimes required, when dealing with critical systems requiring high assurance. Typical examples are provided by the higher Evaluation Assurance Levels (EALs) of the Common Criteria (*ISO/IEC 15408*) [CC] in the field of security, or the higher Safety Integrity Levels (SILs) of the Iec 61508 [IEC] in the field of safety and dependability.

There are numerous types of formal methods. A formal method is first characterised by the scope of problems that it can tackle, and its (justified) completeness – whether or not it detects *all* errors of a certain kind – and correctness – whether or not all reported problems are actual errors. Indeed, nearly any interesting problem that can be considered is associated to undecidability results, proving the absence of generic algorithms; therefore some formal methods aiming at automation work by approximation (leading to false positives, that is reporting a problem without actual error, or false negatives, that is undetected errors).

A formal method is also characterised by its automation and its complexity – that is the level of expertise required to use it. Indeed, when dealing with undecidable problems, the alternative to approximation is to rely on user guidance. The formal spectrum therefore goes from lightweight approaches such as typing systems, in which very specific classes of problems are automatically detected without any guidance of the user, to deductive formal methods, able to tackle a much more wider scope of aspects, but requiring the user to specify, implement, and prove – such interventions definitely justifying appropriate

training and possibly increasing development time.

1.2 Deductive formal methods and security

In this thesis, we provide a detailed analysis of the benefits resulting from the use of formal methods for development of high assurance sytems, but also of the possible limitations, problems and traps.

More specifically, we are concerned with the applications of the so-called deductive formal methods, that is logic-based approaches for the development of systems correct with regard to their specification, such as B, Coq or FoCaLize. Furthermore, we focus on the development of systems having strong security requirements – that is able to withstand attacks by intelligent agents – by contrast with safety or dependability requirements.

Indeed, whereas deductive formal methods aim at a form of perfection and can tackle a wide range of situations through very expressive languages (relying on extensive guidance by the user), there are various reasons for which "proven security" does not mean absolute and inconditional security in the physical world.

We identify these potential pitfalls by considering the whole process of formal developments, from the specification to the implementation through refinements. This analysis can be seen as a guide for independent evaluators – which are important actors of the security certification defined in the Common Criteria – by helping them to understand the true benefits of the use of formal methods and to identify some of the critical aspects that need specific checking.

In parallel, we also discuss possible improvements in the formal techniques or practices, alternative presentation of theoretical subtleties, and the risks induced by potential errors in the theories or the tools implementing formal methods. This leads us to propose an intuitive and generic theory of refinement, to validate the B method through a deep embedding in Coq, and to illustrate the development of mechanically checked tools with a prover for the B logic. We also present side results of these activities: new theorems for the B logic, and some techniques for the mechanisation of languages, including an improved *de Bruijn* representation.

1.3 Preliminary warning

It seems important to emphasise in this introduction that there is no doubt, for the author of this memoir, on the interests and benefits resulting of the application of formal methods.

They indeed provide a form of *silver bullet* for system engineering in general, and software engineering in particular: they have the potential to eradicate many forms of errors, a perspective not even dreamed of with other approaches. The impact on development costs are minor with regard to the gains in terms of confidence; furthermore the overall cost, taking into account not only the development but also the testing, the certification and the maintenance, is likely to be reduced when using formal methods [BDM98, WLBF09, Hal90].

But even considering formal methods as the silver bullet of system development, one has to realise that such a weapon can slay werewolves, but not the other strange creatures that may be lurking around. This is especially relevant when dealing with security, facing intelligent agents that can adapt their attacks when facing formal developments.

The genuine extent of the benefits of the application of formal methods has to be explicited and understood; the worst situation in security is not to use an unsecure system, but to use an unsecure system while assuming it is secure.

1.4 A remark about results and proofs

Readers of the memoir may be intrigued that whereas we discuss at length results and theorems, some of them being rather complex, we never provide any proof. Indeed, they are in fact formalised and proven in Coq, the corresponding developments being freely accessible and reusable (cross-references between this memoir and the developments are provided in Annex A).

We have several motivations for having adopted this approach. First, the use of a proof assistant checking proofs prevents oversights and errors. Second, by managing technical and administrative details, a proof assistant in fact leads to dare to tackle more complex results. Last but not least, we advocate the use of formal methods, and it seems to us that our message is much more likely to be considered if we apply our recommendations to ourselves. We expect this memoir to clearly illustrate the feasibility of complex developments using formal methods.

1.5 Plan of the document

After the introduction of a few conventions in Chapter 2, we provide in Chapter 3 a quick overview of formal methods in general, and of deductive formal methods in particular. It includes a more detailed presentation of B, Coq and FoCaLize, the three deductive formal methods which are used for later illustrations and discussions.

Chapter 4 discusses various notions about security, and the associated problems and practices. More specifically, we put the emphasis on those differences with safety considerations that are of interest with regard to the subject of this memoir. The threat model that we consider in later illustrations is also discussed, describing the objectives and abilities of an attacker.

In Chapter 5, we make a general review of the benefits, but also of the limitations and possible difficulties resulting of or associated to the application of formal methods to security developments. This review is organised according to the different steps encountered during a typical V-cycle development (a popular standard approach for project management, proposing a decomposition of development and acceptance activities); we discuss specification, design and implementation, as well as the tools that are part of the formal development environment.

Refinement, which is one of the key concepts discussed in the general review of Chapter 5, clearly deserves additional consideration. It is therefore analysed in Chapter 6, in which we propose a simple and generic vision of a sometimes elusive concept. Instead of providing illustrations of traps, we explore the associated underlying mechanisms, considering for example the so-called *Refinement Paradox*, as well as other items of interest.

Chapter 5 also discusses the problems that can result of oversights in the theory of a formal method, or of invalid implementations in formal tools. We focus on these questions in Chapter 7, which deals with the validation of a formal method. We discuss how it is possible to verify a formal method and to develop mechanically checked tools for this method. This is illustrated through an embedding of the B method in the Coq proof assistant. In the same chapter we also present new results for the B logic, proven using this embedding.

The development of such an embedding is a relatively complex and long process. The embedding of B in Coq is therefore further considered in Chapter 8, that addresses technical aspects. It discusses for example *de Bruijn* representations and some associated optimisation that we have designed, and present various proof strategies. Whereas these

technical elements are not directly relevant with the main guideline of this memoir, they are provided for the sake of completion and can have wider applications, for example with regard to language mechanisation or proof automation.

Chapter 9 finally concludes this memoir, and considers possible further activities.

Chapter 2

A Few Conventions

2.1 Styles

For the sake of readability, specific styles are used in this memoir to distinguish between mathematical concepts, names, pieces of code, etc.

Proper names are emphasised, as in *N.G. de Bruijn*. The same style is also used to mark the first introduction of important concepts, later appearances of the term being not emphasised, as well as important remarks.

Tools, languages, standards and organisations are denoted using small capital letters, as in FoCaLize.

Logical propositions, theorems, or other mathematical notations can either be included directly in the text of a paragraph, as in $x \in S$, or on the contrary be provided separately:

$$member(x, S) := x \in S$$

We also distinguish pieces of code using a dedicated style. Small snippets, for example identifiers, can be included directly in the text of a paragraph, as in `let x=y`, whereas longer pieces of code are provided separately, in a framed paragraph:

```
let x=y*3 in
 x*x;;
```

Note that we generally favour, for the sake of simplicity, the presentation of code as pieces of mathematical notations. In such a case we distinguish keywords by using a bold style, e.g. "**Variable** $test : \mathbb{N}$.". When using the mathematical presentation for code, we tolerate a few deviations from the genuine syntax (for example by omitting delimiters such as ";" or ".", or even keywords such as **end**); on the contrary when we use the dedicated code style, we stick to the concrete syntax, and these snippets can normally be interpreted or executed without modification.

2.2 Environments

Several LaTeX environments are defined and used in this document; their uses are marked by a category, a reference (chapter, section, number), and an optional title:

Example 2.2.1 (Environment) *This is an illustration of the example category.*

The different categories are as follows:

- Example: examples and illustrations;

- Definition: definition of terms, symbols and concepts;
- Notation: introduction of additional notations and symbols;
- Fact: granted results, for example theorems that are proven elsewhere but are just admitted in this memoir;
- Proposition: theorems that have been proved during the preparation of this memoir (the proofs are generally not provided in here, as it is explained in Section 1.4).

2.3 Notations

This document considers several formal methods and several logics – in some cases in the same example or theorem. We therefore need to deal with numerous concepts and notations. To the extent possible, we have tried to adopt standard notations, and to use them consistently throughout this document. In some cases however it has been considered preferable to reuse the same symbols with different meanings in different chapters, or on the contrary for a specific chapter to use a different notation.

Most of the notations are defined on the fly. They are introduced in the relevant chapter and section, just before using them. We just detail a few basic and common notations hereafter.

Standard notations are used for the logical operators, that is for example $\forall$ for the universal quantification, $\Rightarrow$ for the implication, $\neg$ for the negation. We also use standard notations for arithmetic operations such as $+$, set operations such as $\cup$, etc. *id* denotes the identity function.

Doubled capital letters, as in $\mathbb{S}$, denote sets and types. In particular, we use the following notations in this memoir:

- $\mathbb{N}$ is the set of natural numbers;
- $\mathbb{Z}$ is the set of integers;
- $\mathbb{Q}$ is the set of rational numbers;
- $\mathbb{W}$ is the set of boolean (binary) words;
- $\mathbb{B}$ is the set of boolean values, containing $\top$ (for true) and $\bot$ (for false);
- $\mathbb{U}$ is the set containing only the value **unit**;
- $\emptyset$ is the empty set.

We also consider the logical counterpart of $\mathbb{U}$ and $\emptyset$, that is **True** for the proposition that has a proof (that is a tautology) and **False** for the proposition that has no proof – not to be mistaken with the boolean values $\top$ and $\bot$.

$x{:}T$ means that x is of type T, $T_1 \rightarrow T_2$ is the type of functions whose parameter is in T_1 and the returned value is in T_2.

$x \mapsto y$ is the couple whose constituents are x at the first position and y at the second position. The symbol $\mapsto$ is also used for the definition of anonymous functions (such as in $\textbf{fun}\,(x{:}T) \mapsto x{+}1$), or in pattern matching constructs (e.g. $\textbf{match}\,b\,\textbf{with}\,\top \mapsto \bot \,|\, \bot \mapsto \top$).

As usual, the symbol = is associated to many concepts; to avoid confusions we have therefore adopted the following notations:

- $x{=}y$ is the equality predicate;
- $x \triangleq y$ means that x is defined by y;
- $x{:=}y$ means that x gets the value of y (that is the assignment);
- $[x \backslash T_1]T_2$ is the term obtained by replacing all the free occurrences of the variable x by the term T_1 in the term T_2.

Chapter 3

Deductive Formal Methods

The development of software, equipments or systems has been for long a source of engineering challenges. And whereas recent techniques and technologies offer opportunities to reduce costs or to address richer requirements, they also tend to increase the overall complexity of this task. It is therefore not a surprise that such developments often suffer from various forms of flaws, either in specification, design or implementation.

Various approaches have been proposed to develop better – or at least less flawed – systems, trying for example to improve the development process, to promote good practices, to guide design, or to ensure an appropriate level of testing. In this domain, formal approaches are characterised by strict and scientific visions with the objective of developing correct systems. To take the example of software, using formal methods, whereas programming is still an art, the verification is definitely promoted into a science. Formal methods often offer level of guarantees that are unreachable by more classical approaches; for example they can consider exhaustively the cases of use, whereas tests generally cover only a part of the possible executions.

Unfortunately, one has to recognise that the wide application of formal methods in industry is still an expectable prospect rather than a reality. Most of the industrial applications seem to be justified by certification objectives for high assurance systems, when applicable safety or security standards require formal methods applications. However, formal methods appear to be a mature subject, numerous tools exist, and there has been quite significant achievements on academic and industrial projects[1]: see for example [WLBF09] for a recent survey on the application of formal methods in industry, and for the latest significant developments in the domain of tools or environments the projects CompCert [Ler09], Sel-4 [KEH+09] or the Integrity operating systems [GHS].

In their most general definition formal approaches encompass the methods, techniques and tools using mathematically justified approaches for the analysis and the verification of specification, design, implementation and other related activities, including testing [WLBF09]. That is, they include for example rich type systems, static analyses, semantic interpretations, model checking, and so on.

Whereas any given formal method has a limited scope, either with regard to the phases of the lifecycle that it addresses or the type of verification that it provides, it comes with justified claims about the correctness or the completeness of its analyses. For example, a given static analysis may ensure an exhaustive identification of all buffer overflows in a program – but not of any other bug, and possibly to the cost of false positives; the important point is that the abilities and limitations of such an analyser are known and

[1] Additional results can be expected for example through the projects financed by the European Union, dedicated to the promotion of formal methods for industrial developments (e.g. Deploy and Quasimodo).

documented. The other typical criteria that can be considered to characterise a formal method, beyond expressivity, scope, completeness and correctness, are the existence of tools and their automation, the required expertise, and the accuracy, that is the difference between the considered system and its representation in the method.

We introduce in the rest of this chapter deductive formal methods, that is logic-based methods in which it is possible to reason by deduction on specifications and implementations, used in specification-driven developments. They are indeed the methods that are the most frequently encountered when dealing with secure systems[2] – probably because the security properties that one can try to capture are, to some extent, non-standard, and require expressive specification languages as well as advanced proof techniques.

This does not mean, however, that other formal approaches are useless or meaningless. In fact, as we point out later in this memoir, on the contrary we consider that the association of different types of methods is likely to be the most efficient approach to deal with security.

Note that whereas model-oriented methods (such as model checking) proceed by building a model satisfying a specification, deductive methods are based on inference rules operating at a nearly syntactical level to produce proofs. That is, whereas a model-oriented method identifies $P = \top$ and $Q = \top$ as the only possible model for the formula $P \wedge Q$, a deductive method apply a rule claiming that the proof of $P \wedge Q$ is build from a proof of P and a proof of Q.

We describe in the following the B method, the Coq proof assistant and the FoCaLize environment.

3.1 The B method

The B method, developed mainly by *J-R. Abrial*, is a popular formal method widely used by both the academic world and the industry. It has been applied, among other things, for industrial projects where safety or security is mandatory, such as transportation systems [BDM98, ED07], network devices [Bie96] or smartcards [Jaf07, SL00]. Several development environments for the B method exist, either industrial (e.g. the AtelierB or the B-Toolkit) or academic, such as the Brillant project [CPR+05].

The title of the so-called B-Book ([Abr96]), "Assigning Programs to Meanings", appears to be a reference to an earlier paper of *R. J. Floyd* [Flo67], titled "Assigning Meanings to Programs". Indeed, the B primary objective is to allow for the derivation of a correct imperative program from a specification, rather than to reverse-engineer existing code.

In a nutshell, the B method defines a logic, a language of specification and programmation, and a methodology of development based on the explicit concept of *refinement*.

The logic is a first-order predicate logic completed with elements of set theory. It is used to express preconditions, state invariants, etc. and to conduct proofs.

The language of specification and programmation is the Generalised Substitution Language (Gsl). It defines substitutions that can be abstract, declarative and non-deterministic (*i.e.* specifications) as well as concrete, imperative and deterministic (*i.e.* programs). Let us consider for example the following substitution:

Example 3.1.1 (Square root)

$$\textbf{ANY } x \textbf{ WHERE } x^2 \leq n < (x+1)^2 \textbf{ THEN } s := x$$

[2] At the date of the writing of this memoir, and as far as we consider certification requests in France (Anssi), all projects are based or use deductive formal methods, with a strong dominance of the B method.

It specifies the extraction of a square root (*i.e.* $s := \sqrt{n}$) using the non-deterministic substitution **ANY** (a magic operator finding a value which satisfies a property). This specification looks like an imperative program, but it is not; indeed there is no indication on how such a value can be computed. Note that the GSL is a language with side-effects; it includes an imperative executable sublanguage, name B0, whose translation e.g. into C or ADA is straightforward.

Regarding the methodology, B developments are made of *machines*, that is modules combining a state (in the form of variables), invariants on this state, and *operations* described as generalised substitutions to read or alter the state. A typical development starts with the description of abstract machines, that are pure specifications. The design and implementation then consist into providing, for any abstract machine, a succession of machines that have the same signature but are always more concrete, detailed and effective – until an executable code is produced.

The correctness is ensured by showing that any new machine is a refinement of the previous one; intuitively, a machine M_C refines a machine M_A if any observable behaviour of M_C is a possible behaviour of M_A. In other words, an implementation is compliant if it always stays within the limits defined by the specification. The verifications are incorporated into the methodology (and the tools) by the automated generation of proof obligations at each refinement step.

B refinement is defined in such a way that it is independent of the internal representation of the state of the machines, as illustrated hereafter by a system returning the maximum of a set of stored natural values. This example is very typical and is often referred to in the rest of this memoir under the name *Maximier*:

Example 3.1.2 (Maximier)

MACHINE M_A
VARIABLES S
INVARIANT $S \subseteq \mathbb{N}$
INITIALISATION $S := \emptyset$
OPERATIONS
$store(n) \triangleq$ **PRE** $n \in \mathbb{N}$ **THEN** $S := S \cup \{n\}$
$m \leftarrow get \triangleq$ **PRE** $S \neq \emptyset$ **THEN** $m :=$ **max**(S)

MACHINE M_C **REFINES** M_A
VARIABLES s
INVARIANT $s =$ **max**$(S \cup \{0\})$
INITIALISATION $s := 0$
OPERATIONS
$store(n) \triangleq$ **IF** $s < n$ **THEN** $s := n$
$m \leftarrow get \triangleq m := s$

The state of a machine is described in the **VARIABLES** clause, and the **INVARIANT** clause defines a constraint over this state; for M_A the state is described by a variable S which is a subset of $\mathbb{N}$, whereas for M_C it is a variable s which is a value of $\mathbb{N}$. The **INVARIANT** clause in M_C also describes the so-called *Glue Invariant* between the states of M_A and M_C. This is an important concept in B, describing the transformation of the representation of the state during refinement; here this invariant intuitively claims that if both machines are used in parallel then s is always equal to **max**(S).

The **INITIALISATION** clause sets the initial state and the **OPERATIONS** clause details

the operations used to read or alter it. The two machines differ yet M_C refines M_A: roughly speaking one cannot exhibit a behaviour of M_C which contradicts M_A.

Note the use of the **PRE** P **THEN** S substitution (also denoted $P|S$) defining a precondition, that can be seen as a requirement to enforce P before calling the operation (to avoid undefined, *i.e.* possibly undesirable, behaviours). This is an *offensive* approach; an operation (should not but) can be used when this condition is not satisfied, yet in such a case there is no guarantee about the result (it may even cause a crash). By opposition the *defensive* approach is represented in B by using *guards*, that is an **IF** P **THEN** S_1 **ELSE** S_2 substitution (also denoted $P \Longrightarrow S_1 [] \neg P \Longrightarrow S_2$) that prevent unauthorised uses. These notions are important and standard in formal methods; they will be discussed further later in this thesis.

More formally, a machine M_C refines a machine M_A if they have the same operations and that each operation of M_C refines the associated operation of M_A. This definition prevents changes of the representation of the inputs or outputs of operations (n and m in our example), which is fixed through the whole development; only the representation of the internal state can be modified during refinement.

B-refinement is based on semantics of specifications and programs as *predicate transformers* (cf. for example [HR84, Hoa92]), and refinements concepts similar to those introduced e.g. in [Dij76, Bac81, Bac88, Mor90, BvW00]. These are elegant approaches bridging the gap between imperative programs and stateless logic, yet they can be difficult to master for a developer.

B operations are substitutions, that is predicate transformers, and $[S]P$ denotes the predicate obtained by the application of the operation (substitution) S to the predicate P. The substitutions are then systematically defined by standard operations or logical equivalence, for example:

Definition 3.1.1 (Substitutions semantics)

$$[x := E]P \quad \Leftrightarrow \quad [x \backslash E]P \qquad \textit{Affectation}$$

$$[S_1 [] S_2]P \quad \Leftrightarrow \quad [S_1]P \wedge [S_2]P \qquad \textit{Choice}$$

$$[C|S]P \quad \Leftrightarrow \quad C \wedge [S]P \qquad \textit{Pre-condition}$$

$$[C \Longrightarrow S]P \quad \Leftrightarrow \quad C \Rightarrow [S]P \qquad \textit{Guard}$$

Remember that we have chosen our notations so that in the first line we distinguish the B substitution, which is a predicate transformer used to program, and the standard substitution. This line therefore expresses that the application of the B assignment $x := E$ to the predicate P is equivalent to the standard substitution of x by E in P – this justifies the notation $[S]P$, as well as the name of the GSL.

The intuitive interpretation of these semantics is that if $[S]P$ is provable, then the substitution S is such that its execution ensures that P is satisfied; for example, we have $[x := 1](x > 0) \Leftrightarrow 1 > 0$, that is executing the assignment $x := 1$ ensures that $x > 0$.

Provided these semantics, S_C refines S_A (denoted $S_A \sqsubseteq S_C$) iff:

Definition 3.1.2 (Refinement)

$$S_A \sqsubseteq S_C \quad \Leftrightarrow \quad [S_A]P \Rightarrow [S_C]P \textit{ for any predicate } P$$

A first-order definition is derived in [Abr96] from this definition, the intuition being that refinement weakens preconditions (if $P_1 \Rightarrow P_2$ then $P_1|S \sqsubseteq P_2|S$) and reduces non-determinism

$([S_1 [] S_2]P \sqsubseteq [S_1]P)$. The refinement can also be presented using the relational presentation introduced in the B-BOOK as follows:

Fact 3.1.1 (Refinement)

$$S_A \sqsubseteq S_C \quad \Leftrightarrow \quad \mathbf{pre}(S_A) \subseteq \mathbf{pre}(S_C) \wedge \mathbf{rel}(S_C) \subseteq \mathbf{rel}(S_A)$$

Intuitively, $\mathbf{pre}(S)$ represents the precondition of the substitution S and $\mathbf{rel}(S)$ the operational aspects of S as a relation that associates to any point of its domain one or several points of its co-domain.

3.2 The COQ proof assistant

COQ [Coq, BC04] is a proof assistant based on a type theory. It offers a higher-order logical framework that allows for the construction and verification of proofs, as well as the development and analysis of functional programs in a ML-like functional language with pattern-matching.

It is possible in COQ to define values and types; any value has a type, and any type is a value. Types of sort **Set** represent sets of computational values, while types of sort **Prop** represent logical propositions. For example, the natural number 0 is of type $\mathbb{N}$, whose type is **Set**, whereas a proposition such as $0<1$ is a type whose type is **Prop**.

COQ is based on the Calculus of Inductive Constructions [CP88, Wer94] and provides powerful inductive features. Let us first illustrate this with the definition of natural numbers:

Example 3.2.1 (Inductive definition of a datatype)

$$\mathbf{Inductive}\ \mathbb{N}:\mathbf{Set} \triangleq 0:\mathbb{N} \mid S:\mathbb{N}\rightarrow\mathbb{N}$$

It defines $\mathbb{N}$ as the smallest set of terms stable by application of the constructors 0 and S. $\mathbb{N}$ is exactly made of the terms 0 and $S(\ldots S(0)\ldots)$ for any finite iteration. Being well-founded, induction is possible, and the associated structural induction principles are automatically provided by COQ at the definition of this type, such as for example:

Fact 3.2.1 (Structural induction principle for $\mathbb{N}$)

$$\forall\,(P:\mathbb{N}\rightarrow\mathbf{Prop}),\ P\,0 \rightarrow (\forall\,(n:\mathbb{N}),\ P\,n\rightarrow P\,(S\,n)) \rightarrow \forall\,(n:\mathbb{N}),\ P\,n$$

That is, for a predicate P over $\mathbb{N}$, if one can prove $P(0)$ and that when $P(n)$ then $P(S(n))$, then P is true (provable) for any n.

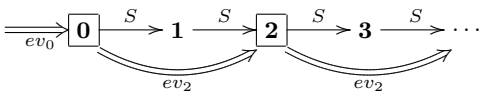
But it is also possible to inductively define propositions, such as the *even* predicate:

Example 3.2.2 (Inductive definition of a predicate)

$$\mathbf{Inductive}\ \mathrm{even}:\mathbb{N}\rightarrow\mathbf{Prop} \triangleq ev_0:\mathrm{even}\,0 \mid ev_2:\forall(n:\mathbb{N}),\ \mathrm{even}\,n\rightarrow\mathrm{even}\,S(S\,n)$$

It defines a family of logical types: *even* 0, *even* 1 and *even* 2 are different types. It is worthwhile to note that reading this definition, ev_0 is a value of type *even* 0 and $(ev_2\,0\,ev_0)$ a value of type *even* 2, while there is no way to build a value of type *even* 1, that is this type is empty. The standard interpretation of this observation is that ev_0 is a proof of *even* 0, $(ev_2\ 0\ ev_0)$ a proof of *even* 2, and that there is no proof of *even* 1, that is we have $\neg even\ 1$.

The intuitive view of these examples is that $\mathbb{N}$ is a set of terms, and *even* a predicate marking some of them, defining a subset of $\mathbb{N}$; its inductive definition describes paths in $\mathbb{N}$, starting from 0, along which the predicate is true:



The Coq logic is by default a constructive logic (a.k.a. intuitionnistic logic) in which the excluded middle is not provable. For example, the standard inductive definition of the disjunction is such that to prove $P \vee Q$ one has either to provide a proof of P or a proof of Q; therefore indeed $P \vee \neg P$ is not always provable, as this would mean being always capable, for any proposition P, to prove either P or $\neg P$. An interesting consequence of the constructive vision is that programs can be extracted from proofs (*Curry-Howard* isomorphism). In essence, a proof of the proposition $\forall (m\,n{:}\mathbb{N}),\, m \leq n \vee m > n$ can be transformed automatically into a functional program that, given two values m and n in $\mathbb{N}$, returns a boolean indicating whether $m \leq n$ or $m > n$.

This approach, extracting programs from proof, is representative of the so-called strong specification style, illustrated here by the division by 2:

Example 3.2.3 (Strong specification style)

```
Require Import Arith.

Theorem div2:forall (n:nat), {m:nat | n=m+m \/ n=S(m+m)}.
Proof.
 induction n as [| n Hind].
  exists 0; left; apply refl_equal.
  destruct Hind as [m Hind].
   destruct (eq_nat_dec n (m+m)) as [Heq | Hdiff].
    exists m; rewrite Heq; right; apply refl_equal.
    assert (Heq:n=S(m+m)).
     destruct Hind as [Hind | Hind].
      destruct (Hdiff Hind).
      apply Hind.
    exists (S m); rewrite Heq; left; simpl;
     rewrite <- plus_n_Sm; apply refl_equal.
Defined.
```

In essence, *div2* is presented as a theorem that claims that for any value n in $\mathbb{N}$ there exists (at least) one value m that is the half of n. In the constructive logic of Coq however, the only way to prove the existence of such a value is to exhibit it – and the proof is therefore in fact a procedure, a program to build a value, and a few checks that indeed this value satisfies the required property. It is then possible to use the command *Extraction div2*, that automatically produces the following code:

Example 3.2.4 (Code extraction)

```
let rec div2 = function
  | O -> O
  | S n0 ->
      let hind = div2 n0 in
      (match eq_nat_dec n0 (plus hind hind) with
         | Left -> hind
         | Right -> S hind)
```

The structure of this code is intimately related to the structure of the Coq proof, and appear to be rather exotic; even using the same algorithm, a developer is much more likely to propose something like the following code:

$$\textbf{let rec } div2(x:\mathbb{N}):\mathbb{N} := \textbf{match } x \textbf{ with } S(S(x')) \rightarrow S(div2(x')) \mid \_ \rightarrow 0 \textbf{ end}.$$

There is however no enforced development methodology in Coq; the user can choose between several styles of specification and implementation, and has to decide on its own about the properties to be checked. In addition to the strong specification style described just before, it is for example possible to use the weak specification style. It consists into defining functions as programs in the internal *ML*-like language and to later check properties of these functions, as illustrated here by the multiplication by 2:

Example 3.2.5 (Weak specification style)

```
Fixpoint double(n:nat){struct n}:nat :=
 match n with
 | O => O
 | S n' => S(S(double n'))
 end.

Theorem double_correct:forall (n:nat), double n=n+n.
Proof.
 induction n as [| n]; simpl.
  apply refl_equal.
  rewrite IHn; rewrite <- plus_n_Sm; apply refl_equal.
Qed.
```

One should note that the availability of inductive constructions can, to some extent, leads to use implementations as specifications: see the previous definition of $\mathbb{N}$ which is an implementation that can be completed e.g. with a program representing *Peano*'s addition. Yet, to have efficient computations, the binary representation $\mathbb{W}$ can be considered. Programs on $\mathbb{W}$ can be validated against the $\mathbb{N}$ library using an interpretation $[\![\,]\!]:\mathbb{W}\rightarrow\mathbb{N}$, to prove for example:

$$\forall\,(w_1\ w_2:\mathbb{W}),\ [\![plus\ w_1\ w_2]\!] = [\![w_1]\!] + [\![w_2]\!]$$

that is an homomorphism property. We consider such approaches, in which various concrete representations are considered, as a form of refinement that can be encountered in Coq, using implementations as specifications and changing data representations (cf. [Mag03a, Mag03b] for further illustrations).

3.3 The FoCaLize environment

FoCaLize [Foc] is a formal method to specify and incrementally progress towards implementations, while proving compliance.

Promoting a form of object-oriented approach, the brick development in FoCaLize is the species, which is a kind of record grouping declarations (types), definitions (ML pure functions), properties, proofs, and the representation (a datatype) of the data manipulated in the species. A species can be built and enriched by parameterisation, multiple inheritance and new declarations, late binding, definitions or redefinition, properties or proofs.

The representation can be declared (being then a type variable) and progressively defined along the development process until it becomes a concrete datatype. Once all declarations are defined and all properties are proved, a species is complete (it is an implementation) and can give rise to a collection, a sort of *Abstract Datatype* obtained by abstracting its representation and masking definitions, whose values can only be manipulated through the provided interface – ensuring the preservation of the proven properties.

Note that late binding allows for redefinition of previously defined methods, but not of the representation: once the concrete support is detailed, it is maintained through the development cycle down to the collections.

The parameters of a species are either collections, whose interface can be used in the species, or values of a parameter collection. In the following example, the species Ω specifies any collection which has a decidable equivalence relation (denoted = as we intuitively expect this relation to represent equality) on its elements:

Example 3.3.1 (Superset)

$$
\begin{array}{l}
\textbf{species}\ \Omega \triangleq \\
\quad \textbf{signature}\ (=)\ :\ \textbf{Self} \to \textbf{Self} \to \mathbb{B} \\
\quad \textbf{property}\ =_{\text{refl}} : \forall\, s : \textbf{Self},\ s = s \\
\quad \textbf{property}\ =_{\text{symm}} : \forall\, s_1, s_2 : \textbf{Self},\ s_1 = s_2 \Rightarrow s_2 = s_1 \\
\quad \textbf{property}\ =_{\text{tran}} : \forall\, s_1, s_2, s_2 : \textbf{Self},\ s_1 = s_2 \Rightarrow s_2 = s_3 \Rightarrow s_1 = s_3
\end{array}
$$

We can then define a species $\wp$ specifying any collection implementing subsets of values from its parameter ω, which represents a collection implementing Ω:

Example 3.3.2 (Powerset)

$$
\begin{array}{l}
\textbf{species}\ \wp(\omega\ \textbf{is}\ \Omega) \triangleq \\
\quad \textbf{signature}\ \emptyset\ :\ \textbf{Self} \\
\quad \textbf{signature}\ (\in)\ :\ \omega \to \textbf{Self} \to \mathbb{B} \\
\quad \textbf{signature}\ (+)\ :\ \omega \to \textbf{Self} \to \textbf{Self} \\
\quad \textbf{property}\ \in_{\emptyset} : \forall\, s : \omega,\ \neg s \in \emptyset \\
\quad \textbf{property}\ \in_{+} : \forall\, s_1, s_2 : \omega, \forall\, S : \textbf{Self},\ s_1 \in (S + s_2) \Leftrightarrow s_1 \in S \vee s_1 = s_2
\end{array}
$$

Note that in the species $\wp$, the symbol = appearing in the property $\in_+$ is the one introduced in the species Ω, that is $x = y$ is more explicitly denoted $\Omega!(=)(x, y)$.

In essence, $\wp(\omega)$ represents any implementation of subsets of values of ω having a decidable membership $\in$. That is, it can be implemented for example by lists of values of ω – only allowing the representation of finite sets – but also by functions in $\omega \to \mathbb{B}$ to represent possibly infinite sets.

Using the latter, there would be no way to implement inclusion provided ω is infinite. To specify extensionally comparable subsets, that is subsets with an inclusion (and therefore an equality) we can define a further species $\mathcal{F}$ inheriting from $\wp$ – as we want to represent subsets – but also of Ω to reuse its specification of equality. A property $=_{ext}$ is also added to further specify this equivalence and ensure that it represents the extensional equality:

Example 3.3.3 (Extensional subsets)

$$
\begin{array}{l}
\textbf{species}\ \mathcal{F}(\omega\ \textit{is}\ \Omega) \triangleq \\
\quad \textbf{inherit}\ \Omega, \wp(\omega) \\
\quad \textbf{signature}\ (\subseteq)\ :\ \textbf{Self} \to \textbf{Self} \to \mathbb{B} \\
\quad \textbf{property}\ \subseteq_{\text{spec}} : \forall\, S_1, S_2 : \textbf{Self},\ S_1 \subseteq S_2 \Leftrightarrow (\forall\, s : \omega,\ s \in S_1 \Rightarrow s \in S_2) \\
\quad \textbf{property}\ =_{\text{ext}} : \forall\, S_1, S_2 : \textbf{Self},\ S_1 = S_2 \Leftrightarrow S_1 \subseteq S_2 \wedge S_2 \subseteq S_1
\end{array}
$$

Note that another consequence of having $\mathcal{F}$ inheriting of Ω is that provided an implementation of Ω, for example *int* for the machine integers, and an implementation *lstset* of $\mathcal{F}$ for example using sorted lists, it is possible to immediatly derive an implementation $lstset(int)$, to manipulate subsets of machine integers, but also an implementation $lstset(lstset(int))$, to manipulate subsets of subsets of machine integers, and so on.

Note that in FoCaLize, inheritance encompasses two concepts. The first one is a composition of specifications to describe specific entities through a richer interface (more operations or more properties), and is illustrated by our examples of $\wp$ and $\mathcal{F}$. The second concept maintains the same interface but provides more details about the effective implementation, for example implementing $\mathcal{F}$ by sorted lists. The latter is a form of refinement whose validity is ensured by type-checking, dependency analysis and verification that all the properties have been proved.

Chapter 4

Information System Security

As mentioned in the previous chapter, this memoir deals with critical systems, for which errors can have inacceptable consequences, and with the application of formal methods to ensure their validity, in particular with regard to security objectives.

This chapter aims at providing some notions about security. It does not intend to cover the subject exhaustively, of course, but just discusses some relevant aspects, such as typical security requirements, or threat descriptions. It emphasises, where required, the differences between the security perspective and the more usual safety (or dependability) perspective. Indeed, one should not confuse safety with security: the former mostly aims at preventing or limiting the consequences of random events, using probabilities as a relevant measure; on the contrary, security concerns are related to malicious actions by intelligent agents, and it is generally much more appropriate to consider a form of cost to evaluate the likelihood of an attack.

The distinction is important, especially as these two fields may use a common vocabulary, but with different definitions; we will for example discuss about the notions of integrity and availability. Similarly, security and safety can deal with non functional requirements, and share some techniques, such as various form of redundancy, but with different – or even contradictory – objectives. It is important to realize the difference between these perspectives, as trying to tackle security using safety reasonnings is error prone.

4.1 Security requirements and practices

Security requirements traditionnally fall into three categories, *confidentiality*, *integrity*, and *availability*. This classification, introduced in by *R. Courtney* (IBM) in the early 70's, is still in use today even if it is generally considered as a structure rather than an exhaustive list; it is often enriched with other notions, such as *authentication* or *accountability*.

There are several definitions associated to these terms – the recognised standards in security do not provide a commonly agreed definition. They can be applied to information, data, processes, services, or other objects; for the sake of clarity, we just discuss in this chapter the application of these notions to data. It is however essential, of course, to understand what we want to protect, and against what type of threat.

The confidentiality is more specifically related to security. It describes the fact that a datum should not be known – in whole or in part – by unauthorised entities.

Integrity, on the contrary, is a characteristic that can be required in safety as well as in security. In both cases, the associated objectives are related to changes of a data, either to ensure that they are impossible (immuability), or that they can be repaired, or more

simply that they can be detected. However, safety integrity and security integrity are very different.

When dealing with safety considerations, integrity can for example be ensured by error detection and correction codes or other similar techniques: a datum D is encoded as $Enc(D)$, producing a new representation with redundant information; it is then possible to evaluate the probability that a random change to $Enc(D)$ occurs but is neither corrected nor detected. Of course, this type of approach is inappropriate when dealing with security: an attacker is assumed to know the encoding scheme *Enc*, and is therefore able to alter the data without being detected.

Availability is another frequent source of confusion between safety and security. In both cases, the objective is to ensure that a datum is accessible when required. But whereas in safety this is associated for example to requirements about MTBF (Mean Time Between Failures) or priority management, in security it is also often a problem of survivability. For a network device, for example, security availability can be translated into the fact that the equipment should not crash when fuzzed.

More generally, the security requirements for a system can be described in a so-called *Security Policy*. It will for example describe, for a given entity and a given state, the set of authorised actions. They can provide a more dynamic vision of security objectives, for example the fact that a given datum is accessible by a given entity in some states of the system, and not accessible in the other states.

To enforce security requirements, different techniques can be applied. A very specific family of techniques for security is related to the use of cryptographic schemes. The confidentiality of the data can be ensured by ciphering, the integrity – in the sense of detecting unauthorised modifications – by signature. Interestingly, in general cryptographic schemes transpose security requirements to the keys; for example the confidentiality of the ciphered data is only ensured provided the key is kept secret, but also unmodified: if the attacker is not able to read the value of the key, he may more simply modify it to set it to a value chosen by him.

Another frequent technique is the use of security monitors enforcing security policy, for example access control. According to the ORANGE BOOK [TCS], a security monitor is a tamperproof, unavoidable, and "simple enough to be trusted" mechanism filtering accesses, and is for example realised by appropriate mechanisms checking rights of users before executing their requests in operating systems.

In both safety and security, it is also a common practice to ensure *defence in depth*. The idea is to have several layers of mechanisms to enforce expected properties. In the case of safety, if for example the objective is to ensure resistance to random failures, a simple redundance of safety mechanisms can ensure defence in depth. If the feared problem is a design flaw, then a duplication will not be sufficient, either for safety or security. A possible approach of the defence in depth concept in security is to ensure that more than one of the following axes is addressed for any security objective:

- Prevention: there is no weakness;

- Protection: weaknesses are not exploitable;

- Detection: attacks are detected;

- Limitation: consequences of attacks are limited;

- Reparation: consequences of attacks can be overcomed.

For example, considering stack overflow attacks leading to arbitrary code execution, a formal design proven to forbid overflows is a prevention, the marking of parts of the memory as non executable is a protection, protecting the integrity of the stack with canaries[1] ensures detection, limited privileges (a user process cannot modify the operating system) is a limitation, and backups guarantee reparation.

As a last remark, we mention in the introduction of this chapter the existence of *non functional* requirements for safety or security. In safety, this can for example encompass objectives about the maximum amount of memory required by a program; it is non functional in the sense that it does not describe a constraint over the inputs and outputs of the program seen as a black box, but on what's going on inside this black box.

Non functional requirements also exist in security. A typical example is the secure erasing of a secret value, such as a session key stored only – expectingly – in the memory. A standard practice is to ensure secure erasing by repeatedly writing different values, and its application can have rather unexpected effects, such as for example:

- A compiler, discovering that the memory location is not read during or after secure erasing, can optimize the program by removing the useless write operations;

- A Flash memory, due to specific technological constraints, if required to make multiple writings at the same address, will distribute these writings at different physical locations, preventing effective overwriting.

These deviations are direct consequences of the non functional nature of secure erasing: it has no influence on the result of the execution of the program, and it is therefore considered "irrelevant" in many situations. And the point is, standard formal methods typically only deal with functional requirements. In the rest of this memoir, functional requirements are said to be *extensional*, and of non functional requirements *intensional*.

4.2 Characterising the threat

The safety analysis for a system is always conducted according to a model of the random events that can have disastrous consequences – adverse weather conditions, failure of components, and so on.

Similarly, in a security analysis, a model of the threat is required. Having identified for a system what we want to protect, against which type of threats, we have to assess the possible attackers, to eventually develop appropriate protection mechanisms. As mentioned, probabilities are generally inadequate for security analyses; an evaluation of the cost of an attack is preferable. Such a cost can include several types of considerations, such as the required expertise, the required time, computing power, network bandwidth or other resources, the location of the attack (in space and in time) but also the risk for the attack to be detected or the attacker to be identified, as well as the consequences of the attack, both in terms of gain for the attacker or in term of losses for the defendant.

In this memoir, we often consider powerful attackers in our illustrations; they have, after all, to attack formally proven systems. One of the worst case situations is the malicious developer. Indeed, whereas in safety developers are trusted (but can be inattentive), in security the developer – or at least a person able to influence the implementation in a team – can be an attacker. Its objective is then to implement inappropriate features in the system, while getting a formal specification. This type of attacker is not always with

[1] A canary is a randomly chosen value placed on the stack and checked regularly to detect unauthorised changes, e.g. of the return pointer.

a very high profile, only applicable at governmental level; consider for example a payment application in a company, and the potential benefits for a developer able to trap it.

We will generally not need further hypotheses regarding our attackers. In particular, we do not need for our illustration to consider social engineering, important resources (such as powerful computing power for cryptanalyses), and so on.

4.3 Security certification

Critical systems need high levels of assurance, that can be obtained through a certification process accordingly to a procedure set forth in an applicable standard. In the field of security, the Common Criteria [CC] are the current reference; inheriting from the United States TCSec and the European ITSec, they are used worldwide, and mutual recognition of certificates is ensured between nations by multinational agreements. The full details of the Common Criteria certification process are not discussed here, we just mention a few aspects which are relevant for our concerns.

When a developer decides to start a Common Criteria certification process for a product, the first step is to describe the associated security objectives in the so-called *Security Target*. This document includes a description of the assets that have to be protected, and the type of threats that are considered applicable, as well as the chosen Eal (Evaluation Assurance Level), between 1 (basic) and 7 – which basically describes the level of effort dedicated to ensuring that the system is indeed secure.

The certificate itself is delivered by a national authority, that is for France the Anssi (the French network and information security agency), on the basis of a report provided by independent evaluators working for an agreed security laboratory. The mission of these experts is to ensure that the developer applies the procedures associated to the chosen Eal (development practices, documentation, testing, etc.), and to conduct further evaluations, such as vulnerability analyses.

Currently, the assurance components for a given level are systematically included in the higher levels. That means for example that the use of formal methods, mandatory at the highest levels[2], does not reduce the amount of testing required. One of the motivation of the work described in the next chapter is to assess whether the use of formal methods could indeed justify the relaxation of other assurance components, possibly provided additional verifications by the independent evaluators.

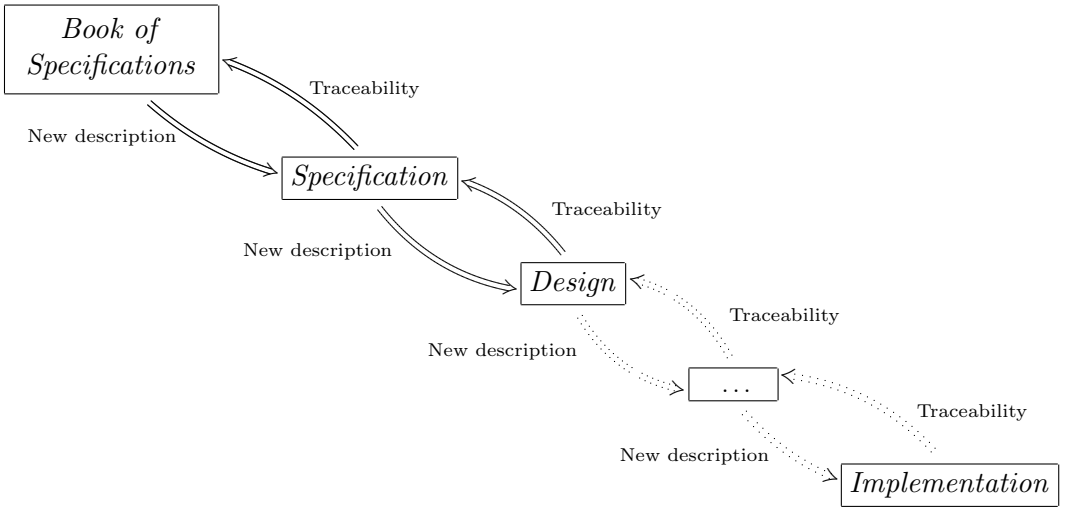
[2]For the version 2.3 of the Common Criteria, a formal model of the security policy is required at Eal-5, and a full formal development is required at Eal-7. For the version 3.1 of the Common Criteria, formal methods are mandatory only for Eal-6 and Eal-7.

Chapter 5

Using Deductive Formal Methods for Secure Developments

As pointed out, the use of formal methods is encouraged, when not required, by standards for the development of systems in which safety is mandatory, e.g. *IEC 61508* [IEC]. The situation is similar for the development of secure systems: for the highest levels of assurance the Common Criteria [CC] for example require the use of formal methods to improve confidence in the development, as well as to ease the independent evaluation process. Indeed, the verification that the delivered product complies with its specification is expected to rely, at least to some extent, on the use of an automated tool, for example mechanically checking a proof of correctness provided by the developer.

Note that safety and security standards generally enforce the typical V-cycle vision; we refer to such type of developments, in this memoir, as being *specification-driven*[1]. The V-cycle identifies several phases such as specification, design, implementation and verification operations. Different languages can be used to describe the system in the different phases; beyond programming languages it is frequent to use natural language, automata, graphical languages, Uml, etc. In a specification-driven development, the problem of the correctness can then be seen as a problem of *traceability* between the various *descriptions* of the system produced at the different phases of its lifecycle.



The deductive formal methods also allow for multiple descriptions of a system; they differ from standard approaches by enforcing the use of languages with explicit and clear

[1] By contrast with other methodologies, for example Test Driven Development.

semantics, and by providing a logical framework to reason on them. Ensuring the correctness then becomes a mathematical analysis of the traceability (or consistency) between these different descriptions.

The intent of this chapter is to show that whereas the use of deductive formal methods for specification-driven development of secure systems is perfectly relevant, and provide a priceless mechanical support to tackle complex problems, specifying a secure system or deriving a secure implementation can still be rather tricky. We analyse the various steps of a development, considering for each of them possible pitfalls, limitations, and problems. These concerns are illustrated through simple examples (sometimes involving a malicious developer) in Coq, B or FoCaLize but most of them are relevant for other deductive formal methods such as Pvs, Isabelle, etc. More generally, some of the problems described here can be applicable to other types of formal methods (e.g. Model Checking) or to developments with safety objectives.

5.1 Preliminary considerations

5.1.1 Formal specification

At least two descriptions of a system are generally considered in formal developments, a *formal specification* and an *implementation*. The specification is often written in a logical language (e.g. based on predicates) and is ideally declarative, abstract, high-level and possibly non-deterministic, describing the *what*. On the other hand, the implementation is imperative, concrete, low-level, and deterministic, describing the *how*.

To emphasise the difference between declarative and imperative approaches, consider the specification of the integer square root function provided in Section 3.1:

Example 5.1.1 (Square root specification in B)

$$\textbf{ANY}\ x\ \textbf{WHERE}\ x^2 \leq n < (x+1)^2\ \textbf{THEN}\ s := x$$

It is fully deterministic (for any n there is at most one acceptable value for s), but it is not a program: how the relevant value is computed is left to the developer. The specification can be seen as an oracle to check whether or not an implementation is correct – this is one of the ideas applied in test tools using formal specifications.

The simple fact of writing a formal specification is already an improvement compared to standard approaches. Indeed, by using a formal language, ambiguities are resolved, and to some extent some forms of common mistakes can be identified. Furthermore, formal methods allow for the assessment, at least partial, of the consistency – that is the absence of internal contradictions – or the completeness of a specification. For example, it is possible to check that a set of basic properties expressed in a specification indeed allows for the derivation of the expected high-level properties – before even starting design or implementation.

5.1.2 Refinement

The term *refinement* is often encountered when dealing with formal methods to describe the underlying development process during the design and the implementation of a system. Numerous formal or semi-formal methods provide a specific definition of this concept (cf. [GFL05] for a few examples). As far as deductive formal methods are concerned, it is more specifically associated to the formalisation of traceability using for example predicate transformers semantics [HR84, Hoa92], such as in Z, B or Event-B.

In this memoir, however, we generalize the concept of refinement to apply it to any specification-driven development using a deductive formal method. Refinement is therefore the process of going from a formal specification to an implementation, while checking the compliance of the latter to the former. It captures the engineering of a system and encompasses a lot of subtle activities, including for example in the case of software development the choice of concrete representations for abstract data or the production of operational algorithms satisfying declarative descriptions. From a logical point of view, a formal specification describes a family of *models* (that is, intuitively, implementations) and the refinement process consists in choosing progressively one of those models.

Note that one has to assume that the formal specification is *sufficient* with regard to the intent of its developer, that is the properties of the final implementation are either part (or consequences) of the formal specification, or on the contrary are incidental and irrelevant, in the sense that they do not represent a genuine requirement. Let us illustrate our point with the specification of a very simple library for manipulating sets: it introduces and describes the empty set $\emptyset$, singletons $\{x\}$, membership $\in$ and union $\cup$ as well as their relationship, that is:

Example 5.1.2 (Specification of set operations)

$$\neg x \in \emptyset \qquad x \in \{y\} \Leftrightarrow x = y \qquad x \in S_1 \cup S_2 \Leftrightarrow x \in S_1 \vee x \in S_2$$

This specification is sufficient, and implemented for example using lists. Such implementations *de facto* introduce additional properties, for example they define an order of appearance of the elements of a set in the list implementing it, an order which can become observable for example using a pretty-printing function. Yet these additional properties do not result from the requirements, but are artificial and irrelevant. This is typical of refinement. On the contrary if a new description is introduced to provide additional requirements, then we are still in a specification stage; in such a case this is not a refinement but a form of composition of specification. In both cases the number of acceptable models – in the logical sense, that is the number of acceptable implementations – decreases, but the intentions fundamentally differ.

Note that formal methods do not automatically produce refinements[2] but explain how to check that a refinement is valid, that is they ensure that very different objects (a logical description and an operational implementation) are sufficiently "similar".

5.1.3 About logic

Behind any deductive formal method, there is logic – or more accurately, a logic. We do not want here to discuss at length the various types of logic, once pointed out the common fact that a specification can be inconsistent.

A specification is *inconsistent* if it is self-contradictory – a trivial example is to specify v as a natural value equal to both 0 and 1 at the same time. Such a specification is also said to be unsatisfiable, that is it does not admit a model in the logical sense. There are three points that are worth mentioning about inconsistent specifications:

- the detection of inconsistency cannot be automated in the general case (the problem of satisfiability is undecidable for the predicate calculus);

- an inconsistent specification cannot be implemented;

[2]Automated refinement is however a field of study and can for example select simple and standard models, e.g. according to design patterns [Req08].

- an inconsistent specification can be used to prove any property about any compliant implementation – and more generally to prove anything, as soon as the existence of a compliant implementation is assumed, implicitly or not.

Because of the first point, tools implementing formal methods considered in this memoir do not even try to detect inconsistencies, even in trivial cases such as $v = 0 \wedge v = 1$. On the other hand, if there is an implementation objective, because of the second point this inconsistency will be detected sooner or later. The last point results from the fact that for any proposition P we have $\mathbf{False} \Rightarrow P$ (using false assumptions one can prove anything). The consequences of these points are discussed later.

5.2 Specifying secure systems

The first point to note, when considering the potential gains resulting of the application of deductive formal methods to secure developments, is that whereas they offer tools to express specifications, there is no way to force a developer to describe the required properties. Clearly, using even the most efficient formal method without adopting the "formal spirit" is meaningless, as there is no benefit compared to standard approaches if the formal specification is empty, that is trivially valid for any system.

Note also that a formal development is a development, and can therefore benefit from standard practices such as naming conventions, modularity, documentation, etc. In the case of formal methods, in fact, the very process of deriving a formal specification from the book of specifications should be documented, justifying the formalisation choices and identifying, if any, aspects of the system left out – as it is generally not reasonable or even feasible to aim at a full formalisation of a complete system.

Assuming a developer that has adopted the formal spirit, there are further points to care about in order to develop an adequate formal specification for a secure system, that is a specification not only expressing the required properties, but also ensuring that those properties are enforced at all stages of the development as well as in any (reasonable) use case of the implementation.

Some of the concerns discussed thereafter are applicable for safety or any high assurance system; others are more specific to security as the intervention of a malicious developer is assumed. The ultimate objective of such a malicious developer is to exploit any weakness of a specification, in order to trap a system while delivering a mechanically checked proof of compliance. One could consider that such traps would be detected through code review or testing. Yet, beyond the fact that formal methods are expected to reduce the need for such activities, we warn the reader that our illustrations are voluntarily simplistic, and that real life examples of Trojan Horse can be much more difficult to detect.

5.2.1 Inconsistent specifications

As pointed out in Section 5.1.3, inconsistent specifications are disastrous. Indeed, whereas inconsistency cannot be automatically detected, it also permits to discharge any proof obligation expressed – that is an inconsistent specification can in practice make the developer life more comfortable. An inconsistent specification is therefore dangerous for safety developments if a distracted developer fails to notice that its proofs are a little too easy to produce, and more so for security developments as a malicious developer identifying such a flaw would be able to prove whatever he wants.

Of course, an inconsistent specification is not implementable. It is therefore possible to check the consistency by providing an implementation – any one will do the trick, so

even a dummy implementation is sufficient. Yet there are in security situations in which a formal specification is mandatory while a formal implementation is not. This is the case for the COMMON CRITERIA, that at some assurance levels (e.g. EAL-5 for COMMON CRITERIA v2.3.) just require a formal specification of the Security Policy. An undetected inconsistent specification is therefore a possibility.

In B for example the consistency of a specification is partially checked through proof obligations that have to be discharged by the developer. In essence, operations have to be specified in such a way that the establishment and the later preservation of the invariant is indeed guaranteed. Yet the obligations related to the existence of values satisfying the expressed constraints for parameters, variables and constants are deferred. Both following specifications are inconsistent, yet all the explicit proofs obligations can be discharged (that is, most B tools will report for such a development a 100% proven status):

Example 5.2.1 (Insatisfiable specification)

MACHINE absurd_var
VARIABLES v
INVARIANT $v \in \mathbb{N} \wedge v = 0 \wedge v = 1$
ASSERTION $0 = 1$

MACHINE absurd_cst
CONSTANTS f
PROPERTIES $f \in \mathbb{N} \rightarrow \mathbb{N} \wedge \forall\, x, y,\ x < y \Rightarrow f(x) > f(y)$
ASSERTION $0 = 1$

Delaying such proof obligations is justified, as implementing the specification will force the developer to exhibit a witness for v that meets the specification (a constructive proof that the specification is satisfiable). Therefore, B ensures that any inconsistency is detected, at the latest, at the implementation stage. But as a formally derived implementation is not always required, one should consider additional manual verifications to check the existence of valid values for parameters, constants and variables.

Inconsistencies can be rather easy to introduce, accidentally or not, by contradicting implicit hypotheses associated to the used formal method. In the current version of FOCALIZE, the termination of recursive functions is assumed. One should care to check that all the defined functions indeed terminate to avoid to introduce an inconsistency. In B the clause **SETS** allows for the declaration of abstract sets used in a machine; one can easily forget that such a set, in B, is always finite and non-empty. If the developer contradicts one of these implicit hypotheses the specification becomes inconsistent without any warning by the tool; in fact the automated prover is likely to very efficiently detect the contradiction as a lemma usable to discharge any proof obligation.

Contradicting implicit principles of the underlying logic can also be illustrated in COQ with two very simple examples.

Example 5.2.2 (Violation of properties of inductive types) *We attempt to define* $\mathbb{Z}$ *from the already defined* $\mathbb{N}$*, distinguishing between positive and negative values; we also ensure that* $+0 = -0$ *with an axiom:*

Inductive $\mathbb{Z} : \mathbf{Set} \triangleq \mathrm{plus} : \mathbb{N} \rightarrow \mathbb{Z} \mid \mathrm{minus} : \mathbb{N} \rightarrow \mathbb{Z}.$

Hypothesis $\mathrm{zero\_unsigned} : \mathrm{plus}(0) = \mathrm{minus}(0).$

Whereas this specification is unlikely to shock readers not accustomated to inductive constructions, it is not valid. Indeed, $\mathbb{Z}$ *is not described here as an abstract specification but as an implementation: it is the set of all terms of the form* $\mathrm{plus}(n)$ *or* $\mathrm{minus}(n)$*, and it is implemented as such for example in* OCaml. zero_unsigned *introduces an inconsistency because it contradicts the injectivity of constructors – or more precisely the principle of non confusion – that claims here that for any natural values* n *and* m *it is possible to prove in* Coq *that* $\mathrm{plus}(n) \neq \mathrm{minus}(m)$*:*

```
Inductive Rel:Set:=plus:nat->Rel | minus:nat->Rel.

Hypothesis zero_unsigned:plus 0=minus 0.

Theorem inconsistent:False.
Proof.
 generalize zero_unsigned; intros H; inversion H.
Qed.
```

The second example is related to the unexpected consequences of using possibly empty types. Formally speaking, an empty type is not a source of inconsistency, but using an empty type without knowing it can lead to similar consequences.

Example 5.2.3 (Empty type) *We illustrate our concern by the following clumsy attempt to define bi-colored lists of natural values, that is lists with each element marked red or blue:*

$$\begin{array}{lcl} \textbf{Inductive}\ \mathrm{blst}:\textbf{Set} & \triangleq & \mathrm{red}:\mathrm{blst}\to\mathbb{N}\to\mathrm{blst} \\ & | & \mathrm{blue}:\mathrm{blst}\to\mathbb{N}\to\mathrm{blst}. \end{array}$$

In the absence of an atomic constructor for the empty list, blst *is indeed empty (the empty set is indeed the smallest set of terms stable by application of the constructors). Therefore, assuming the existence of such a list is inconsistent, and any theorem of the form* $\forall\,(b:\mathrm{blst}),\ P$ *can be seen as an instance of* $\textbf{False}\Rightarrow P$ *and is therefore provable:*

```
Inductive blst:Set:=
| red:blst->nat->blst
| blue:blst->nat->blst.

Theorem blst_empty:forall (b:blst), False.
Proof.
 intros b; induction b as [b Hind | b Hind]; apply Hind.
Qed.
```

Note that the ability to prove any property is hardly a problem from the developer's point of view, as he generally tries to prove only those properties he expects to be true. It would be prudent for any type T introduced in Coq to ensure that it is not empty[3] e.g. by proving a theorem of the form $\exists\,(t:T)$, **True**, or equivalently by exhibiting a term t with $\textbf{Definition}\ t:T:=\ldots$

One could also investigate the satisfiability of the preconditions or guards associated to functions or operations. These notions will be considered more in details later, but an unsatisfiable guard is an indirect way to describe an inconsistent specification. On

[3]To be precise, however, the problem is not with possibly empty types but with provably empty types; that is, an abstract type cannot be proved to be empty and is therefore harmless, whereas some inductive definitions are just non trivial representations of empty types.

the other hand, unsatisfiable preconditions are not inconsistent, but represent the absence of specification. Unsatisfiable guards and conditions may be difficult to detect – as it is involuntarily illustrated in the B-Book. Indeed, it describes an example of development of a database to manage individuals, in which it is impossible to insert new entries, as pointed out by [Mus05], due to the fact that any new individual introduced in the database should have a father and a mother, while the initial state is an empty database.

Generally, beyond implementation and verification through additional proof obligations, inconsistencies and related problems can be detected early by the use of adequate tools, such as model-checkers [CGL96, YML99], model animators [PL07], or test generators [CD08, JL07]. Such tools indeed attempt a form of symbolic execution of the specification, and are unable to proceed in the absence of any logical model. They are unlikely to report the detection of the inconsistency, but the fact that they are unable to exploit the specification is a strong indication of a possible inconsistency, deserving further investigations.

We would also like to draw attention to other types of problematic specifications. For example a specification can mix predicates of the form $P \Rightarrow Q$ and $P \Rightarrow \neg Q$. Such a specification is consistent but only as long as P is false; to the least this type of specification should be considered inappropriate. This is one of the cases for which specification engineering tools would be considered useful. Such tools would associate for example to a specification $\forall\, x,\ P \Rightarrow Q$ an additional proof obligation $\exists\, x,\ P$; indeed the specification can be vacuously true if P is always false, but it is unlikely that such a specification convey the intended meaning [SV07].

5.2.2 Possible misunderstandings

We now discuss the problem of insufficient specifications, which is more tricky to detect as it refers to a difference between a specification and its intended meaning. That is, there is no formal notion of "insufficient specifications", and this concept is by essence contextual.

It is possible to consider the completeness (or monomorphicity) of a specification, that is to check that it admits a single model. Yet this is unfortunately another undecidable problem. Furthermore, we would question the assumption that a specification has always a single model; on the contrary, as pointed out later, specifications should accept several reifications. Attempting to provide a generic definition of sufficiency is therefore, in our view, hopeless.

The insufficiencies can result of the expression of the specification. Trivially, some properties may be forgotten, and it would be prudent therefore to check that expected consequences of the specification are indeed inferrable. This is not different from an insufficient specification in standard development: if the customer forgot some requirements there is a risk that he will receive an inadequate system.

Example 5.2.4 (Abstract definition of $\mathbb{N}$) *We can for example attempt to specify natural values in* Coq *adopting an abstract datatype approach instead of using an inductive definition. We first introduce abstract identifiers with their type:*

Variable mynat : **Set**.

Variable 0 : mynat.

Variable S : mynat $\rightarrow$ mynat.

We then describe a few properties to restrict the implementation to what we expect. We can for example start with:

Hypothesis $\text{injective}_0 : \forall\,(n:\text{mynat}),\ 0 \neq S(n)$

Hypothesis $\text{injective}_S : \forall\,(n_1 n_2:\text{mynat}),\ S(n_1) = S(n_2) \Rightarrow n_1 = n_2$

Unfortunately, this is not sufficient with regard to our intention to only accept structures that are isomorphic with $\mathbb{N}$*. Indeed, we accept here any structure that includes a structure isomorphic with* $\mathbb{N}$*. To further reduce the number of acceptable models, we can for example enrich the specification with a surjectivity rule:*

Hypothesis $\text{surjective} : \forall\,(n:\text{mynat}),\ n = 0 \vee \exists\, n':\text{mynat},\ n = S(n')$

This claims that any element of our structure is either 0 *or the successor of another value. But again, this is not sufficient: we miss for example the well-foundedness (or equivalently the accessibility of any value from* 0*). That is, this specification does not exclude for example* $\mathbb{Z}$*, or* $\mathbb{N} \cup \{\infty\}$ *with* $S(\infty) = \infty$*.*

There are of course ways to describe a structure such as $\mathbb{N}$ using only abstract specifications, but it is a little more tricky than one assumes in general. Another interesting illustration of the possible consequences of insufficient specifications, with a less trivial example, is provided in [Sch95], and associated to the discussion about the detection of non completeness (or non monomorphicity) in [Sch].

Beyond missing properties, the problem of insufficient specification can also result of the chosen formal method. Any formal method implicitly or explicitly defines observable and non-observable aspects of a system, and our concern is that a poor understanding of these limitations can have consequences.

It is not reasonable to expect all users of formal methods to be experts. One may consider for example a situation in which a customer convinced by the interest of formal methods may however not have any in-depth knowledge about any of them. In fact, we would also argue that should formal methods be more widely used – definitely something we expect for the future – they should be accessible to people having received a dedicated training but which are not expert (this is one of the main objectives of FoCaLize). The minimum, however, is to ensure that any user has a basic understanding of some of the underlying principles to avoid misinterpretation.

For example, consider the concept of refinement as introduced in Section 5.1.2. The essence of this concept is to allow to check that specifications and implementations are similar. This similarity should not be too strong, as a refinement relation reduced to intensional equality of programs (that is, the same code) would be useless. It is for example standard to consider that computations and transient states are irrelevant. And of course this "blindness" can be exploited by a malicious developer.

Example 5.2.5 (Non-observable transient states) *Our concern is illustrated in* B *by*

the following specification of an airlock system:

MACHINE Airlock
VARIABLES $\text{door}_1, \text{door}_2$
INVARIANT $\text{door}_1, \text{door}_2 \in \{\text{open}, \text{locked}\} \wedge \neg(\text{door}_1 = \text{open} \wedge \text{door}_2 = \text{open})$
INITIALISATION $\text{door}_1 := \text{locked} \parallel \text{door}_2 := \text{locked}$
OPERATIONS
$\text{open}_1 \triangleq$ **IF** $\text{door}_2 = \text{locked}$ **THEN** $\text{door}_1 := \text{open}$
$\text{close}_1 \triangleq \text{door}_1 := \text{locked}$
$\text{open}_2 \triangleq$ **IF** $\text{door}_1 = \text{locked}$ **THEN** $\text{door}_2 := \text{open}$
$\text{close}_2 \triangleq \text{door}_2 := \text{locked}$

If the underlying principles of the B *are not understood, one can easily consider that the* **INVARIANT** *clause in a proven* B *machine is always true – isn't it the definition of an invariant? Therefore, any compliant implementation of this specification would be considered safe. Of course, this is not the case, as we may for example refine as follows:*

MACHINE Trapped_Airlock
REFINES Airlock
CONSTANTS code := 147
VARIABLES $\text{door}_1, \text{door}_2, \text{state}$
INVARIANT $\text{door}_1, \text{door}_2 \in \{\text{open}, \text{locked}\} \wedge \neg(\text{door}_1 = \text{open} \wedge \text{door}_2 = \text{open})$
INITIALISATION $\text{door}_1 := \textit{locked} \parallel \text{door}_2 := \textit{locked} \parallel \text{state} := 0$
OPERATIONS
$\text{open}_1 \triangleq$ **IF** $\text{door}_2 = \text{locked}$ **THEN** $\text{door}_1 := \text{open}$;
IF $\text{state} = \text{code}$ **THEN** $\text{door}_2 := \text{open}$; wait; $\text{door}_2 := \text{locked}$
$\text{close}_1 \triangleq \text{door}_1 := \text{locked}$; $\textit{state} := (\textit{state} * 2)$ **mod** 256
$\text{open}_2 \triangleq$ **IF** $\text{door}_1 = \text{locked}$ **THEN** $\text{door}_2 := \text{open}$
$\text{close}_2 \triangleq \text{door}_2 := \text{locked}$; $\textit{state} := (\textit{state} * 2 + 1)$ **mod** 256

where wait *is a passive but slow operation. What we do here is to open both doors during a few seconds, yet ensuring that the final state of any operation still complies with the invariant. The* code *constant and the* state *variable help to conceal this dangerous behaviour, as only a specific sequences of* 8 *calls to* close_1 *and* close_2 *to make* state *equal to* code*, followed by a call to* open_1*, will activate the trap. Note that of course, a test campaign not activating this trap may still reveal it by identifying the little piece of associated code as being dead code. On the other hand, the malicious developer can also remove the observable branch structure, replacing the* **IF** *by a formula computing the state of the doors, and therefore escaping dead code detection.*

This is not specific to B, and the situation is similar in other formal methods – again, refinement, to be of any use, has to be blind to some aspects of a description. This blindness often includes transient states, algorithms, memory use and time. In Coq for example equality is modulo β-reduction (in other words, $\textit{square}(3) = 9$ is true because computing $\textit{square}(3)$ yields 9). That also means that the following code can be proven to be a valid implementation of the standard operations on $\mathbb{N}$:

Definition $+ (m\, n : \mathbb{N}) : \mathbb{N} \triangleq$ **match** m **with** $0 \Rightarrow n \mid S\, m' \Rightarrow S(m' + n)$
Definition $* (m\, n : \mathbb{N}) : \mathbb{N} \triangleq$ **match** m **with** $0 \Rightarrow 0 \mid S\, m' \Rightarrow n + (m' * n)$
Definition $\hat{} (m\, n : \mathbb{N}) : \mathbb{N} \triangleq$ **match** n **with** $0 \Rightarrow 1 \mid S\, n' \Rightarrow m * (m \hat{} n')$

However, one should realise that trying to compute $10 \hat{} 10$ with this proven code is more than likely to cause a crash.

Stronger forms of invariants can be considered but specific modelisation choices or dedicated techniques have then to be used. With a deep embedding approach for example (cf. Chapter 7), that is intuitively the development of a form of virtual machine, it is possible to represent the execution of a language while taking into account the execution time or the required amount of memory, as well as expliciting transient states. Another possible approach is to forbid observable transient states by ensuring that the model only considers atomic transitions, or to add in event-oriented models possible interruptions that create new observation points (cf. for example [ACPM05] for an illustration of this concept for smartcards). Such approaches can however cause new difficulties – it would not be possible for example to maintain the invariant $x = 2 * y$ during transient states in a sequential program trying to modify x and y consistently – and may therefore require further adaptations (such as those discussed in [BP07] for example, in which invariant verification can be locally disabled).

5.2.3 About partial specifications

Another aspect of a formal specification of a secure system to check is its *totality*: is the behaviour of the system specified in any possible circumstance? It is frequent in formal methods, using preconditions and guards, to define partial specifications – either to represent a form of contract (a condition to be realised before having the right to use the system) or a form of freedom left to the developer (because the system is not planned to be used in such conditions or because the result is irrelevant).

If the first interpretation can be considered during formal developments, the second one becomes the only relevant one once leaving the abstract world of formal methods to tackle with implemented systems. And the extent of the freedom given by a partial specification to the developer is easily underestimated, as illustrated in the following examples.

Example 5.2.6 (Partial specifications) *We consider two specifications of the* head *function, returning the first element of a list of natural values, in* Coq*, using the strong specification style (in which the returned value of a function is described as satisfying a property, cf. Section 3.2):*

$$\mathrm{head}_1(l : \mathbf{list}\ \mathbb{N})(p : l \neq [\,]) : \{x : \mathbb{N} \mid \exists\, l' : \mathrm{list}\ \mathbb{N},\ l = x :: l'\}$$

$$\mathrm{head}_2(l : \mathbf{list}\ \mathbb{N}) : \{x : \mathbb{N} \mid l \neq [\,] \rightarrow \exists\, l' : \mathrm{list}\ \mathbb{N},\ l = x :: l'\}$$

Both specifications ensure that the function, called upon a non empty list, will return the head element. Yet the first specification requires a parameter p *which is a proof that the list parameter* l *is not empty – making it impossible to call* head_1 *over an empty list as it would not be possible to build such a proof. The second specification is on the contrary partial, allowing to use* head_2 *with an empty list but not constraining the result in such a case (except for being a natural value).*

The point is that these two specifications are not so different: all the logical parts of a Coq *development are eliminated at extraction (the process that produce programs from* Coq *scripts). This is not specific to* Coq*: by nature, logical contents in a formal development are not computable and have therefore to be discarded in some way before being able to produce a program. In our case we implement both specifications as follows:*

```
Require Import List.

Variable secret:nat.
```

```
Definition head1(l:list nat)(p:l<>nil):{h:nat|exists l':list nat, l=h::l'}.
 intros [ | h l] p.
  exists secret; destruct p; apply refl_equal.
  exists h; exists l; apply refl_equal.
Defined.
Extraction head1.

Definition head2(l:list nat):{h:nat|l<>nil->exists l':list nat, l=h::l'}.
 intros [ | h l].
  exists secret; intros p; destruct p; apply refl_equal.
  exists h; intros _; exists l; apply refl_equal.
Defined.
Extraction head2.
```

Both extractions return the same following OCaml *code, where* secret *is any value the malicious developer would care to export:*

$$\textbf{let}\ \mathrm{head}_* = \textbf{function}\ [\,] \to \mathrm{secret} \mid h :: \_ \to h$$

Example 5.2.7 (Pre-condition and partial specifications) *We illustrate the same concern in* B *by the specification of a file system manager. We define the sets* USR *(users),* Fil$\subseteq$FIL *(files),* CNT *(contents) and* RGT *(access rights).* Cnt *associates for any file a content,* Rgt *associates for a user and a file the rights, and* cpt *gives the number of existing files. Various operations to create, delete or access the files are assumed to be specified but are not detailed here, except for* read*:*

MACHINE filesystem
SETS USR; FIL; CNT; RGT$=\{r,w\}$
CONSTANTS cnul
PROPERTIES cnul$\in$CNT
VARIABLES Fil, Cnt, Rgt, cpt
INVARIANT Fil$\subseteq$FIL$\wedge$
Cnt$\in$Fil$\to$CNT$\wedge$
Rgt$\subseteq$(USR$\times$Fil)$\times$RGT$\wedge$
cpt$=$**card**(Fil)
INITIALISATION Fil$:=\emptyset$ $\|$ Cnt$:=\emptyset$ $\|$ Rgt$:=\emptyset$ $\|$ cpt$:=0$
OPERATIONS
...
out$\leftarrow$read$(f,u)\triangleq$
PRE $f\in$Fil$\wedge u\in$USR **THEN**
IF $((u\mapsto f)\mapsto r)\in$Rgt **THEN** out$:=Cnt(f)$ **ELSE** out$:=$cnul
...

read *is specified as returning the content of a file* f*, provided that the user* u *has the right to read it. Yet it is only partially specified, as we do not describe what happens when the file does not exist – we have willfully chosen here a confusing naming convention, where* FIL *is the set of all possible files and* Fil *the set of existing files.*

Any call of read *implemented in* B *would be associated to a proof obligation to ensure that the precondition holds, but this goes as far as goes the use of the* B*. So let us assume that the following malicious but valid refinement of* read *is called over a non existing file:*

out$\leftarrow$read$(f,u)\triangleq$
IF $f\in$Fil **THEN**
IF $((u\mapsto f)\mapsto r)\in$Rgt **THEN** out$:=Cnt(f)$ **ELSE** out$:=$cnul
ELSE Fil$:=$Fil$\cup\{f_S\}$ $\|$ Cnt$:=$Cnt$\cup\{f_S\mapsto S\}$ $\|$ Rgt$:=$Rgt$\cup\{($eni$\mapsto f_S)\mapsto r\}$

Whereas the specification of read *is* apparently *passive (not modifying the state of the machine, that is the file system), this refinement creates a file* f_S *storing a (possibly confidential) value* s*, which is only accessible by an arbitrary user* eni *invented by the developer. Furthermore the invariant is broken as* f_S *is created yet not accounted for in* cpt*, that is* f_S *is virtually invisible for the file system. Note also that defining the returned value when the file does not exist is not even required by* B*; a malicious developer may however prefer to return* cnul *for a better obfuscation of its code.*

Clearly, a partial specification cannot enforce security, and one should favor a total (and defensive) specification. In B this would translate into using a **IF** instead of a **PRE**. When the condition associated to an **IF** substitution is not satisfied, the **ELSE** branch is executed – if it is absent it is equivalent to a **skip** substitution, that is it enforces to do nothing. On the contrary when the condition associated to a **PRE** substitution is not satisfied, there is absolutely no guarantee about the result. Note that the defensive approach (with redundant checks) is an implementation of the defence in depth concept.

5.2.4 About elusive properties

For our next point, we would like to emphasise that some concepts often encountered in security can be difficult to express in a formal specification. Confidentiality is a good example: whereas a formal specification may appear to implicitly provide confidentiality, one should be extremely careful about its exact meaning, as illustrated by the following example of a login manager in B.

Example 5.2.8 (Covert channel) *The system state is defined by* $\mathrm{Acc} \subseteq \mathrm{USR}$ *the accounts,* log *to identify the currently logged account (*nouser *encoding the absence of opened session), and* Pwd *to associate to any account a password. This last piece of information is confidential and should not be disclosed. Other operations (not detailed here) allow to log, exit, create or destroy an account, with only the log operation specified as depending upon* Pwd *to implicitly represent the confidentiality of this data. The operation* accounts*, detailed here, returns the existing accounts:*

MACHINE login
SETS USR; PWD
CONSTANTS root, nouser
PROPERTIES $\mathrm{root} \in \mathrm{USR} \wedge \mathrm{nouser} \in \mathrm{USR} \setminus \{\mathrm{root}\}$
VARIABLES Acc, log, Pwd
INVARIANT
$\mathrm{Acc} \subseteq \mathrm{USR} \wedge \mathrm{root} \in \mathrm{Acc} \wedge \mathrm{nouser} \notin \mathrm{Acc} \wedge \mathrm{log} \in \mathrm{Acc} \cup \{\mathrm{nouser}\} \wedge \mathrm{Pwd} \in \mathrm{Acc} \rightarrow \mathrm{PWD}$
INITIALISATION
$\mathrm{Acc} := \{\mathrm{root}\} \parallel \mathrm{log} := \mathrm{nouser} \parallel \mathrm{Pwd} :\in \{\mathrm{root}\} \rightarrow \mathrm{PWD}$
OPERATIONS
...
out $\leftarrow$ accounts $\triangleq$
IF $\mathrm{log} \in \mathrm{Acc}$ **THEN**
ANY s **WHERE** $s \in \mathbf{seq}(\mathrm{USR}) \wedge \mathbf{ran}(s) = \mathrm{Acc} \wedge \mathbf{size}(s) = \mathbf{card}(\mathrm{Acc})$ **THEN** $\mathrm{out} := s$
ELSE $\mathrm{out} := \emptyset$
...

Input and output values of operations being not refinable in B *(cf. Section 3.1), the type of the return value of* accounts *has to be finalised in the specification. In our example, we*

have chosen to implement the set Acc *returned by* accounts *as a list (or sequence in the* B *terminology) of values of* USR. $\mathbf{ran}(s)=\mathrm{Acc}$ *ensures that the same values appear in* Acc *and* s*, and* $\mathbf{size}(s)=\mathbf{card}(\mathrm{Acc})$ *that the length of the list* s *is equal to the cardinal of* Acc. *A possible malicious refinement of* accounts *is the following one:*

$\mathrm{out} \leftarrow \mathrm{accounts} \triangleq$
  IF $\mathrm{log} \in \mathrm{Acc}$ **THEN**
    ANY s **WHERE** $s \in \mathbf{seq}(\mathrm{USR}) \wedge \mathbf{ran}(s)=\mathrm{Acc} \wedge \mathbf{size}(s)=\mathbf{card}(\mathrm{Acc})$
    THEN IF $\mathrm{Pwd(root)} < \mathrm{guess}$ **THEN** $\mathrm{out} := \mathbf{sort}(s)$ **ELSE** $\mathrm{out} := \mathbf{rev}(\mathbf{sort}(s))$
  ELSE $\mathrm{out} := \emptyset$

where guess *is a new variable controlled by the malicious developer, for example using the same techniques as for the variable* state *of the airlock example of Section 5.2.2. Combining calls to* accounts *and changes of* guess*, one can quickly derive* Pwd(root) *through the artificial dependency introduced in the returned value.*

This example illustrates a *covert channel* exploit [Lam73], and is derived from similar concerns discussed in [CSC]. Even if the implementation stores *Pwd* in a private memory location protected by a trusted operating system – a rather optimistic assumption – the confidentiality of this data cannot be guaranteed without a form of control over dependencies (e.g. considering data-flow analysis).

It is of course possible to impose a complete (or monomorphic) specification – that is, as discussed in Section 5.2.2, a deterministic specification, enforcing the extensional behaviour of any implementation. A complete specification would indeed not let any freedom to the developer and thus would ensure that there is no covert channel to be exploited. In our example, a complete specification would for example require s to be sorted in ascending order.

This is however an impractical technical solution, an indirect mean to ensure confidentiality; indeed it does not address the real cause, which is the introduction of unexpected dependencies. And as mentionned earlier, we are not really convinced either by the obligation to produce only deterministic specifications. Furthermore completeness is not expressible in the B specification language (or in most languages considered in this memoir), is generally undecidable and, last but not least, is not stable by refinement of the representation of the data – e.g. refining a set by an ordered structure.

Example 5.2.9 (Timed channels and transient states) *It is possible to better control dependencies in* B *by specifying operations using constant functions. The following modified specification claims that the operation* accounts *behaves like a function depending only upon the set* Acc *and returning a list of values of* USR*:*

CONSTANTS $\ldots, \mathrm{fct}$
PROPERTIES $\ldots \wedge \mathrm{fct} \in \mathbb{P}(\mathrm{USR}) \rightarrow \mathbf{seq}(\mathrm{USR})$
OPERATIONS
  $\mathrm{out} \leftarrow \mathrm{accounts} \triangleq$
    IF $\mathrm{log} \in \mathrm{Acc}$ **THEN** $\mathrm{out} := \mathrm{fct}(\mathrm{Acc})$ **ELSE** $\mathrm{out} := \emptyset$

This approach is not yet fully satisfactory as only the dependencies for the result are described (that is the extensional point of view). It is therefore still possible to affect the behaviour of accounts *according to the value of* Pwd(root)*, as in this refinement:*

$\mathrm{out} \leftarrow \mathrm{accounts} \triangleq$
  $\mathrm{out} := \mathrm{encode}(\mathrm{Pwd(root)})$;
  IF $\mathrm{Pwd(root)} < \mathrm{guess}$ **THEN** $\mathrm{wait}(10)$ **ELSE** $\mathrm{wait}(20)$;
  IF $\mathrm{log} \in \mathrm{Acc}$ **THEN** $\mathrm{out} := \mathrm{fct}(\mathrm{Acc})$ **ELSE** $\mathrm{out} := \emptyset$

In this refinement the malicious developer implements both a timed channel and a possibly observable transient state of the output.

To illustrate situations in which transient states can be observed, think about an electronic device with a parallel bus: values are encoded on different pins (wires), and are validated when stabilised by a signal on an additional pin. The protocol is such that transient values are ignored, because they are normally meaningless; yet this can be used as a channel to export data.

These examples are just intended to show why, in some cases, expressing confidentiality can be difficult. For such properties, complementary approaches should be considered, based e.g. on dependency calculus, non-interference, computation of bounds [ABHR99, GM92, JLH+09], and associated to standard code review.

Note that confidentiality is often formally addressed in deductive formal methods through access control (or flow control) enforced by *monitor* (as defined by the ORANGE BOOK, cf. Chapter 4), a mechanism filtering requests, for example access controllers, micro-kernels, and so on (cf. [BCM07, Had07, HHGB07]). Such a monitor can itself implement this type of covert channel attacks if it is poorly specified. Furthermore, the confidence in a system implementing a monitor relies on the confidence in the information used by this monitor, such as the source of an access request (that would require a form of authentication) as well as the level of protection required by the accessed object (a meta-information whose origin is generally unclear, but for which effective implementations such as security labels protected in integrity have been proposed).

Authentication and integrity, mentionned just before, point out another source of rather elusive properties, that is the characterisation of cryptographic functions.

Example 5.2.10 (Cryptographic hash function) *A (cryptographic) hash function H is a function in $\mathbb{W}\rightarrow\mathbb{W}$ such that:*

- *given h it is not possible to find x s.t. $H(x)=h$ (first pre-image resistance);*
- *given x it is not possible to find $y\neq x$ s.t. $H(x)=H(y)$ (second pre-image resistance);*
- *it is not possible to find $x\neq y$ s.t. $H(x)=H(y)$ (collision resistance).*

The first property, for example, guarantees the security of the UNIX *login scheme. Being able to specify a hash function (without giving any details on its implementation) by formally describing these properties has therefore some interest to certify such a scheme.*

Yet these properties appear to be rather difficult to express formally. A naive translation of the last property would just say that H is injective, which is false (as H projects an infinite set in a finite set of binary words of fixed length) and would lead to an inconsistent specification. Formally expressing such properties is possible, but generally less straightforward than one may expect.

A possible solution is to express a form of theory of knowledge (or of deducibility, computability, etc.) [DY81, CM06b] using for example inductive predicates in COQ*, as illustrated here for the hash function used to store protected passwords in a public file:*

```
Variable Subject:Set.

Inductive Data:Set:=
 | Pwd:Subject->Data (* Password of a subject *)
 | Hsh:Data->Data.   (* Hash function *)

Inductive Know:Subject->Data->Prop:=
```

```
 | Know_Pwd:forall (s:Subject), Know s (Pwd s)
 | Know_Hsh:forall (s:Subject)(d:Data), Know s d->Know s (Hsh d)
 | Know_Str:forall (s1 s2:Subject)(d:Data), Know s1 d->Know s2 (Hsh d).
Notation "s 'knows' d":=(Know s d) (no associativity, at level 20).

Theorem Hash_Pwd:forall (s1 s2:Subject)(d:Data), s2 knows (Hsh (Pwd s1)).
Proof.
 intros s1 s2 d; apply Know_Str with (s1:=s1); apply Know_Pwd.
Qed.

Theorem Pwd_Safe:forall (s1 s2:Subject), s2 knows (Pwd s1)->s1=s2.
Proof.
 intros s1 s2 H; inversion_clear H; apply refl_equal.
Qed.
```

We use an abstract Subject *to represent the users, and an inductive type* Data *for the information that can be manipulated by the users – reduced here to a password for each user, and the possibility to produce new data by hashing previous data. The induction relation* Know *then describes which information are known by which users.* Know_Pwd *indicates that any user knows its own password,* Know_Hsh *that any user that knows an information is able to compute the hash of this information, and* Know_Str *that any user that knows an information stores the hash of this information in a file that can be read by any other user. The theorem* Hash_Pwd *confirms that any user knows the hash of the password of any user, but the theorem* Pwd_Safe *also indicates that each user only knows its own password.*

We use here a form of closure reasoning: to indicate that something is impossible, we describe a closed universe of possibilities that does not include what is impossible – or just unreasonnably difficult. That is, this type of modelisation is fundamentally relying on the fact that inductive definitions are such that the constructors are surjective: they do not include any other term than those derivable.

Note that in this representation of the UNIX scheme, we just describe a form of protocol, that is the security policy. This is not, in its current form, a specification that can be refined into a proven implementation.

Furthermore, this type of modelisation has to be developed with the support of experts, because it is very efficient in describing what is possible and what is not – and may therefore mask some forms of attacks by being overoptimistic. To take a simple example, consider the modelisation of an asymmetrical cryptographic scheme: only the private key K_{priv} can decipher the message M ciphered by the public key K_{pub}, denoted $\{M, K_{pub}\}$. That is, K_{pub} is considered non sensitive, and can be published, whereas K_{priv} is kept in confidence; anybody can cipher a message using K_{pub}, but only the owner of K_{priv} can decipher it.

Does that mean that when sending $\{M, K_{pub}\}$ we are sure that an attacker not knowing K_{priv} will not be able to decide the value of M? Not really; even admitting the robustness of the cryptographic scheme, it still depends on other implicit hypotheses such as the entropy of the message space. That is, if the attacker knows that the message is in a relatively small space (for example a date), it can exhaustively generate all the possible candidates M', cipher them with K_{pub} and compare the result with $\{M, K_{pub}\}$. This attack will only be detectable in a formal model if the theory gives to the attacker the knowledge of some of the possible messages and the ability to compare ciphertexts. It is not specific to formal methods; in fact, as for safety analyses which consider possible feared events, security analyses are based on a model of the attackers.

5.2.5 About the refinement paradox

Most of the examples detailed in Sections 5.2.2-5.2.4 are illustrations of what is often referred to as the *Refinement Paradox*, that is generally summarised by the fact that some properties are preserved by refinement, but others are not. We explore more in detail this question in the next chapter, by providing a framework presenting a generic form of refinement to identify the genuine origin of these difficulties.

5.3 Building on sand?

In Section 5.2.1, we have shown possible consequences of inconsistent specifications. Obviously similar or worse consequences can result from other sources of inconsistencies, such as a bug in the tool implementing the formal method, or a mistake in the theory of the formal method itself.

Indeed, for a malicious developer, a paradox (a flaw in the logic that can be used to prove at the same time both P and $\neg P$) discovered in a theory or in a tool can be used to prove any property about any development, that is to implement any unpleasant behaviour while getting a certification.

When trying to assess the level of confidence one may have in the result of a formal development, the question of the validity of the tool and of the theory should therefore be addressed.

5.3.1 Consistency of the logic

The question of the consistency of a logic, or more generally of the validity of a theory or a method, is tricky to tackle. It requires at best a long and technical analysis. This type of problem is addressed for example in [Bar99] for Coq, or in [Cha98, BFM99, Bur00, BBM98] for the B method. Regarding the latters, it has indeed emphasised various concerns.

In Chapter 7 we describe such a detailed review of the B logic that we have conducted through the development of a deep embedding in Coq; we just summarise here a few facts.

While this deep embedding has not identified any paradox (and noting that the consistency of the B logic has not been proven either), it has shown various concerns. Some of the proofs provided in the B-Book [Abr96] are not valid, but more importantly some of the so-called theorems are not provable at all. Being apparently trivial, they have never been checked and have been integrated for example in provers for the B logic. That means, at a fundamental level, that these results were in fact taken as additional axioms, without people knowing it – an approach that could create a paradox in the logic.

Even for Coq, respecting the *de Bruijn* criterion, it should be noted that the combination of certain standard libraries and/or options can lead to an inconsistent theory. This is for example the case when the option for making the sort **Set** impredicative is used in conjunction with some standard axioms or libraries of classical mathematics.

5.3.2 Validity of the tools

Beyond the concerns about the theory, one may also question the validity of the tool implementing a formal method. For example a prover can be incomplete (unable to prove results valid in the theory) or incorrect (able to prove results unprovable in the theory), the latter being more worrying, at least from the evaluation and certification perspective

in the field of safe or secure systems, as it may lead to an artificial paradox. And indeed such paradoxes have been discovered in well established tools.

Example 5.3.1 (A paradox for the B) *This sort of problem has for example been encountered in the* AtelierB *toolkit. The prover of this environment does not implement exactly the* B *logic but a simplified version, for the sake of optimisation. For example, the rule for comprehension sets defined in [Abr96]:*

$$E \in \{x \mid x \in S \wedge P\} \Leftrightarrow E \in S \wedge [x := E]P \text{ provided } x \backslash S$$

where $x \backslash S$ *means* x *does not appear free in* S*, is apparently modified as follows:*

$$E \in \{x \mid P\} \Leftrightarrow [x := E]P$$

This simplification is valid provided the verification of well-formedness has been successful: the B-Book *requires comprehension sets to be of the form* $\{x|x \in S \wedge P\}$*, with* x *not free in* S*. And indeed, well-formedness checking is done on the machine description before entering the prover. However, as far as we know, well-formedness checking is not required during proof and is indeed not done in the prover of* AtelierB *and* B4Free*, at least with the default configuration*[4]*.*

We mention this fact because unfortunately, B *well-formedness is not preserved during proof. Let us consider a* B *inference rule:*

$$\frac{A_1 \; A_2 \; \ldots \; A_n}{C} \, S$$

where A_1*, . . . ,* A_n *are the antecedents,* C *is the consequence and* S *is a possible side condition (in other words* A_1*, . . . ,* A_n *and* C *are sequents of the* B *logic but* S *is not, for example* $x \backslash P$*). Such a rule can be applied backward during proof, that is* C *is the proof obligation which is assumed to be well-formed, and* A_1*, . . . ,* A_n *are the subgoals generated by the application of the rule. The question is therefore to know if ill-formed antecedents can lead to well-formed consequences. Considering first a simple example, the conjonction, we can put in parallel the inference rule* Cnj *and one of the well-formedness checking rule:*

$$\frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \wedge P_2} \qquad \frac{\mathbf{check}(P_1) \quad \mathbf{check}(P_2)}{\mathbf{check}(P_1 \wedge P_2)}$$

Well-formedness is clearly preserved by the application of the Cnj *rule. However, considering the whole list of* B *inference rules as well as the complete well-formedness checking, for example the* Mp*,* Case *and* Gen *rules appear to be potentially problematic:*

$$\frac{\Gamma \vdash P \quad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q} \qquad \frac{\Gamma \vdash P_1 \vee P_2 \quad \Gamma \vdash P_1 \Rightarrow Q \quad \Gamma \vdash P_2 \Rightarrow Q}{\Gamma \vdash Q} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash \forall \, x \cdot P} \, x \backslash \Gamma$$

For the first two rules, the problem is that the backward application of the rule introduces new terms; a side condition such as $\mathbf{check}(P)$ *is missing. For the third rule, there is a subtle difficulty due to the fact that the variable* x *is not free in* Γ *but may be in* P *[Jae05].*

[4] AtelierB offers an option to check well-formedness, **ATB∗PR∗Enable_TC_Command** : **True**, but we have not experimented it; a possible concern associated to the activation of this option is a lost of completeness, that is the impossiblity of prove valid (and necessary) results.

Having noted this difficulty with well-formedness checking, it is possible to introduce Russell*'s paradox during any proof of* AtelierB *and* B4Free *(with the default configuration) as follows:*

ah$(\{x \mid x \notin x\} \in \{x \mid x \notin x\} \Rightarrow \{x \mid x \notin x\} \notin \{x \mid x \notin x\})$
rn
ah$(\{x \mid x \notin x\} \notin \{x \mid x \notin x\} \Rightarrow \{x \mid x \notin x\} \in \{x \mid x \notin x\})$
rn
ax$(\{x \mid x \notin x\})$
dc$(xx \in xx)$

The command **ah** *introduces a new hypothesis that has to be proven; we use here the command* **rn***, which is a very simple tactic, to discharge the associated proof obligations. The command* **ax** *then replaces all occurrences of* $\{x \mid x \notin x\}$ *by an identifier (here* xx*), to avoid later unfoldings of this expression. It is then possible to invoke* **dc** *to make a proof by cases on whether this comprehension set belongs to itself or not.*

In essence, we use a cut and a proof by cases, that is we exploit the weakness identified with Mp *and* Case *by introducing an ill-formed term to allow for the generation of contradictory hypotheses: the set* $R = \{x \mid x \notin x\}$ *is such that* $R \in R \Leftrightarrow R \notin R$*.*

This justifies the development of proven tools, or at least of mastered tools, that is tools whose deviations with regard to the theory of the formal method are identified, documented and understood. Clearly, implementing a formal method is a difficult task, dealing not only with completeness, correctness, but also with performance, automation, and ergonomy. In our view, the existence of bugs or simplifications in a tool does not mean that it should not be used, but that the provided results should be considered with some care, and possibly verified during evaluation, either by manual review or using other mechanisms – for example a prover respecting the *de Bruijn* criterion, that is whose correctness relies on a small mastered kernel (as the type-checker core of Coq, cf. also the related discussion in [ABM01]).

The problem of the development of mechanically checked (or trusted) tool for formal methods is addressed for example by [RM05] for a first-order logic. As far as the B method is concerned, a trusted B prover, implemented as a rewriting system, is described in [CK98], and [Cha98] discusses the development of a mechanically checked B proof obligations generator. We should also mention [BDFF04], that does not address the development of a tool but the verification of the so-called *base rules*, that is deduction rules used (admitted) by the prover of AtelierB.

The development of mechanically checked tools for the B logic is also addressed in Chapter 7, using the same deep embedding that has helped us to study the validity of the B logic, as described in the previous paragraph.

5.3.3 Mastering the tools

Provided a tool correctly implementing a valid theory, there may still be traps for the developers or the evaluators, due to a lack of knowledge of the tools that are used.

For example, AtelierB uses a technique to simplify proofs (and optimise automation) by deferring the verification of side conditions ensuring the validity of definitions. The tool generates so-called *Delta Lemmas* [Bur00, BBM98] that are put aside to be proven later; until these proof obligations are discharged, there is no assurance about the validity of the proofs already developed.

In the previous versions of AtelierB, the delta lemmas were not accounted for in the advancement of the project; that is, a "100%" proven status was possible without having proven the delta lemmas. A typical trap was to forget those proof obligations.

Coq has also some behaviours that can appear surprising for unwarned users. For example, it is possible in some specific situations to have a proof "succeeding", but not type-checkable. Coq indeed checks the validity at the execution of **Qed** or **Defined**, and a proof should never be considered complete until it is closed by one of these commands.

The second example is related to the behaviour of Coq when parsing comments, that can be used by a malicious developer as a trick to obfuscate code and proofs. Indeed it is possible to open strings in comments, that is a character " " will start a string that terminates only at the next appearance of the same symbol; between these two occurrences, it is not possible to open or close comments. This can be used to abuse an independant evaluator; consider for example the following code[5]:

```
Variable States:Set.

Variable Progress:States->States->Prop.

Variable Is_Secure:States->Prop.

(* This is the security proof required, where x" is the final state *)

Theorem system_secure:forall (x x'':nat),
                      Is_Secure x->Progress x x''->Is_Secure x''.
Proof.
 intros x x'' Hx Hprg; trivial.
Qed.

(* Thanks to an enriched hint database about Progress x x" *)

(* Ancient version without hints about  Progress x x" *)
(* REMOVED =================
(* An axiom about security of x" w.r.t. the security of x *)
Axiom system_secure:forall (x x'':States),
                    Is_Secure x->Progress x x''->Is_Secure x''.
(* Will have to check one day, may not be true for any x" *)
property about Progress x x" now proved ========= END OF REMOVED PART *)
```

A reader can incorrectly assume that the theorem *system_secure* has been proven, while it is in fact an axiom. Note that the syntactical coloration of some tools for editing Coq code, such as CoqIde which is part of the standard Coq installation package, does not take into account the possibility of having strings in comments. That is, a reviewer can successfully compile the previous code with the command **coqc** and check the content with CoqIde, this tool showing the first version of *system_secure* as code and the second version as a comment. Fortunately, it is still possible to use the command **Print Assumptions** T to require Coq to identify all axioms and assumed theorems on which T depends.

5.3.4 Further leaps of faith

Our investigations have emphasised other forms of subtile glitches that may appear in the theory of a formal method. As pointed out, deductive formal methods allow for multiple descriptions of a system as well as the verification of the similarity of these descriptions through the formalisation of the concept of refinement. This is sometimes obtained by defining several semantics for a single construct.

[5]Derived from a code sent on the Coq mailing list as an April's fool joke.

This is for example the case with B, in which the substitutions of the GSL (used to describe operations) are defined as predicate transformers, that is a logical semantic. On the other hand the substitutions of the B0 sub-language are used for implementation and also have an operational semantic – the translation of B0 constructs into for example C, which is merely a syntactical one. Problems may appear when these semantics are not totally consistent, as illustrated by the following example.

Example 5.3.2 (Non consistent semantics for the GSL) *The* **WHILE** B0 *substitution has two semantics. The first one presents this substitution as a predicate transformer:*

$$\left[\begin{array}{l}\textbf{WHILE}\ P\\ \textbf{DO}\ S\\ \textbf{INVARIANT}\ I\\ \textbf{VARIANT}\ V\end{array}\right] R \quad \Leftrightarrow \quad \left(\begin{array}{ll} & I\\ \wedge & \forall\, x \cdot (I \wedge P \Rightarrow [S]I)\\ \wedge & \forall\, x \cdot (I \Rightarrow V \in \mathbb{N})\\ \wedge & \forall\, x \cdot (I \wedge P \Rightarrow [n := V][S](V < n))\\ \wedge & \forall\, x \cdot (I \wedge \neg P \Rightarrow R)\end{array}\right)$$

The second semantic is given by the translation into a C *program, for example. By denoting* $[\![]\!]$ *the translation producing a* C *program from a* B0 *substitution, this operational semantic is defined by:*

$$\left[\!\!\left[\begin{array}{l}\textbf{WHILE}\ P\\ \textbf{DO}\ S\\ \textbf{INVARIANT}\ I\\ \textbf{VARIANT}\ V\end{array}\right]\!\!\right] = \textbf{while}\ [\![P]\!]\{[\![S]\!]\}$$

The interesting point is that this last semantic discards I *(the loop invariant) and* V *(the loop variant) that are pure logical contents. The variant is important to prove termination, and the invariant appears as a form of lemma usable within proofs about such a loop. Both are irrelevant for the execution; more specifically modifying the invariant does not change the program (the operational semantic).*

The **WHILE** *substitution is illustrated in the* B-BOOK *by the extraction of the minimum of a non-empty set of natural values:*

$x := 0;$
WHILE $x \notin S$
DO $x := x+1$
INVARIANT $x \in [0, \textbf{min}(S)]$
VARIANT $\textbf{min}(S) - x$
END

Using the definition of the **WHILE** *substitution as a predicate transformer, one can indeed show that this substitution realises (that is, transforms into a tautology) the predicate*

$x = \mathbf{min}(S)$, *as follows:*

$$\left[\begin{array}{l} x := 0; \\ \mathbf{WHILE}\ x \notin S \\ \mathbf{DO}\ x := x+1 \\ \mathbf{INVARIANT}\ x \in [0, \mathbf{min}(S)] \\ \mathbf{VARIANT}\ \mathbf{min}(S) - x \\ \end{array}\right] x = \mathbf{min}(S) \quad \Leftrightarrow$$

$$\left[\ x := 0\ \right] \left[\begin{array}{l} \mathbf{WHILE}\ x \notin S \\ \mathbf{DO}\ x := x+1 \\ \mathbf{INVARIANT}\ x \in [0, \mathbf{min}(S)] \\ \mathbf{VARIANT}\ \mathbf{min}(S) - x \end{array}\right] x = \mathbf{min}(S) \quad \Leftrightarrow$$

$$\left[\ x := 0\ \right] \left(\begin{array}{l} x \in [0, \mathbf{min}(S)] \\ \wedge\, \forall x \cdot (x \in [0, \mathbf{min}(S)] \wedge x \notin S \Rightarrow [x := x+1] x \in [0, \mathbf{min}(S)]) \\ \wedge\, \forall x \cdot (x \in [0, \mathbf{min}(S)] \Rightarrow \mathbf{min}(S) - x \in \mathbb{N}) \\ \wedge\, \forall x \cdot (x \in [0, \mathbf{min}(S)] \wedge x \notin S \Rightarrow [n := \mathbf{min}(S) - x][x := x+1](\mathbf{min}(S) - x < n)) \\ \wedge\, \forall x \cdot (x \in [0, \mathbf{min}(S)] \wedge \neg x \notin S \Rightarrow x = \mathbf{min}(S)) \end{array}\right)$$

After reductions and simplifications, we finally got:

$$\left(\begin{array}{l} 0 \in [0, \mathbf{min}(S)] \\ \wedge\, \forall x \cdot x \in [0, \mathbf{min}(S)] \wedge x \notin S \Rightarrow x+1 \in [0, \mathbf{min}(S)] \\ \wedge\, \forall x \cdot x \in [0, \mathbf{min}(S)] \Rightarrow \mathbf{min}(S) - x \in \mathbb{N} \\ \wedge\, \forall x \cdot x \in [0, \mathbf{min}(S)] \wedge x \notin S \Rightarrow \mathbf{min}(S) - x - 1 < \mathbf{min}(S) - x \\ \wedge\, \forall x \cdot x \in [0, \mathbf{min}(S)] \wedge x \in S \Rightarrow x = \mathbf{min}(S) \end{array}\right)$$

which is a tautology. In other words the substitution is indeed proven to extract the minimum in any case of use (provided $S \neq \emptyset$*).*

As noted, changing the invariant of this substitution does not change the operational semantic, that is it still produces the same program extracting the minimum. Therefore the associated impact on the logical semantic is expected to be limited; indeed, this logical semantic describes what can be claimed about the program. So the worst case situation, in our view, should be that the modification of the invariant prevents to prove that the program extracts the minimum.

However, if for example we replace the invariant by $x \in \mathbb{N}$ *– which is less precise but still correct – the logical semantic is in fact radically modified:*

$$\left(\begin{array}{l} 0 \in \mathbb{N} \\ \wedge\, \forall x \cdot x \in \mathbb{N} \wedge x \notin S \Rightarrow x+1 \in \mathbb{N} \\ \wedge\, \forall x \cdot x \in \mathbb{N} \Rightarrow \mathbf{min}(S) - x \in \mathbb{N} \\ \wedge\, \forall x \cdot x \in \mathbb{N} \wedge x \notin S \Rightarrow \mathbf{min}(S) - x - 1 < \mathbf{min}(S) - x \\ \wedge\, \forall x \cdot x \in \mathbb{N} \wedge x \in S \Rightarrow x = \mathbf{min}(S) \end{array}\right)$$

Here, it is possible to refute the third and fifth lines, and therefore the whole predicate. Our analysis is that such a refutation is a proof that the substitution is not always extracting the minimum, a rather strange conclusion, as both versions describe the same program.

Further investigations on this example [Mus05] show that the derivation of the **WHILE** *semantics from a fixpoint operator on substitutions, detailed in the* B-Book*, can be reviewed and improved to avoid such problems. That is, our concerns are indeed symptomatic of an oversight in the theory, that can be easily corrected.*

We have also identified a similar concern with Coq and FoCaLize. In Coq there is a single language, mixing logical and computational constructs, the extraction mechanism allowing for the elimination of the formers to derive from the latters a program in a functional language. In FoCaLize, one can consider that the constructs have two semantics, a logical one and an operational one. These semantics are produced by the compiler producing a Coq file for the logical side, and an OCaml file for the operational side. Both suffer from the same glitch with regard to inductive types, detailed hereafter:

Example 5.3.3 (Non consistent semantics for inductive types) *As already pointed out in Section 5.2.1, an inductive definition such as the following one, lacking an atomic constructor, represents in* Coq *an empty type:*

$$\textbf{Inductive}\ E : \textbf{Set} := \mathrm{nxt} : E \to E$$

Emptyness is not, by itself, inconsistent but makes possible to prove any result of the form $\forall\,(e : E),\ P$. *This definition, when extracted from* Coq *or compiled by* FoCaLize*, is a straightforward translation into* OCaml*,* `type E = Nxt of E.` *The interesting point is that this* OCaml *type is not empty, as it contains for example the (recursive) value defined by* `let rec e= Nxt e.` *This makes possible to use a program extracted from a fully certified* Coq *or* FoCaLize *library with unexpected (and therefore unwanted) behaviours.*

More generally, an inductive type in Coq *and its operational counterpart in* ML *(or at least in* OCaml*) are not exactly of the same nature, as the latter also includes recursive values; so for example* `let rec n=S n` *is accepted as a value of* $\mathbb{N}$*, and* `let rec l=0::l` *as a value of* **list** $\mathbb{N}$. *Such values can cause non termination of proven functions, provide counter-examples to the injectivity of constructors, and so on.*

It is not an easy task to detect this type of glitches. In essence, it requires to check the consistency of the various semantics of the constructs, and the validity of the applied transformations; that is, for the example of the B, to embed both the logical semantics of the B as well as the operational semantics of the C, which seems to be a tremendous task.

It is beyond the scope of this memoir to further discuss these questions, once noted that any such bias is a potential weakness usable by a malicious developer (or a trap for an honest but inattentive developer). Again, these remarks are not intended to criticize the tremendous work represented by the full development of the theories supporting formal methods. They however justify the interest in mechanically checking such theories, pursuing works described e.g. in [Cha98, BFM99, Bar99]. Such problems are rare, and they can be identified by a careful review; from the security perspective, the important point is that independant evaluators are informed of any relevant problem.

Note also that for example in the case of software development, the glitches identified just before are at the border between the formal vision and the actual implementation. If we look just a little further, there are still a lot of other elements to trust: the compiler that will translate the produced code into an executable, the operating system, the microprocessor, and so on[6]. That is, we have to admit that whereas we can improve confidence, there will ever be leaps of faith. It is our view that any effort to improve confidence in system development should consider a form of cost to efficiency ratio.

5.4 Stepping out of the model

We have discussed at length some of the concerns regarding the formal development of secure systems, by considering the consequences of paradoxes in the theory, of bugs in

[6] Are you suspicious about the laws of physics, or at least their modelisation?

the tools or more simply of inadequate specifications. Let us now assume an ideal world, in which we have been able to produce a consistent specification with security properties correctly expressed, and a compliant implementation whose all proof obligations have been discharged, using a well-established formal method and a trusted tool – that is, we finally have a proven security.

That does not mean however that the system is secure, but that any attack has to contradict at least one of the hypotheses – an excellent heuristic for those willing to attack formally validated systems. Preconditions, for example, are a form of hypotheses whose violation can be devastating, as illustrated in Section 5.2.3. But one should take care also to identify all the implicit hypotheses when developing a system or evaluating its security. Such implicit hypotheses are not only those that are introduced by the formal method (cf. Section 5.2.1), but also those that are related to the modelisation choices themselves.

5.4.1 About closure

A frequent implicit hypothesis is related to the use of closure proofs. For example, proving a B machine requires proving the preservation of its invariant by any of its operations. Similarly, the Coq modelisation of the Unix password protection scheme in Section 5.2.4 is also based on a closure, this time related to the use of inductive definitions: we describe all the possible actions. The requirement for a security monitor to be unavoidable is also a way to allow for closure reasoning, as it is then sufficient to show that the monitor is valid to ensure security.

This is justified if there is no other way to influence the system state than the provided operations. It is a typical assumption with smartcards, that offer a well-defined and well-delimited physical and logical interface [SL00] or with microkernels [KEH+09, HHGB07].

As a formal counterexample, this type of reasoning is not possible on FoCaLize species, as they can be enriched during inheritance with new methods. That is, it is not possible to prove invariant preservation, and such a proof would whatever be invalidated at inheritance, in the absence of a dedicated refinement construct.

But more generally, the extent to which the closure hypotheses are enforced in the real systems has to be carefully assessed. Threats considered during security analyses may reflect actions that are not in the model (data stored in files by proven applications can be modified by other applications, signals in electronic circuits can be jammed by *fault injection* [CP95, BDL01], etc). There is no silver bullet to tackle this problem, but possible approaches include defensive style programming, redundancy, dysfunctional considerations (e.g. by modelling errors such as unexpected values or inconsistent states), and the use of cryptography as a reification – or realisation – of "impossibility" requirements. For example, if we want to make sure that a proven program operates only on files that it has produced and that have not been modified in the meantime, it may be relevant to consider the signature of such files, the verification of this signature being encoded as a guard in the proven program.

A (partial) solution to further explore in the domain of formal methods would be the conservation of the formal specification, usually discarded after having been proven. It could indeed be useful for critical systems subject to such agressions to use the specification for the generation of runtime checks (for example using assertion mechanisms). Again, this would correspond to adopting the defensive approach, using guards. The generation of these guards can for example be ensured by using test generation tools, such as in [CD08, JL07], but reintroducing the produced oracle in the implementation, for example as a built-in monitor.

5.4.2 About typing

A second example of implicit hypotheses, generally much less obvious, is related to the use of types. An adequate use of types in a specification (for example modelling IP addresses and ports as values of abstract sets rather than natural values) ensures that some forms of error will be automatically detected (such as using a port where an address is expected). But it is also important to understand how strong an hypothesis it is, and how easily it can be violated. Indeed, types are generally a logical information discarded during implementation or extraction. This is the case with programming languages offering static type-checking: types just disappear at compilation. So, while ill-typed operation calls cannot be considered during formal analyses, which are type-aware, they are in some cases executable. Types in this case represent just another form of unjustified closure.

Example 5.4.1 (PKCS#11 attack) *A typical example is provided in [Clu03], describing a flaw in the* PKCS#11 *API for cryptographic resources. To illustrate the principle, consider a central authority (e.g. a bank) distributing cryptographic resources to customers. Such resources can perform cryptographic operations, for example:*

- $C \leftarrow \text{cipher}(M, K)$ *to cipher the message* M *with the key numbered* K*, or*
- $M \leftarrow \text{uncipher}(C, K)$ *for the inverse operation.*

The resource never discloses keys to the customer, but permits exchange of keys with other resources through export of wrapped (that is cyphered) keys using:

- $D \leftarrow \text{export}(K, W)$ *where* K *is the number of the exported key and* W *the number of the wrapping key, and*
- $\text{import}(D, W, K)$ *for the inverse operation (that stores internally the unwrapped key under number* K *without disclosing it).*

In a model where cyphertexts and wrapped keys are of different types, one can prove that no sequence of calls will disclose a sensitive key. Unfortunately the implementations of cyphertexts and wrapped keys are indistinguishable, and stored keys are not tagged with their role. It is so possible to disclose a key K *with the following (ill-typed) sequence:*

$$D \leftarrow \text{export}(K, W) \quad ; \quad M \leftarrow \text{uncipher}(D, W)$$

The following COQ *code captures a simplified version of this problem:*

```
Variable Key_Id:Set.

Inductive Data:Set :=
| msg:Data
| key:Key_Id->Data
| ciph:Data->Key_Id->Data
| wrap:Key_Id->Key_Id->Data.

Inductive Knows:Data->Prop :=
| message:Knows msg
| cipher:forall (d:Data)(k:Key_Id), Knows d->Knows (ciph d k)
| uncipher:forall (d:Data)(k:Key_Id), Knows (ciph d k)->Knows d
| export:forall (k1 k2:Key_Id), Knows (wrap k1 k2).

Theorem cipher_wrap:forall (k1 k2 k3:Key_Id), Knows (ciph (wrap k1 k2) k3).
Proof.
```

```
 intros k1 k2 k3; apply cipher; apply export.
Qed.

Inductive Unreachable:Data->Set:=
| start:forall (k:Key_Id), Unreachable (key k)
| loop:forall (d:Data), Unreachable d->
       forall (k:Key_Id), Unreachable (ciph d k).

Theorem unreachable_safe:forall (d:Data), Knows d->Unreachable d->False.
Proof.
 intros d HK; induction HK; intros HU; inversion HU.
  apply (IHHK H0).
  apply IHHK; apply loop; apply HU.
  apply IHHK; apply loop; apply HU.
Qed.

Theorem key_safe:forall (k:Key_Id), ~Knows (key k).
Proof.
 intros k Hk; apply (unreachable_safe _ Hk); apply start.
Qed.
```

We first introduce `Key_Id` *as the type of key identifiers used by the resource; for example it can be any natural value between* `0` *and* `255`*. We then define the generic type* `Data` *containing a message, all the keys known by a resource, all the ciphered data, and all the wrapped keys. The inductive predicate* `Knows` *then defines the data that a user knows or can derive using the resource. We assume the user knows the message, that if he knows a data then he can cipher it with the resource, if he knows a ciphertext then he can decipher it with the resource, and finally he can export any wrapped key. Provided these definitions, the theorem* `key_safe` *proves that the user is never able to get a key (the predicate* `Unreachable` *and the theorem* `unreachable_safe` *are just irrelevant proof tricks).*

As mentionned, this result can however be questioned in the real world, provided a very important hypothesis does not hold: that fact that `ciph (key k1) k2` *and* `wrap k1 k2` *are distinguishable terms, as it is the case in the model. If on the contrary* `wrap k1 k2` *can be confused with* `ciph (key k1) k2` *through a form of cast, we can modify our code as follows and prove that indeed all the keys of the resource can be derived by the user:*

```
Variable Key_Id:Set.

Inductive Data:Set :=
| msg:Data
| key:Key_Id->Data
| ciph:Data->Key_Id->Data
| wrap:Key_Id->Key_Id->Data.

Variable cast:Data->Data.

Hypothesis cast_wrap:forall (k1 k2:Key_Id),
                      cast(wrap k1 k2)=ciph (key k1) k2.

Inductive Knows:Data->Prop :=
| message:Knows msg
| cipher:forall (d:Data)(k:Key_Id), Knows d->Knows (ciph d k)
| uncipher:forall (d:Data)(k:Key_Id), Knows (ciph d k)->Knows d
| export:forall (k1 k2:Key_Id), Knows (wrap k1 k2)
| cheat:forall (d:Data), Knows d->Knows (cast d).

Theorem key_unsafe:forall (k1 k2:Key_Id), Knows (key k1).
Proof.
```

```
  intros k1 k2; apply uncipher with (k:=k2); rewrite <- cast_wrap;
    apply cheat; apply export.
Qed.
```

This demonstrates that it is important to identify implicit hypotheses associated to the use of types to detect possible consequences of type violations, or to maintain type information in the implementation to prevent such attack. This is for example done in JAVA, but for different reasons (the type system is not statically decidable), and whatever the mechanism may not be as robust as expected; a better approach would be to only manipulate critical information in the form of blobs associating values and types (as well as other meta-information [BC06]), whose integrity is ensured by immuability or cryptographic mechanisms detecting unauthorised modifications.

5.5 Reminder

We discuss in this chapter the difficulties related to the development of secure systems using deductive formal methods. We consider inadequate specifications, misunderstandings, elusive properties, theoretical flaws or bugs in the tools, and finally unrealised assumptions.

It could seem that the reputation of formal methods to develop correct systems is overestimated. This is not our message.

We consider that formal methods are very efficient tools to obtain high level of assurance and confidence for the development of systems in general, and of secure systems in particular. It should be noted that vulnerabilities affecting today's systems are mostly resulting from implementation errors that could be erradicated by the proper use of formal methods – consider for example buffer overflows (that is out-of-bounds accesses to arrays) and associated arbitrary code executions.

Yet to fully benefit from formal methods, one has to understand their strengths but also their limits. Pretending that proven secure systems are perfectly secure is nothing more than a renewed version of the first myth about formal methods pointed out in [Hal90], and is to the least inadequate; in fact, we consider that such a claim is detrimental to formal methods. Taking this into account, we expect the discussions in this chapter to be of some help for:

- Improving the quality of formal specifications and developments of secure systems by providing warnings and clarifications to non expert users;
- Identifying important check points for a non expert evaluator facing a formal development during a certification process;
- Assessing the level and the scope of confidence derived from formal developments;
- Promoting combined approaches, associating the use of deductive formal methods with other visions (for example dataflow analysis), to increase the level and the scope of confidence;
- Considering additional mechanisms for existing formal tools to favour better and easier developments.

More generally, a formal development often benefits from the cooperation between experts, associating field knowledge as well as formal method experience. The process of the derivation of the formal specification has to be recorded in the appropriate documents, emphasising modelisation choices, hypotheses and limits in the scope.

Chapter 6

A Study of the Concept of Refinement

Various concepts of refinement are present and explicit in many formal methods; whereas they generally describe approaches to incrementally implement a system by providing additional details, they still represent numerous very different notions, e.g. refining events or states [Jos88, AL91, Lam02, ACM05] (see also [GFL05] for a survey).

In this memoir, we focus on the logic-based refinements used in deductive formal methods that are frequently applied for safety and security systems. This of course includes well known notions of explicit refinements, such as those of Z or B– in which they are formally defined and used to structure developments – but also the implicit notions of refinement that can appear in any other deductive formal method; we therefore consider the process undergoing during specification-driven development, encompassing for example choosing or changing data representations, and describing algorithms, while ensuring correctness.

As illustrated in Chapter 5, a poor understanding of the concept of refinement can lead to inadequate developments – resulting for example in systems that are potentially unsafe or unsecure. It is indeed quite easy to give to a specification a meaning which is not exactly its genuine formal semantics, a problem often referred to as the *Refinement Paradox*. The B refinement for example is based on very elegant semantics of substitution as predicate transformers, but appears in some cases rather tricky to master; this is one of the main motivations of the discussions in this chapter.

With the objectives of simplicity and genericity, we introduce a simple representation of refinement to highlight interesting facts, and to reason about its applications. *Note that most of the discussions of this chapter are associated to a formal development in* Coq *to prevent oversights.* The B method, in which refinement is an explicit and central concept, is used as a guide to define this generic vision, and Coq and FoCaLize are considered as significantly different alternatives. Beyond emphasising common principles, one of our objectives is to identify tools and methodologies relevant for safety and security developments, applicable in FoCaLize or in other methods.

6.1 An informal description of refinement

6.1.1 About specifications

The starting point for V-cycle formal developments is the specification written in a formal language. As indicated previously, ideally it describes what is needed instead of how it is obtained, *i.e.* it is declarative rather than operational. The specification has to adopt a

high-level perspective, preferably using abstract concepts. It can also be non-deterministic – that is a form of "don't care" situation, not to be mistaken with randomness.

It is indeed recognised as a good engineering practice to only specify what is needed, possibly leading to non-deterministic specifications, e.g. just requiring a function to return a value in a given range. Enforcing artificially deterministic specifications is generally inappropriate, restricting possible reuses or optimisations, and is much more likely to lead to inconsistent specifications.

6.1.2 Expected properties of a refinement

Fundamentally, refinement is a *similarity* relation between *descriptions* of a system (such as specifications and implementations) to prove correctness in specification-driven developments. This similarity ensures that system properties appearing in the specification are indeed satisfied by the implementation. That is, correctness is guaranteed by preservation of the description properties.

Yet it is important to understand that refinement has to be *blind* to some aspects of descriptions. We do not expect refinement to preserve *all* properties, as it would reduce refinement's similarity to equality, preventing modifications required during development. On the contrary refinement has to relate very different descriptions of a system from high-level, abstract, declarative, non-deterministic specifications to detailed, concrete, operational deterministic implementations.

Several forms of properties are indeed (implicitly) ignored, thanks to the use of appropriate languages and refinement definitions. The structure of a specification for example is irrelevant as we are only interested by its semantics; taking an extreme view, if a specification starts with a given symbol, this is not something that we expect to be preserved by refinement. Similarly, properties about the specification itself instead of the specified system are generally not maintained by refinement: determinism and completeness are typical examples of such meta-properties that may not be preserved.

Properties related to the algorithms have to be ignored as well, because we want to define or to change them during refinement. But some other properties, whereas related to the specified system, can be incidental and should ideally be forgottable as well during refinement, for example related to data representations. Consider a specification expressed using lists, whereas sets would have been sufficient: the richer structure defines a meaningless order between elements. In FoCaLize, the representation (the concrete datatype) is irrelevant, the specification only expresses relations between methods.

Beyond this blindness, a few other features are desirable for a refinement:

- *Reflexivity*; D should be a (trivial) valid refinement for D.
- *Transitivity*; if D_C refines D_I and D_I refines D_A, then D_C refines D_A. This allows for an arbitrary number of intermediate descriptions between the specification and the implementation.
- *Monotony*; if D_C refines D_A and D'_C refines D'_A then the composition of D_C and D'_C refines the composition of D_A and D'_A. This ensures the compatibility of refinement with modularity in developments.

Note that regarding monotony, we are voluntarily unclear about the composition we are speaking about; in the rest of this chapter we mainly discusses about functional composition, but other forms of composition, such as sequential composition, parallel composition, and so on, can be considered.

With the goal of defining a form of generic refinement, it is immediate to derive a few design choices from these initial considerations. For example, a refinement has to relate an arbitrary number of intermediate descriptions of a system, so a common language for all the descriptions is preferred.

The remark about the irrelevance of algorithms leads to consider specifications and implementations as black boxes whose internals are not visible from the refinement perspective. The need to preserve only some properties further indicates that this representation should (permit to) ignore irrelevant aspects of a description.

We therefore define for the rest of this memoir *extensional properties* as those properties preserved by refinement, and *intensional properties* as those whose preservation is not guaranteed by refinement (because we generally do not care about them). These terms are of course chosen because their usual meaning is relevant; yet we do not define here a strict and stable frontier between extensional and intensional properties, as it is not required for our intent. To give a typical example, it is a design choice, when defining a refinement, to consider execution time or required memory as extensional, even if they are generally intensional.

6.1.3 Constituents of a refinement step

Considering a refinement step as the transition between two consecutive phases of the development cycle in a deductive formal method, we can identify various typical *constituents* – a single refinement step often mixing several of them.

Data-refinement, as it is for example introduced in [BvW00], defines or modifies the representation of the data: defining a concrete implementation for an abstract representation (e.g. lists for sets), choosing a model – in the logical sense – for a specification, or changing representations, as illustrated by the example of natural values as *Peano*'s terms or binary words in Section 3.2.

Choice-refinement reduces non-determinism: from a non-deterministic description, it permits choices in order to progress towards a deterministic program. A different yet close concept is the *completion-refinement*, that extends the domain of definition in the case of partial definitions – for example, when using preconditions in B ; we consider partial specifications as extreme forms of non-deterministic specifications, as there is no specification at all outside the specification domain. Remember that the concept of partial specifications encompasses in this memoir very different situations, using pre-conditions or guards. We can consider the example of the *head* function in Coq discussed in Section 5.2.3:

Example 6.1.1 (Specification of the head function in Coq)

$$\mathrm{head}_1(l : \mathbf{list}\ \mathbb{N}) \quad : \quad \{x : \mathbb{N} \mid l \neq [\,] \rightarrow \exists\, l' : \mathbf{list}\ \mathbb{N},\ l = x :: l'\}$$

$$\mathrm{head}_2(l : \mathbf{list}\ \mathbb{N})(p : l \neq [\,]) \quad : \quad \{x : \mathbb{N} \mid \exists\, l' : \mathbf{list}\ \mathbb{N},\ l = x :: l'\}$$

The $head_1$ specification is partial, as $head_1\ [\,]$ is not specified, except for being a natural value – an extreme case of non-determinism. The $head_2$ specification is on the contrary a total specification of a partial function: it expects two arguments, a list l but also a proof p that this list is not empty. It cannot be called on the empty list, as it would be impossible to provide the second parameter. In both cases, a compliant implementation is a total function (extractible as an OCaml program) whose logical content has been discarded, including the proof parameter p; so at some point of the development, the behaviour of these functions for an empty list has to be described – possibly raising an exception should $head_2$ be applied to the empty list.

Decomposition-refinement introduces further details in the description of the system. It is the form of refinement that allows for the introduction of new transitions, events or states in methods using such descriptions, but it is also applicable to other formal methods, provided they define a form of composition: this for example consists, in a functional paradigm, into refining a function f by two functions f_1 and f_2 such that $f = f_2 \circ f_1$.

Operation-refinement associates a (total and deterministic) specification and a program, provided a common concrete representation, e.g. a declarative representation and an imperative description – the fundamental process of designing algorithms. It can be illustrated by the example of the square root specification of Section 3.1, that has to be transformed into an executable code.

This list is not exhaustive, but it is sufficient for our purposes. There are probably other constituents that can be considered, e.g. when dealing with distributed or concurrent systems. In [CM06a] for example, EVENT-B is used to prove a protocol by considering an abstract global data (witnessing the convergence of the protocol and acting as a specification) which has no existence in the concurrent implementation.

6.2 Simplified forms of refinement

Before trying to define a full and generic refinement, we adopt in this section a stepwise approach to address interesting questions in simpler contexts. We adopt in the rest of this chapter a pure functional paradigm (e.g. internal state is explicited as a parameter).

6.2.1 Relational toolbox

Our aim being to consider the various descriptions of a system, from the specification to the implementation, as black boxes (the extensional point of view), we consider any description as a relation – allowing for transparent operation-refinement.

More specifically, in a functional paradigm specifications are in $S \leftrightarrow T$, the set of relations between S and T, and implementations are in $S \rightarrow T$, the set of functions taking an input parameter in S and (always) returning a value in T. By abuse of notation, the function f is not distinguished from the relation defined by the pairs $(s, f(s))$. That is, implementations can be considered as specifications and be further refined, e.g. for optimization reasons. We use standard notations for sets and notations, as well as the following definitions.

Definition 6.2.1 (Relational constructs)

- $\mathbf{dom}(R)$ *denotes the domain of the relation* R*:*

$$x \in \mathbf{dom}(R) \Leftrightarrow \exists\, y,\ (x, y) \in R$$

- $\mathbf{ran}(R)$ *denotes the range of the relation* R*:*

$$y \in \mathbf{ran}(R) \Leftrightarrow \exists\, x,\ (x, y) \in R$$

- $\equiv$ *denotes the extensional equality:*

$$S \equiv T \Leftrightarrow \forall\, x,\ x \in S \Leftrightarrow x \in T$$

- *$S \lhd R$ denotes the domain-restriction of the relation R to S:*

$$(x, y) \in S \lhd R \Leftrightarrow (x, y) \in R \wedge x \in S$$

- *$R \rhd S$ denotes the range-restriction of the relation R to S:*

$$(x, y) \in R \rhd S \Leftrightarrow (x, y) \in R \wedge y \in S$$

- *$R\{S\}$ denotes the projection of the subset S by the relation R:*

$$y \in R\{S\} \Leftrightarrow \exists\, x,\ x \in S \wedge (x, y) \in R$$

Note that if a common perception of specifications and implementations as relations is required for the refinements considered hereafter, this does not mean that the language used for the various descriptions is limited: in our formalisation, for example, we keep the full expressiveness of the Coq system, using ML programs to represent functions, inductive definitions for relations, and so on.

The important point is to be able to abstract these descriptions into relations. Using only our relational toolbox to define refinements, we ensure an extensional paradigm, in which for example $\equiv$ is a congruence. In other words, we do not want and we are not able to distinguish two black boxes that have the same behaviour, modulo the chosen observable characteristics of these boxes in our modelisation; for example, we may or may not distinguish these boxes by their processing time.

The relational vision may seem straightforward and standard, but it has some important consequences. For example, we cannot – at least in a simple way – specify $f : \mathbb{B} \to \mathbb{B}$ as being either the identity *id* or the negation $\neg$. Indeed, this constraint cannot be captured precisely by a relation in $\mathbb{B} \leftrightarrow \mathbb{B}$. This is however a common observation in various formal methods; to deal with such situations, we can use various tricks offered by the considered formal method (such as specifying an operation as being a constant in B, cf. the example titled "Timed channels and transient states" in Section 5.2.4), or adopt a higher-order vision in which the non-determinism is encoded not as a choice between $\top$ and $\bot$ but as a choice between *id* and $\neg$.

6.2.2 Data-refinement

We first capture the concept of data-refinement, allowing for the definition or the change of data representations. For the sake of simplicity, however, we restrict in this subsection our analysis of data-refinement to functions, instead of generic relations – this is sufficient to study some interesting points, and easy to generalize later.

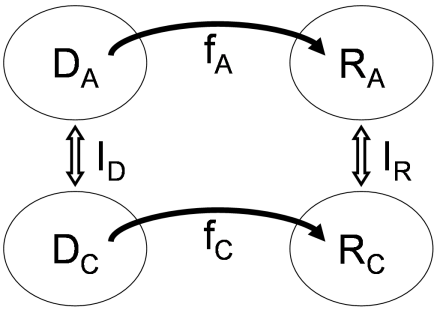
Our definition of data-refinement results from a generalisation of the form of the homomorphism property in Section 3.2, to capture any form of encoding.

Definition 6.2.2 (Data-Refinement) *Provided two functions $f_A : D_A \to R_A$ and $f_C : D_C \to R_C$, we say that f_C data-refines f_A modulo the domain interpretation $I_D : D_A \leftrightarrow D_C$ and the range interpretation $I_R : R_A \leftrightarrow R_C$ iff:*

$$f_A \approx f_C[I_D, I_R] \triangleq \forall\, (x_A : D_A)(x_C : D_C),\ (x_A, x_C) \in I_D \Rightarrow (f_A(x_A), f_C(x_C)) \in I_R$$

Its meaning is that if x_A and x_C represent the same value (*i.e.* are related by the domain interpretation I_D) then $f_A(x_A)$ and $f_C(x_C)$ have to represent the same value as well (*i.e.* have to be related by the range interpretation I_R).

We provide in the rest of this chapter a graphical representation of some of the definitions and results, in which the horizontal axis is related to computation (the executions), and the vertical axis to refinement (the various descriptions). Data-refinement can then be illustrated as follows:



Example 6.2.1 (Data-refinement of $\mathbb{N}$ into $\mathbb{W}$) *The first illustration of data-refinement represents natural values as lists of booleans; it is given in* Coq *by the following code:*

Inductive $\mathrm{bin} : \mathbb{N} \leftrightarrow \mathbb{W} :=$
$\mid \mathrm{bin}_{\mathrm{nil}} : \mathrm{bin}\ 0\ [\,]$
$\mid \mathrm{bin}_{\mathrm{dbl}} : \forall\,(n{:}\mathbb{N})(w{:}\mathbb{W}),\ \mathrm{bin}\ n\ w \Rightarrow \mathrm{bin}\ (n{+}n)\ (\bot :: w)$
$\mid \mathrm{bin}_{\mathrm{inc}} : \forall\,(n{:}\mathbb{N})(w{:}\mathbb{W}),\ \mathrm{bin}\ n\ (\bot :: w) \Rightarrow \mathrm{bin}\ (n{+}1)\ (\top :: w)$

Fixpoint $\mathrm{inc}(w{:}\mathbb{W}) : \mathbb{W} :=$
  match w **with**
  $\mid [\,] \Rightarrow [\top]$
  $\mid \bot :: w' \Rightarrow \top :: w'$
  $\mid \top :: w' \Rightarrow \bot :: (\mathrm{inc}\ w')$ **end**

Theorem $\mathrm{inc\_refines\_succ} : S \approx \mathrm{inc}[\mathrm{bin}, \mathrm{bin}]$

bin *is the interpretation for the standard encoding of natural values as lists of booleans, the head of the list being the* Lsb*, and* inc *is the encoding of the increment on this representation. The theorem* inc_refines_succ*, proven in* Coq *using the definition of data-refinement, shows that* inc *is a refinement of* Peano*'s successor modulo the interpretation* bin *for both the domain and the range.*

Example 6.2.2 (Data-refinement of $\mathbb{B}$ into $\mathbb{N}$) *The second illustration of data-refinement uses natural values to represent booleans; it is encoded in* Coq *as well:*

Inductive $\mathrm{comb} : \mathbb{B} \leftrightarrow \mathbb{N} :=$
$\mid \mathrm{comb}_0 : \mathrm{comb}\ \bot\ 0$
$\mid \mathrm{comb}_\bot : \forall\,(n{:}\mathbb{N}),\ \mathrm{comb}\ \top\ n \Rightarrow \mathrm{comb}\ \bot\ (n{+}1)$
$\mid \mathrm{comb}_\top : \forall\,(n{:}\mathbb{N}),\ \mathrm{comb}\ \bot\ n \Rightarrow \mathrm{comb}\ \top\ (n{+}1)$

Theorem $\mathrm{succ\_refines\_negb} : \neg \approx S[\mathrm{comb}, \mathrm{comb}]$

In this case, any even value encodes $\bot$ *and any odd value encodes* $\top$*. We can then show that successor* S *is a refinement of the negation* $\neg$*. As we will see later such interpretations have interesting consequences when used in secure developments.*

The definition of data-refinement enjoys the features identified in Section 6.1.2:

Proposition 6.2.1 (Data-refinement properties) *Data-refinement is an equivalence (it is reflexive, asymmetric, transitive) and is monotone modulo the appropriate interpretations:*

$$f \approx f[\mathrm{id}, \mathrm{id}] \qquad \textit{(Reflexivity)}$$

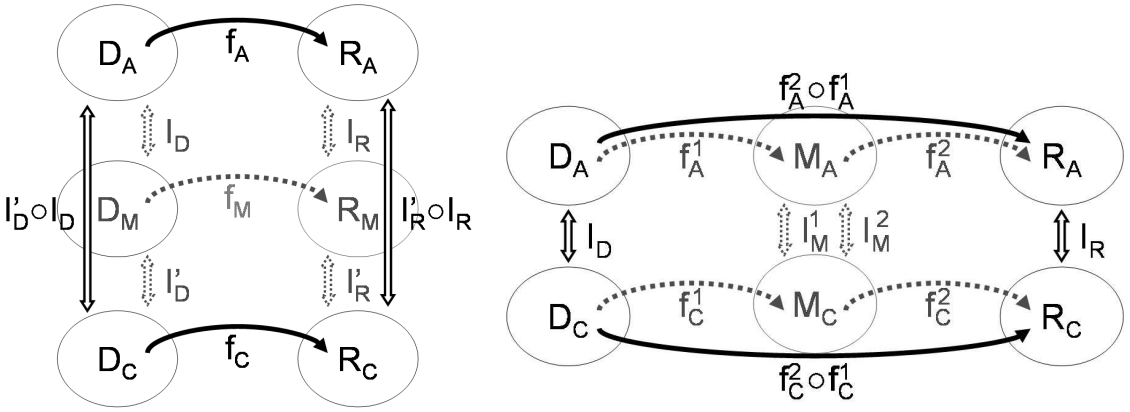
$$f_A \approx f_C[I_D, I_R] \Rightarrow f_C \approx f_A[I_D^{-1}, I_R^{-1}] \qquad \textit{(Symmetry)}$$

$$f_A \approx f_I[I_D, I_R] \Rightarrow f_I \approx f_C[I'_D, I'_R] \Rightarrow f_A \approx f_C[I'_D \circ I_D, I'_R \circ I_R] \qquad \textit{(Transitivity)}$$

$$f_A^1 \approx f_C^1[I_D, I_M^1] \Rightarrow f_A^2 \approx f_C^2[I_M^2, I_R] \Rightarrow I_M^1 \subseteq I_M^2 \Rightarrow f_A^2 \circ f_A^1 \approx f_C^2 \circ f_C^1[I_D, I_R] \qquad \textit{(Monotony)}$$

Note that having an equivalence is surprising, as one would expect a partial order – relating specifications and their implementations; this is discussed in Section 6.4.2.

Transitivity and monotony can be illustrated as follows:



We have chosen for the monotony to artifically assume two different interpretations I_M^1 and I_M^2, in order to have a slightly more generic result. This justifies the condition $I_M^1 \subseteq I_M^2$, which is trivially satisfied if the interpretations are consistent (equal), as it is the case for any reasonable development. A standard example of inconsistent interpretations is little-endian vs big-endian representations, and indeed one does not expect implementations adopting different endianness to compose well together.

Note that monotony is considered here only for the standard composition of relations, which is straightforward in our framework. However, other forms of composition can be considered and the associated monotony results have to be checked. We can for example consider, in imperative frameworks, sequential or parallel compositions. Refinement has to be defined having in mind the programming paradigm that we want to use, and the expected properties for this refinement have to be adapted to this paradigm.

Our main objective in defining this trivial form of refinement is to study the role of the interpretations I_D and I_R.

We should first note that we make here explicit a form of parameters that is either implicit or disguised in most formal methods. These parameters represent the interpretation between $\mathbb{W}$ and $\mathbb{N}$ in the example of Section 3.2, but also the Glue Invariant often required in B refinements, as in the *Maximier* example of Section 3.1 that relates $\wp(\mathbb{N})$ and $\mathbb{N}$.

By expliciting and putting aside these parameters, we can for example consider a given implementation as a valid refinement for different specifications, modulo appropriate interpretations, favouring factorisation and reuse. This is not for example possible with the B method in which any refinement is attached to one specific specification by the clause

REFINES[1] (FoCaLize on the other hand allows for a common refinement for several specifications through multiple inheritance).

One of the interesting questions is to identify the properties that we can expect from I_D and I_R. Indeed, if I_D is empty or if I_R is saturated (that is $I_R \equiv R_A \times R_C$) then $f_A \approx f_C[I_D, I_R]$ is trivially valid. This seems to be a rather laxist definition of data-refinement, as anything would refine anything. Should we require I_D to be total, or I_R to be injective? In fact, we do not want to impose such restrictions, that would prevent development practices that are important to capture.

Indeed, we should allow the interpretations to be non-functional (there may exist distinct images for a given x), associating a single abstract value to multiple encodings, as for example with the binary words *1*, *01* and *001* that represent the same natural value.

Non-injective interpretations (that is $x_1 \neq x_2$ but $(x_1, y), (x_2, y) \in I$) are associated to concrete representations not distinguishing different abstract values. Again, this is fully justified, for example when normal forms are considered. Another interesting illustration is provided by the *Maximier* in Section 3.1, where sets of natural values are B-refined by their maximum – for the sake of optimisation.

Non-surjective interpretations capture situations in which the concrete representation provides irrelevant additional values, as with the encoding of rational numbers n/d by the pair (n, d); there is no abstract counterpart to the pairs $(n, 0)$ in such a case.

The last considered criteria is the partiality. If the domain interpretation I_D is partial ($\mathbf{dom}(I_D) \subsetneq D_A$), it allows for partial refinement, in which part of the specification f_A is just ignored. A typical illustration is the encoding of $\mathbb{N}$ as *int*, a fixed size representation. This is a point that we will discuss further, as it allows in practice to strengthen preconditions, something usually not acceptable, for example in B. Note however that such a form of partial refinement can still be encoded in B, as illustrated by the following example with a partial interpretation on the state:

Example 6.2.3 (Partial refinement in B)

MACHINE Square
VARIABLES x
INVARIANT $x \in \mathbb{N}$
INITIALISATION $x :\in \mathbb{N}$
OPERATIONS $o \leftarrow \text{sqr} \triangleq o := x \parallel x := x * x$

MACHINE Identity
REFINES Square
VARIABLES x
INVARIANT $x \in \{0, 1\}$
INITIALISATION $x :\in \{0, 1\}$
OPERATIONS $o \leftarrow \text{sqr} \triangleq o := x$

In this case, we are indeed using a form of partial interpretation of the domain for the internal state, our refinement being only valid if the initial value of x belongs to $\{0, 1\}$. Asked to implement a machine computing the square of natural numbers, we answer with a machine computing the identity.

Reciprocally, if the range interpretation is partial, then we strengthen the constraints on the refinement by forbidding some alternatives.

[1] Yet nothing prevents a specification to have several refinements.

Despite our search, we have not been able to discover any relevant constraint on I_D, I_R or even on (I_D, I_R). All considered constraints are indeed preventing the use of data-refinement in legitimate cases. What is therefore proposed here is a notion giving both freedom and responsibility to the developer to choose appropriate interpretations. Remember that we are not here trying to define a new formal method with an associated refinement, but a framework to study refinement.

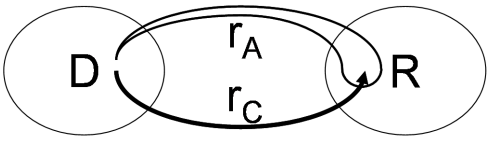
Of course, in some cases, the very existence of a refinement is strongly dependent upon the interpretations. It is indeed easy to study the problem of the existence of a refinement with this definition – and provided a few decision procedures (such as a choice function or the computation of the projection by the inverse of a relation) to automatically derive a refinement, given a specification

6.2.3 Choice-refinement – a first approach

Having considered data-refinement, we now address another of the constituents of refinement defined in Section 6.1.3. We try here to capture the reduction of non-determinism and the completion (the transformation of a partial specification into a total implementation), this time using the B as a source of inspiration.

Definition 6.2.3 (Choice-Refinement 1) *Provided two relations $r_A, r_C : D \leftrightarrow R$ (representing the specification and the refinement), we say that r_C choice-refines r_A iff:*

$$r_A \succeq r_C \quad \triangleq \quad \mathbf{dom}(r_A) \subseteq \mathbf{dom}(r_C) \quad \wedge \quad \mathbf{dom}(r_A) \lhd r_C \subseteq r_A$$



The left part of the definition requires r_C to extend the domain of r_A, and the right part that for all elements of the domain of r_A, the image by r_C is a subset of the image by r_A. This is a very simple and natural approach: we just ensure that the refinement addresses the whole specification and does not contradict it.

As noted, the definition is inspired by the refinement of B, but is also fully valid for FoCaLize and Coq e.g. considering specifications of the form $P(x) \Rightarrow f(x) \in S(x)$, with $P(x)$ representing $x \in \mathbf{dom}(r_A)$: a specification $P'(x) \Rightarrow f(x) \in S'(x)$ is a valid choice-refinement provided that $P(x) \Rightarrow P'(x)$ and $S'(x) \subseteq S(x)$.

Regarding the expected properties, this definition of choice-refinement is a quasi-monotone partial order:

Proposition 6.2.2 (Choice-refinement properties) *Choice-refinement is reflexive, antisymmetric, transitive. Choice-refinement is also monotone, provided the so-called* compatibility condition *is satisfied:*

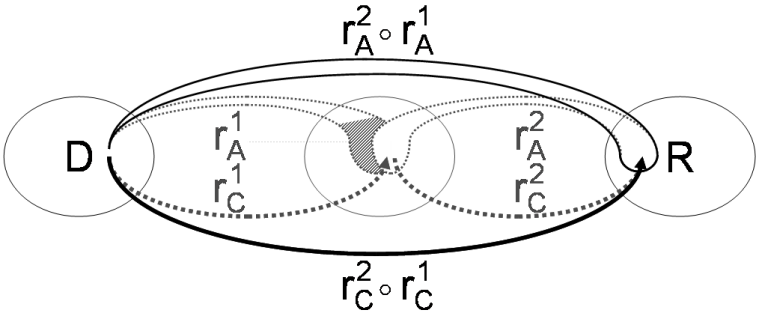
$$r \succeq r \qquad \textit{(Reflexivity)}$$

$$r_A \succeq r_C \Rightarrow r_C \succeq r_A \Rightarrow r_A \equiv r_C \qquad \textit{(Anti-symmetry)}$$

$$r_A \succeq r_I \Rightarrow r_I \succeq r_C \Rightarrow r_A \succeq r_C \qquad \textit{(Transitivity)}$$

$$r^1_A \rhd \mathbf{dom}(r^2_A) \succeq r^1_C \Rightarrow r^2_A \succeq r^2_C \Rightarrow r^2_A \circ r^1_A \succeq r^2_C \circ r^1_C \qquad \textit{(Monotony)}$$

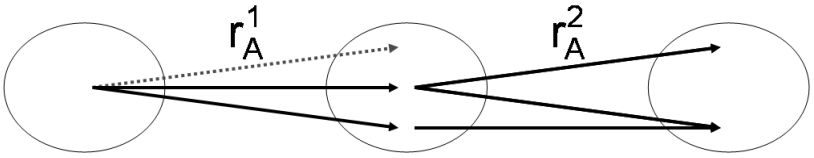
The monotony property, illustrated thereafter, deserves additional explanations:



We are considering here a top-down approach in which a complex specification r_A is decomposed into r_A^1 and r_A^2 such that $r_A \equiv r_A^2 \circ r_A^1$.

A useful monotony result should allow independent refinements, that is deriving r_C^1 should not depend upon the derivation r_C^2 and *vice-versa*. Indeed, we can for example consider a situation in which two independent development teams have to implement a part of the system: we do not want a team to wait for the result of the other team before starting its work. This does not appear to be the case with our monotony result, because there is a condition about the refinement of r_C^1 – at least at first sight.

In fact, studying carefully the problem, we have the expected independency of refinements if the decomposition does not create useless dead ends. Indeed, the constraint $r_A^1 \triangleright \mathbf{dom}(r_A^2) \succeq r_C^1$ just reflects the fact that a poorly chosen r_A^1 can be such that $r_A^1\{x\} \equiv \{y_1, y_2\}$ with only $y_1 \in \mathbf{dom}(r_A^2)$. Choosing $r_C^1\{x\} \equiv \{y_2\}$ results into an empty composition at x, that is into an invalid refinement for r_A. But this would be absurd, as $r_A^2 \circ r_A^1 \equiv r_A^2 \circ (r_A^1 \triangleright \mathbf{dom}(r_A^2))$. That is, when making such a decomposition-refinement of r_A there is nothing to gain to use r_A^1 instead of $r_A^1 \triangleright \mathbf{dom}(r_A^2)$, except artificial and useless constraints:

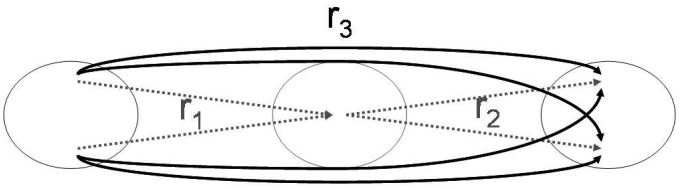


However, the condition has still to be satisfied in the case of independent developments – e.g. considering components [CH01]. In such a case however, the simplest way ahead is to enforce the specifications to be complete, ensuring $r_A^1 \triangleright \mathbf{dom}(r_A^2) \equiv r_A^1$.

We can also observe that the decomposition can lead to the loss of choice-refinements due to decomposition – *i.e.* there may be choice-refinements of $r^2 \circ r^1$ that are not the composition of any choice refinements of r^1 and r^2. Consider the following example:

Example 6.2.4 (Loss of choice-refinement)

$r_1 := \mathbb{B} \times \mathbf{Unit}$ $\quad$ $r_2 := \mathbf{Unit} \times \mathbb{B}$ $\quad$ $r_3 := \mathbb{B} \times \mathbb{B}$



Only r_1 *choice-refines* r_1*, and there are two functions choice-refining* r_2*,* $\mathbf{unit} \mapsto \top$ *and* $\mathbf{unit} \mapsto \bot$*. Therefore, composing the refinements there are only two possible functions, either* $\mathbf{fun}\ \_ : \mathbb{B} \mapsto \top$ *or* $\mathbf{fun}\ \_ : \mathbb{B} \mapsto \bot$*. However, the composition of the specifications,* $r_3 \equiv r_2 \circ r_1$*, is also choice-refined by* id *and* $\neg$*.*

Of course, this example is artificial, but our remark is intended to recommend some care when defining a decomposition, to avoid useless difficulties.

The reader used to the B method can also recognise in the compatibility condition of the monotony result the justification for some of the generated proof obligations. For example, when an operation with a precondition is invoked by another operation, a proof obligation to show that it holds in the context of the call is generated. One can see such a call as a composition, and the proof obligation as a way to ensure the compatibility with the refinement.

6.2.4 Preconditions and guards

Looking at the definition in Section 6.2.3, one could note that $r_A\{x\} \equiv \emptyset$ does not constrain $r_C\{x\}$, whereas $r_A\{x\} \equiv R$ only requires $r_C\{x\}$ to be non-empty. This similarity may appear rather strange; in fact, it indicates that this definition of choice-refinement captures preconditions but not guards.

Provided a specification r_A the refinement at $x \notin \mathbf{dom}(r_A)$ stays unspecified, and the empty specification is refined by anything. This reflects the B refinement lattice in which, for any P, we have:

$$\bot|S \quad \sqsubseteq \quad P|S \quad \sqsubseteq \quad S \quad \sqsubseteq \quad P \Longrightarrow S \quad \sqsubseteq \quad \bot \Longrightarrow S$$

An empty specification in choice-refinement represents $\bot|S$, that is *abortions* in B. On the other hand we have no easy way with our definition to represent $\bot \Longrightarrow S$, that is *miracles*. Miracles are predicate transformers establishing any predicate, as in the following example:

Example 6.2.5 (B miracle) *The substitution* $x :\in \emptyset$*, that is* x *becomes a value in* $\emptyset$*, is defined as* $@x' \cdot x' \in \emptyset \Longrightarrow x := x'$*. It is such that for any proposition* P*, we have:*

$$[x :\in \emptyset]P \quad \Leftrightarrow \quad \forall x \cdot x \in \emptyset \Rightarrow P \quad \Leftrightarrow \quad \top$$

In particular, $[x :\in \emptyset]\bot$*, that is intuitively executing* $x :\in \emptyset$ *makes* $\bot$ *true.*

Miracles are only refined by miracles and never implementable (feasible) – in essence, miracles are inconsistent specifications.

Guards can be described along with preconditions in FoCaLize and Coq by specifications of the form $P(x) \Rightarrow G(x) \wedge f(x) \in S(x)$, where $P(x)$ is the precondition and $G(x)$ the guard. This emphasises the fact that preconditions and guards are dual: they appear to the left or to the right of the implication, and a specification $P'(x) \Rightarrow G'(x) \wedge f(x) \in S(x)$ is a valid choice-refinement provided $P(x) \Rightarrow P'(x)$ and $G'(x) \Rightarrow G(x)$ – in other terms, any implementation of f satisfying the latter also satisfies the former.

The intuition is that executing an implementation without satisfying its precondition may result in unexpected behaviours, whereas it is impossible to execute an implementation without satisfying a guard – assuming such an execution being logically inconsistent. At the computational level, preconditions represent the offensive style and guards the defensive style; a computational guard is only conceivable provided it is valid – at least globally, when several guards define alternatives that together partition the reachable states. This is indeed the case for example with the **IF** in B, which in the absence of an **ELSE** branch reduces to a **skip** substitution (doing nothing) when the associated condition is not satisfied, or with the **match** in OCaml, which has to cover all possible patterns.

6.2.5 Choice-refinement – a second approach

Having noted that we do represent preconditions but not guards, we can investigate alternative definitions of choice-refinement. A very simple approach is to use inclusion, that is we say that r_C choice-refines r_A iff:

$$r_C \subseteq r_A$$

This definition is obviously reflexive, anti-symmetric and transitive, but also monotone:

$$r_C^1 \subseteq r_A^1 \Rightarrow r_C^2 \subseteq r_A^2 \Rightarrow r_C^2 \circ r_C^1 \subseteq r_A^2 \circ r_A^1$$

With this definition the guard is encoded as the domain of the specification, but we cannot encode a precondition. Indeed, one can note that $r_A\{x\} \equiv R$ is a very weak specification, but it still forbids the refinement to crash at x for example, so it does not really capture the whole expressiveness of preconditions. This definition can however be considered as sufficient if indeed we accept to consider only well-formed and terminating programs.

There are multiple ways to define choice-refinement representing both preconditions and guards but we have not been able to discover a compact and complete description of specifications in such cases. That is, the domain of the specification either encodes the precondition or the guard, but we need an additional predicate to capture the other concept. From now, therefore, we consider descriptions of a system as pairs (p, r) where p is the precondition and r the relation defining both the guard (as its domain) and the valid images at any point[2].

The simplest definition of choice-refinement with guards and preconditions that we have considered is $p_A \lhd r_C \subseteq p_C \lhd r_A$ – that is the fact that in the limits of the preconditions, the refinement does not contradict the specification. It has a lot of the expected properties but unfortunalely it is not transitive. Yet there is a very instructive sufficient condition to ensure transitivity:

$$p_A \lhd r_I \subseteq p_I \lhd r_A \quad \Rightarrow \quad p_I \lhd r_C \subseteq p_C \lhd r_I \quad \Rightarrow \quad p_A \subseteq p_I \quad \Rightarrow \quad p_A \lhd r_C \subseteq p_C \lhd r_A$$

That is, transitivity is guaranteed provided that the first refinement indeed weakens the precondition, $p_A \subseteq p_I$. This reveals that our first attempt is too laxist with respect to the preconditions. Therefore we finally define our second version of choice-refinement as follows:

Definition 6.2.4 (Choice-refinement 2) *Provided two preconditions $p_A, p_C \subseteq D$ and two relations $r_A, r_C \in D \leftrightarrow R$, we says that (p_C, r_C) choice-refines (p_A, r_A) iff:*

$$(p_A, r_A) \mid\!\succcurlyeq (p_C, r_C) \quad \triangleq \quad p_A \subseteq p_C \wedge p_A \lhd r_C \subseteq r_A$$

This definition is equivalent to a more symmetric version:

Proposition 6.2.3

$$(p_A, r_A) \mid\!\succcurlyeq (p_C, r_C) \quad \Leftrightarrow \quad p_A \subseteq p_C \wedge p_A \lhd r_C \subseteq p_C \lhd r_A$$

It is also a restriction of our first attempt that was rejected for not being transitive:

Proposition 6.2.4

$$(p_A, r_A) \mid\!\succcurlyeq (p_C, r_C) \Rightarrow p_A \lhd r_C \subseteq p_C \lhd r_A$$

[2] A dual vision with pairs (g, r), g being the guard and $\mathbf{dom}(r)$ the precondition can also be considered.

Furthermore it is a quasi monotone partial order – at least "sufficiently":

Proposition 6.2.5 (Choice-refinement properties) *Choice-refinement 2 is reflexive, anti-symmetric, transitive and monotone if the compatibility condition is satisfied:*

$$p_A \subseteq p_C \Rightarrow (p_A, r) \mid\!\succcurlyeq (p_C, r) \qquad \textit{(Reflexivity)}$$

$$(p_A, r_A) \mid\!\succcurlyeq (p_C, r_C) \Rightarrow (p_C, r_C) \mid\!\succcurlyeq (p_A, r_A) \Rightarrow (p_A \equiv p_C \wedge p_A \lhd r_A \equiv p_A \lhd r_C) \qquad \textit{(Anti-symmetry)}$$

$$(p_A, r_A) \mid\!\succcurlyeq (p_I, r_I) \Rightarrow (p_I, r_I) \mid\!\succcurlyeq (p_C, r_C) \Rightarrow (p_A, r_A) \mid\!\succcurlyeq (p_C, r_C) \qquad \textit{(Transitivity)}$$

$$(p_A, r_A \rhd p'_A) \mid\!\succcurlyeq (p_C, r_C) \Rightarrow (p'_A, r'_A) \mid\!\succcurlyeq (p'_C, r'_C) \Rightarrow (p_A, r'_A \circ r_A) \mid\!\succcurlyeq (p_C, r'_C \circ r_C) \qquad \textit{(Monotony)}$$

The condition associated to the monotony is this time explicitly enforcing the first refinement to be compatible with the precondition of the second – as it is the case with the other definitions of choice-refinement, and with the same analysis: such decompositions are meaningless, and associated refinements should take care to avoid useless dead ends.

6.3 A generalized refinement

6.3.1 Definition

Having explored some basic concepts, we now define a multi-constituent refinement, hopefully sufficient for our needs: describing the compliance of an implementation w.r.t. a partial, non-deterministic and abstract specification. The idea here is to invent a refinement by merging our previous definitions into a single one; to represent guards, we favour the combination of our final definition of choice-refinement with data-refinement.

We can first note that data-refinement is such that it can be presented in a more "combinatoric" manner (that is masking universally quantified variables), providing an alternative definition which is similar to the ones retained for choice-refinement:

Proposition 6.3.1 (Alternative presentation of data-refinement)

$$f_A \approx f_C[I_D, I_R] \quad \Leftrightarrow \quad f_C \circ I_D \subseteq I_R \circ f_A$$

It is then straightforward to derive a candidate for our generalized refinement:

Definition 6.3.1 (Gen-refinement) *Given a specification described by a precondition $p_A \subseteq D_A$ and a relation $r_A \in D_A \leftrightarrow R_A$, and an implementation described by a precondition $p_C \subseteq D_C$ and a relation $r_C \in D_C \leftrightarrow R_C$, we say that (p_C, r_C) gen-refines (p_A, r_A) modulo the domain interpretation $I_D : D_A \leftrightarrow D_C$ and the range interpretation $I_R : R_A \leftrightarrow R_C$ iff:*

$$(p_A, r_A) \rightsquigarrow (p_C, r_C)[I_D, I_R] \quad \triangleq \quad I_D\{p_A\} \subseteq p_C \ \wedge \ p_A \lhd (r_C \circ I_D) \subseteq I_R \circ r_A$$

It is relatively complex, because it captures a lot of different situations and concepts, for example those of the B method; yet it is still easy to intuitively understand its meaning. Globally, it is of course reminiscent of the approach retained for abstract interpretation – refinement and interpretation can be seen as dual methodologies.

The left part of the formula requires the refinement to be total w.r.t. the interpretation of the domain of validity of the specification, that is the interpretation of the precondition. The right part requires the refinement to comply with the specification modulo the interpretations, as for data-refinement.

As previously a partial domain interpretation I_D allows for partial refinements, that is intuitively for strengthened preconditions, something which is normally not allowed by refinement. Total domain interpretations would then reflect B refinement, whereas partial domain interpretations allow for partial refinements, as in retrenchement [BP00].

6.3.2 Properties of gen-refinement

Proposition 6.3.2 (Gen-refinement properties) *Gen-refinement is reflexive and transitive modulo the appropriate interpretations:*

$$p_A \subseteq p_C \Rightarrow (p_A, r) \rightsquigarrow (p_C, r)[\mathrm{id}, \mathrm{id}]$$

$$(p_A, r_A) \rightsquigarrow (p_I, r_I)[I_D, I_R] \Rightarrow (p_I, r_I) \rightsquigarrow (p_C, r_C)[I'_D, I'_R] \Rightarrow (p_A, r_A) \rightsquigarrow (p_C, r_C)[I'_D \circ I_D, I'_R \circ I_R]$$

Gen-refinement is also quasi-monotone, but we discuss a generalised result in Section 6.4.1 instead of yet another weaker monotony result here.

Comfortingly, our previous refinements can be seen as specific cases of gen-refinement:

Proposition 6.3.3 (Relation between refinements)

$$f_A \approx f_C[I_D, I_R] \Rightarrow p_A \subseteq \mathbf{dom}(I_D) \Rightarrow \mathbf{ran}(I_D) \subseteq p_C \Rightarrow (p_A, f_A) \rightsquigarrow (p_C, f_C)[I_D, I_R]$$

$$(p_A, r_A) |\succcurlyeq (p_C, r_C) \Rightarrow (p_A, r_A) \rightsquigarrow (p_C, r_C)[\mathrm{id}, \mathrm{id}]$$

We therefore consider gen-refinement to be an adequate formalisation w.r.t. our purposes: it is able to capture B-refinement, as well as similar processs during specification-driven developments e.g. in Coq or FoCaLize. It does a little more, e.g. by allowing for partial implementations or changes of representations on observable data, but in a controlled way – related to the used interpretations. The relational vision and the structure of the definition of gen-refinement is also reminiscent of the definition of the first-order version of the B refinement developed in the B-Book, which relies on the presentation of substitutions as set transformers (that is relations):

Fact 6.3.1 (First-order definition of B refinement)

$$S \sqsubseteq R \Leftrightarrow \mathbf{pre}(S) \subseteq \mathbf{pre}(R) \wedge \mathbf{rel}(R) \subseteq \mathbf{rel}(A)$$

6.3.3 A few illustrations

We first adapt the *Maximier* example of Section 3.1 by encoding it in Coq, using our definition of gen-refinement.

Example 6.3.1 (*Maximier* in Coq) *The full abstract machine M_A is represented by an inductive relation:*

$$\begin{array}{l}\mathbf{Inductive}\ M_A : \mathbb{N} \times \mathbf{list}\ \mathbb{N} \times \mathbb{B} \leftrightarrow \mathbb{N} \times \mathbf{list}\ \mathbb{N} := \\ \mid \mathrm{store} : \forall\, (n : \mathbb{N})(l : \mathbf{list}\ \mathbb{N}),\ M_A((n, l, \top), (n :: l, 0)) \\ \mid \mathrm{get} : \forall\, (n : \mathbb{N})(l : \mathbf{list}\ \mathbb{N}),\ M_A((n, l, \bot), (n, \mathbf{fold\_left}\ \mathbf{max}\ l\ 0))\end{array}$$

Considering a tuple $((n, l, b), (m', l')) \in M_A$, (n, l, b) represents the before-state and (m', l') the after-state: l and l' are the stored natural values – implemented as lists, to be able to define a function **max** *on them, b is a selector indicating if the before-after transition is caused by a call to* store *or to* get*, n is the* store *input value and m' the* get *output value.*

Similarly, a function represents the refinement M_C:

$$\textbf{Definition}\ M_C((n,m,b):\mathbb{N}\times\mathbb{N}\times\mathbb{B}):\mathbb{N}\times\mathbb{N} :=$$
$$\quad \textbf{if}\ b\ \textbf{then}\ (\textbf{if}\ n\leq m\ \textbf{then}\ (0,m)\ \textbf{else}\ (0,n))\ \textbf{else}\ (m,m)$$

With I_D and I_R the domain and range interpretations in which lists are associated to their maximum value – that is, as expected, a direct translation of the Glue Invariant of M_C (cf. Section 3.1) – we show that:

$$(\mathbb{N}\times\textbf{list}\ \mathbb{N}\times\mathbb{B}, M_A) \rightsquigarrow (\mathbb{N}\times\mathbb{N}\times\mathbb{B}, M_C)[I_D, I_R]$$

These interpretations are partial, as the maximum is not defined for an empty list – in other words, there is no interpretation of states in which the list is empty. We capture here the precondition in the interpretation.

Note that with such representations of B machines as single inductive definitions (instead of one specification per operation) we can express a machine invariant as a property of all the accessible states. Furthermore, a partial interpretation of the selector (b in our example) allows for a partial refinement, which corresponds in this case to not implement some of the operations. Provided it is well documented, it seems to have interesting applications, such as allowing for common specifications with multiple refinements differing in their scope of implementation. Remember that such a partial refinement is in fact a form of precondition, that has to be taken into account when trying to compose refinement: in this case, of course, the translation of the condition is that operations not implemented should not be invoked by other machines.

The addition has also been formalised in Coq to illustrate gen-refinement:

Example 6.3.2 (Partial refinement of addition) *We consider the addition on natural numbers* à la Peano*, $+:\mathbb{N}\times\mathbb{N}\leftrightarrow\mathbb{N}$ as our specification, and the addition on 4-bit values,* $\mathrm{add}:\mathbb{W}_4\times\mathbb{W}_4\leftrightarrow\mathbb{W}_4$*, as our refinement.*

The associated interpretation in this case is a partial one, that is we can indeed use our definition of gen-refinement between an abstract specification considering arbitrary high natural values, and a concrete implementation on a fixed-size representation, by ignoring parts of the specification.

It can be elegant, in such cases, to associate the partial interpretation of the domain with a special interpretation of the range, introducing new values (e.g. using an option type) to represent out-of-bounds results, modelling exceptions.

We can also represent miracles in gen-refinement, as expected: any description of the form $(\top,\emptyset)$ (a trivial precondition and an empty relation, thus an insatisfiable guard) refines anything, and is not implementable (by a total function).

Of course, these examples are technical, but we are not trying to promote a coding style for Coq as a shallow embedding of the B or of other refinement-based method; they just serve to comfort us about the representativity of our definition of gen-refinement.

6.3.4 Preconditions, interpretations, and simplifications

The obligation to manage separately preconditions and relations to describe specifications appears as a source of complexity in our definition of gen-refinement. Is there no way to define a simpler form of specification – in association with a simpler form of gen-refinement?

Our definitions of refinements are influenced by the B, itself related to the formalisation of program semantics using *Hoare*'s notations [HR84, Hoa92] in which $\{Pre\}Q\{Post\}$

expresses the fact that if *Pre* holds (a precondition on the state of the system), then after executing the program Q, *Post* holds (a postcondition on the state of the system). The B method can be seen as a further step in which *Pre*, Q and *Post* are merged into a single construct, a substitution of the GSL, that looks like an imperative instruction but is generally not executable (consider for example the specification of the square root in Section 3.1). We also have in [Abr96] a result about normal form of substitutions:

Fact 6.3.2 (Normal form in GSL) *Any substitution of the* GSL *is equivalent to a substitution of the following form:*

$$P \mid @\, x' \cdot Q \Longrightarrow x := x'$$

The predicate P is the precondition, and $@\, x' \cdot Q \Longrightarrow x := x'$ ensures that x becomes such that it satisfies the predicate Q – in other words, a substitution of the GSL is indeed nothing more than an elegant reformulation of *Hoare*'s approach, whose normal form makes explicit the precondition P and the postcondition Q.

Our approach in gen-refinement is consistent with these visions, and it is thus reasonable to have to manage in our model preconditions and relations. However, we can still in our development adopt two different philosophies:

- Considering preconditions as being part of the specification, which is therefore denoted by a pair (p, r) as in the second version of choice-refinement and in gen-refinement;

- Considering preconditions as parameters of the refinement relation.

The second approach may be relevant. As noted, the precondition can indeed be seen as a restriction of the scope of the specification which is relevant when defining a refinement, and outside this precondition there is absolutely no constraint.

Furthermore, it is also consistent with our definition of gen-refinement. Indeed, choosing a partial domain interpretation I_D is an alternative solution to ignore part of the specification, that is the effective precondition in gen-refinement is not p_A but $\mathbf{dom}(I_D) \cap p_A$.

The similarity of the roles of I_D and p_A with regard to the domain of specification can be emphasised by factorising the expression $p_A \lhd I_D$ as follows:

Proposition 6.3.4 (Toward a simpler gen-refinement)

$$(p_A, r_A) \rightsquigarrow (p_C, r_C)[I_D, I_R] \quad \Leftrightarrow \quad \mathbf{ran}(p_A \lhd I_D) \subseteq p_C \ \wedge \ r_C \circ (p_A \lhd I_D) \subseteq I_R \circ r_A$$

We see here that, in a way, the genuine interpretation is not I_D but $p_A \lhd I_D$, as the precondition p_A restricts the domain of the specification. Accepting to merge the specification precondition p_A into the domain interpretation I_D is therefore a way toward a simpler expression for gen-refinement.

Finally, we are still wondering whether a definition such as $r_C \circ I_D \subseteq I_R \circ r_A$ would represent an acceptable form of refinement. Graphically, it indicates that running the domain interpretation then the refinement yields a result which is valid according to the specification followed by the range interpretation. However, it does not mention anymore the preconditions, in particular p_C. The exact properties of such a definition have yet to be analysed in depth; we do not consider it further in this memoir.

6.4 Some observations using gen-refinement

6.4.1 The distracted composer

Facing a complex requirement to gen-refine a specification (p_A, r_A) (with $p_A \subseteq D_A$ and $r_A \in D_A \leftrightarrow R_A$) using the interpretations $I_D : D_A \leftrightarrow D_C$ and $I_R : R_A \leftrightarrow R_C$, a team leader tackles the problem through decomposition.

He tasks two teams of developers to independently gen-refine (p_A, r_A^1) and (p_A^2, r_A^2), with $r_A^1 : D_A \leftrightarrow M_A$, $p_A^2 \subseteq M_A$ and $r_A^2 : M_A \leftrightarrow R_A$ such that $r_A = r_A^2 \circ r_A^1$.

Unfortunately, being a rather distracted person, he provides I_D and I_R as part of the specification but does not enforce a common intermediate interpretation, nor even a common intermediate concrete type M_C.

After a few days, he therefore receives:

- A refinement (p_C^1, r_C^1), with $p_C^1 \subseteq D_C$ and $r_C^1 : D_C \leftrightarrow M_C^1$,
- A proof of refinement $(p_A, r_A^1) \rightsquigarrow (p_C^1, r_C^1)[I_D, I_M^1]$ with $I_M^1 : M_A \leftrightarrow M_C^1$,
- A refinement (p_C^2, r_C^2), with $p_C^2 \subseteq M_C^2$ and $r_C^2 : M_C^2 \leftrightarrow R_C$
- A proof of refinement $(p_A^2, r_A^2) \rightsquigarrow (p_C^1, r_C^1)[I_M^2, I_R]$ with $I_M^2 : M_A \leftrightarrow M_C^2$.

Of course he very soon discovers that $M_C^1 \neq M_C^2$, that is these refinements cannot be composed, because the composition is not even well-typed.

Fortunately he can benefit from the following extended monotony property, whose interpretation is provided thereafter:

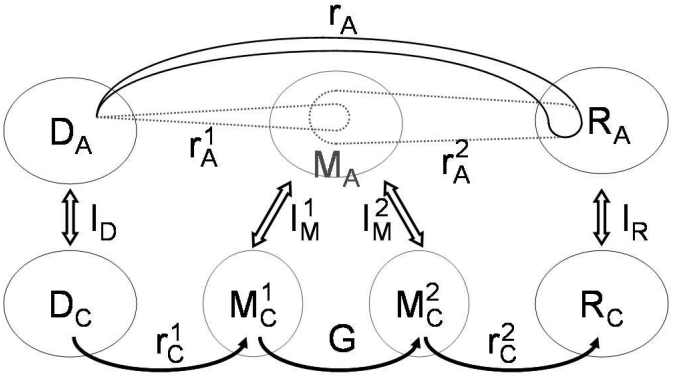
Proposition 6.4.1 (Gen-refinement monotony)

$$(p_A, r_A^1 \rhd \mathbf{dom}(p_A^2 \lhd I_M^2)) \rightsquigarrow (p_C^1, r_C^1)[I_D, I_M^1] \quad \Rightarrow$$

$$(p_A^2, r_A^2) \rightsquigarrow (p_C^2, r_C^2)[I_M^2, I_R] \quad \Rightarrow$$

$$G \circ I_M^1 \subseteq I_M^2 \quad \Rightarrow$$

$$(p_A, r_A^2 \circ r_A^1) \rightsquigarrow (p_C, r_C^2 \circ G \circ r_C^1)[I_D, I_R]$$



The first line is the usual condition about the compatibility of the first refinement with the precondition of the second (cf. Section 6.2.3); we assume that such useless dead ends are avoided by the team leader by an appropriate decomposition. The second line is a standard refinement.

Provided these refinements, the third line indicates that if $G \circ I_M^1 \subseteq I_M^2$ then G is a valid adapter to interconnect r_C^1 and r_C^2 while ensuring refinement of $r_A^2 \circ r_A^1$. Note that this condition has been crafted during the development of this result, that is it is an *ad hoc* sufficient specification for G designed only for the completion of the Coq proof. The fundamental meaning of the condition $G \circ I_M^1 \subseteq I_M^2$ is however very intuitive.

It indeed ensures that G is a partial but sufficient refinement of id modulo the interpretations I_M^1 and I_M^2. That is, considering quietly the difficult situation he is facing, our distracted composer could indeed rewrite $r_A = r_A^2 \circ id \circ r_A^1$.

Having received the refinements r_C^1 and r_C^2, with $r_A^1 \rightsquigarrow r_C^1[I_D, I_M^1]$ and $r_A^2 \rightsquigarrow r_C^2[I_M^2, I_R]$, what he is just missing is a piece of its system, an adaptor G such that $id \rightsquigarrow G[I_M^1, I_M^2]$.

This reading of the conditions "popping out" of the Coq proof for the monotony comfort us w.r.t. the consistency and relevance of the definition of gen-refinement.

More generally, it also emphasises the fact that there are numerous possible refinements for identity, that can appear to be rather exotic for some interpretations. A few typical situations arise:

- The domain interpretation and the range interpretation are equal; the refinement preserves equivalence classes defined by the interpretation.

- The domain interpretation or the range interpretation is the identity; the refinement is a valid translation w.r.t. the interpretations.

There are other interesting examples when considering possible refinements of identity. Let us illustrate that with a specification which is the identity over a set V of values, the domain interpretation being a total relation represented by $I_D = Err \circ Cod$ with $Cod : V \rightarrow W$ an injective function and $Err : W \leftrightarrow W$ any total relation, and our range interpretation being the identity. In this context, a valid total refinement $Corr : W \rightarrow V$ (with a trivial precondition) is such that:

$$Corr \circ Err \circ Cod \subseteq id$$

This is the equation of an error correcting code, *Err* describing the noised channel, but it is viewed here as a refinement problem, and can for example be encoded as such in the B method. That is, we start from a specification which is the identity, and we refine with interpretations that represent the introduction of noise; a valid refinement is then a program able to cancel the effect of the introduced errors.

6.4.2 About the refinement arrow

It can be worthwhile to emphasise the fact that whereas refinement defines a lattice of descriptions of a system, it does not enforce a progress from specifications towards implementations; in other words, the refinement lattice does not capture the V-cycle order.

Refinement is expected to reduce non determinism, but as pointed out in Sections 6.1.2 and 6.2.2, non-determinism is a concept related to the definition of observability – depending upon the chosen definition of extensionality and equality – and its "reduction" may not always be guaranteed.

More fundamentally, there is no refinement-related definition of abstract and concrete. Refinement considering only black boxes, it is not an appropriate tool to distinguish for example between declarative and imperative descriptions. This can be illustrated by the fact that provided very minor modifications, it is possible to prove for the *Maximier* in Section 3.1 that $M_A \sqsubseteq M_C$ but also $M_C \sqsubseteq M_A$ – it is a perfect simulation.

The incentive for progressing towards an implementation is therefore elsewhere; in B it is based on language restrictions, in Coq (when using the strong specification style) on the distinction between types and terms, and in FoCaLize it is the obligation to ultimately exhibit a collection, *i.e.* a specific model (in the logical sense) for a specification admitting several ones.

6.4.3 The refinement paradox

A B-refinement step reduces non-determinism, or more precisely it does not allow a loss of determinism on observable values (outputs). On the contrary with our definition a gen-refinement step can introduce non-determinism using a non-functional interpretation that associates several reifications for an abstract value. It is therefore a natural question to assess the impacts of such interpretations in a B development.

They may indeed have interesting consequences as illustrated by the following example:

Example 6.4.1 (Non-functional interpretation in B) *We consider the very simple specification of a machine with a single operation returning a boolean value:*

MACHINE bool_choice
OPERATIONS $b \leftarrow \text{getbool} \triangleq b := \top \,[]\, b := \bot$

This specification has of course two trivial refinements: one in which the operation getbool *always returns* $\top$*, and one in which it always returns* $\bot$*.*

What is often forgotten is that there are many others valid refinements, such as for example the following one:

MACHINE covert_channel
REFINES bool_choice
VARIABLES secret
INVARIANT $\text{secret} \in \mathbb{N}$
OPERATIONS $b \leftarrow \text{getbool} \triangleq b := (\text{secret}\%2 = 0); \text{secret} := \text{secret}/2$

In this case, similarly to what we have done in Section 5.2.4, we abuse the freedom given by the specification to export information through a form of covert channel.

This is a typical example of the so-called Refinement Paradox: whereas refinement is expected to ensure the preservation of the properties of the specification, it seems here to disregard assumptions about the dependencies for computing b. Yet we cannot really consider in our example that refinement does not preserve the confidentiality of *secret* or transform the dependency graph: there is no explicit specification of these.

Remember that as pointed out in Section 6.1.2, it is not desirable for a refinement to preserve all properties; refinement does not guarantee anything about intensional properties, and dependency is here intensional, describing how a result is computed rather than what it is. It is generally not possible to describe intensional properties in a specification – being able to express intensional properties, but not to ensure their preservation by refinement, would whatever appear to be a design flaw – and therefore nothing can be expected during refinement. Similarly, there is no claim about the meta-properties such as determinism, completeness, and so on; so the fact that they may not be preserved should not be a surprise. Provided these clarifications, the term of paradox is clearly an overstatement, as it just reflects of a poor understanding of specifications and refinements (cf. for example [MM04]).

But our point is elsewhere. Adopting the gen-refinement vision, we can see that the state of the specification *bool_choice* is in **Unit**, but is gen-refined with the interpretation **Unit**×ℕ. We do not only introduce new values during the refinement, but a new dimension, that is a hidden variable. We have mentioned that we consider necessary to have non deterministic specifications, and non functional interpretations.

However, when there are security objectives, for example related to the confidentiality of a data, then the coexistence of non deterministic specifications with non functional interpretations should definitely ring a bell for an independent evaluator, and advocate for a thorough code review or the use of other appropriate tools, such as dependency calculus or dataflow analysis [ABHR99, GM92].

Another interesting learning of gen-refinement is that the Refinement Paradox, often being viewed as specific to methods such as Z or B, can be represented in other formal methods, e.g. in Coq or FoCaLize, using the principle of non functional interpretations. Consider the following specification in FoCaLize:

Example 6.4.2 (Refinement Paradox in FoCaLize)

species $\Omega \triangleq$
  signature $(=) : \mathbf{Self} \to \mathbf{Self} \to \mathbb{B}$
  property $=_{\text{refl}} : \forall\, s : \mathbf{Self},\ s = s$
  property $=_{\text{symm}} : \forall\, s_1, s_2 : \mathbf{Self},\ s_1 = s_2 \Rightarrow s_2 = s_1$
  property $=_{\text{tran}} : \forall\, s_1, s_2, s_2 : \mathbf{Self},\ s_1 = s_2 \Rightarrow s_2 = s_3 \Rightarrow s_1 = s_3$
end

species mybool $\triangleq$
  inherit Ω;
  signature true, false : **Self**;
  property surjective : $\forall\, b$: **Self**, $b =$ true $\vee\ b =$ false;
  property injective : true $\neq$ false;
  signature bool_function : **Self** $\to \mathbb{B}$;
end

The trivial implementation is to use $\mathbb{B}$ *for the representation and to define* bool_function *as one of the four possible functions in* $\mathbb{B} \to \mathbb{B}$*, that is* id, $\neg$, $\top$ *or* $\bot$.

Yet nothing prevents a devious implementation using int *for the representation, odd values representing* $\top$ *and even values* $\bot$ *– the trick is that it is acceptable for* $(=)$ *not to be an observational congruence – and defining* bool_fun *with a rather unexpected behaviour (e.g.* boolfun$(n) = \top$ *iff* n *is a square). And there is no valuable reason to forbid such refinements, which can be very useful is some cases.*

6.5 A few last remarks

Refinement is a very important notion in deductive formal methods, formalising the concept of *compliance* (or correctness) between very different descriptions, such as a specification and an implementation. It is important for refinement to be able to deal with non-determinism, abstraction, and so on to capture standard engineering practices. But the consequences of this flexibility are easily underestimated – Chapter 5 explores the potential consequences of such misunderstandings.

The vision of refinement proposed in this chapter aims at being simple, intuitive and generic, our objective being to shed some lights on concerns related to security properties.

It is applicable to formal methods in which there is no such concept of refinement – at least no explicit one – and for example it indeed helps us to illustrate the refinement paradox in methods such as Coq and FoCaLize. Other observations, such as the distracted composer problem and the refinements of identity, are merely side effects.

The difficulties encountered with refinement when dealing with security properties are well known, and considered in many papers. The following example, discussed at [CD09], is very illustrative:

Example 6.5.1 (Substitutions are not programs) *$s{:}\mathbb{B}$ is a variable which is internal and therefore not visible, whose value is expected to be kept secret. $v{:}\mathbb{B}$ is a variable whose final value is exported and therefore observable. A complex process is specified, in which v has the same value than s at some point in time, before being overloaded:*

$$v{:=}s\,;\, v{:=}\top\,;\, v{:=}\bot\,;\, v{:}{\in}\mathbb{B}$$

This specification is expected to ensure that the final value of v is independent of s. Unfortunately, this is not the case, as the following code is a valid refinement:

$$v{:=}s$$

To deal properly with these situations, [MMM09] for example discusses various adaptations of refinement associated with modified semantics for specifications – such as requiring that $x{:}\in\mathbb{B}$ is different (distinguishable) from $x{:=}\top [\!] x{:=}\bot$.

It is not at all our intent; as mentioned, we are not looking in this chapter for a new definition of refinement or new semantics for substitutions, but for a simpler vision. The previous example is just another illustration of the difficulty to master the semantics of substitutions as predicate transformers: whereas they look like a program, they are not, and in some cases their apparent simplicity is misleading. We prefer to explicit and study the role of significant parameters, such as the interpretations in data-refinement.

It is our feeling that we need to build upon existing theories, rather than trying to define new ones – possibly more complex and without tool support. To some extent, it seems that the combination of several methods and tools can be preferable; as mentioned in Section 5.2.4, confidentiality is likely to be much more easy to ensure by using deductive formal methods and flow control or dependency calculus than with integrated but *ad hoc* theories[3].

Finally, the development of more complex theories should not conceal the real objectives. A proof does not replace a security analysis, but helps to give more confidence in its results. To illustrate our point, imagine a situation in which we are able to provide a mathematical guarantee about the confidentiality of a key used to cipher messages, having applied the most advanced techniques; it would be a pity in such a situation not to notice that the attacker, rather than trying to read the key, can overwrite it to choose its value.

[3]It seems, by the way, that a theory for a dependency calculus compatible with refinement – and in particular data-refinement – is still to be developed.

Chapter 7

The Validation of a Formal Method

The use of formal methods increases the confidence in the correctness of developments, and has the potential to fully eradicate some forms of errors in systems. Yet one may argue about the actual level of confidence obtained, when the method or its implementation are not themselves formally checked. In the field of safety for example, an inconsistent theory or an error in the formal tools can potentially allow for the accidental derivation of invalid results, that is the certification of potentially unsafe systems.

Of course, this is a rather extreme question, and one can legitimately wonder where to stop. What is the interest of a full validation of a formal method and of its associated tools, to develop a system whose correct functioning is dependent upon untrusted and untrustable elements? Furthermore, even if we indeed check the validity of a formal method using another method, what is the gain if the latter is not itself verified?

These remarks can be considered to be relevant in the field of safety. Provided an honest developer and an appropriate review process, the probability of having a realistic failure case undetected during a formal development, resulting of an unwilling exploitation of an accidental flaw in the method or the tool, and not discovered during independent evaluation, is rather low.

But the situation is slightly different when dealing with security – where any flaw can be deliberately exploited by a malicious developer to obfuscate undesirable behaviours of a system while still getting a certification. If such a flaw exists, and is known only by an attacker, the reputation of formal methods is a bias toward obtaining unjustified confidence. Therefore, whereas the concern of the validation of a formal method may appear to be a rather academic one, and the cost of addressing it too important when dealing with safety, we still consider that the validation of a formal method and of its tool is a legitimate objective in the field of security.

B appears to be a popular industrial choice for example for railways transportation systems [BDM98, ED07], but also for security systems, such as smartcards [Jaf07, SL00], firewalls [Bie96], microkernels [HHGB07] and so on. It is therefore a good candidate for addressing our concern: when the prover says that a development is right, who says that the prover is right? To answer this question, one has to check the theory as well as the prover w.r.t. this theory (or, alternatively, to provide a proof checker). We describe in this chapter BiCoq, a deep embedding of the B logic in Coq, with the objective of checking the B theory, but also to implement mechanically checked B tools, in our case a proved prover that can be extracted and used independently of Coq. Note however that this embedding does not address B machines or B proof obligations, and that the representation of the

GSL and of refinement is not fully developped.

The effort associated with this embedding is relatively important. The first complete version of BICOQ amounts to about 10000 lines of COQ, 550 definitions, 750 theorems and proofs. Two other versions have been developed, to explore alternative representations for terms, optimisations of functions, and various improvments of the proof techniques – with a significant reduction in terms of number of lines, definitions and proofs. Considering these figures, we only describe in this chapter this embedding from a high level perspective, focusing mainly on the results obtained through this development. This discussion is completed in the next chapter, in which technical aspects of the embedding(s) are described.

7.1 Deep and shallow embeddings

Embedding, in a proof assistant, consists in mechanizing a *guest* language or logic by encoding its syntax and semantics into a *host* language or logic [M.J88, BGG+92, AP02]. It is now a well-established practice in the academic community, to answer different types of concerns. For example, it is used to study normalisation of terms and influence of evaluation strategies for a programming language, or the consistency for a logic. It is also a way to promote interesting concepts and features from a language to another, or to develop mechanically checked tools to deal with a language.

Notation 7.1.1 (Guest constructs) *When dealing with an embedding considering a guest language and a host language, constructs of the guest language are represented by dotted notations.*

Various approaches can be considered for an embedding, between two extreme cases: *shallow embeddings* describe a translation from the guest language into the host language, whereas *deep embeddings* consider constructs of the guest language as data to be manipulated in the host language.

In a shallow embedding, the encoding is (at least partially) based on a direct translation of the guest language into similar constructs of the host language; in terms of programming languages, a shallow embedding can intuitively be seen as the development of a translation function $[\![\,]\!]$ between two languages, that is a compiler. It is for example typical to translate a redex (a computable expression) of the guest language by a redex of the host language, that can be evaluated.

Example 7.1.1 (Shallow embedding of arithmetic expressions) *We consider a guest language providing machine integers (unsigned, and represented by a byte) and addition, embedded into a host language offering* Peano*'s natural values and addition:*

$$[\![\dot{1}\dot{+}\dot{2}]\!] \rightarrow [\![\dot{1}]\!] + [\![\dot{2}]\!] \rightarrow S(0) + S(S(0)) \rightarrow S(S(S(0)))$$

The redex $\dot{1}\dot{+}\dot{2}$ *is translated into the redex* $S(0)+S(S(0))$ *which is automatically evaluated.*

On the contrary, a *deep embedding* is better intuitively described as the development of a virtual machine: the syntax and the semantics of the guest language are formalised as datatypes, functions or relations of the host language. Considering the previous example, a redex of the guest language is a term of the syntactical datatype encoded in the host language, and the evaluation of the guest language is encoded in the host language for example as a relation or a function.

Example 7.1.2 (Deep embedding of arithmetic expressions) *We consider the guest language of the previous example, embedded in the host language as a description of the syntax of expressions – the addition being a syntactical constructor. The semantics of expressions are provided by an evaluation function in the host language that, for a term of the guest language, returns the normal form in the guest language:*

$$\begin{array}{rcl} L & \triangleq & \dot{0} \mid \dot{1} \mid \dot{2} \mid \ldots \mid 2\dot{5}5 \\ & \mid & \dot{+}(L, L) \\ & \mid & \ldots \end{array}$$

$$\begin{array}{rcl} \mathrm{eval}(l\!:\!L)\!:\!L & \triangleq & \dot{0}\dot{+}X \to X \\ & \mid & \ldots \\ & \mid & \dot{1}\dot{+}\dot{2} \to \dot{3} \\ & \mid & \ldots \end{array}$$

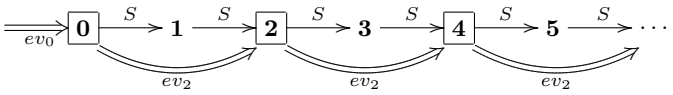
The guest redex $\dot{1}\dot{+}\dot{2}$ *is a term of the syntactical datatype defined in the host language, and is related to the term* $\dot{3}$ *by the evaluation function.*

Applying these concepts to logics is straightforward. A shallow embedding translates statements of the guest logic into statements of the host logic (e.g. $[\![P \dot{\wedge} Q]\!] \to [\![P]\!] \wedge [\![Q]\!]$), which can be proven using the inference rules of the host logic.

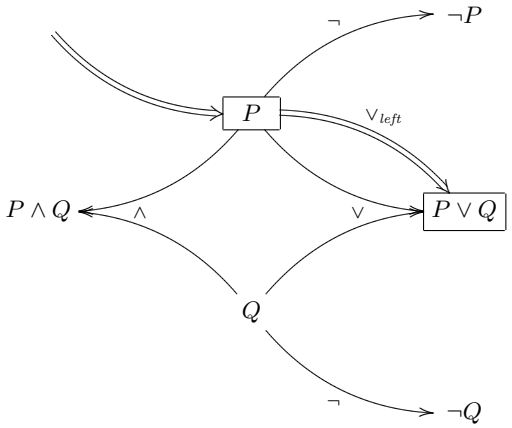
A deep embedding represents the syntactical datatype of statements of the guest logic in the host logic, and encode the inference rules as relations between such statements. Remember the intuitive view presented in Section 3.2, describing $\mathbb{N}$ as a set of terms and *even* as a predicate marking some of them; the definition of *even* describes paths in $\mathbb{N}$ along which the predicate is true:

$$\textbf{Inductive}\ \mathbb{N}\!:\!\textbf{Set} \quad \triangleq \quad 0\!:\!\mathbb{N} \mid S\!:\!\mathbb{N}\to\mathbb{N}$$

$$\textbf{Inductive}\ even\!:\!\mathbb{N}\to\textbf{Prop} \quad \triangleq \quad ev_0\!:\!even\ 0 \mid ev_2\!:\!\forall(n\!:\!\mathbb{N}),\ even\ n \to even\ S(S\ n)$$



Similarly, the deep embedding of a logic defines the set of all statements – the terms – and a provability predicate identifying the subset of statements that are provable. Of course, the rules to construct terms are much richer, and the inference rules of the guest logic describe rather complex trajectories to build provable statements from previous ones:



7.1.1 A quick comparison

Shallow and deep approaches have, of course, pros and cons [WN04]. The standard view is that shallow embeddings require less time to develop, and are appropriate to study programs of the guest language, while deep embeddings are long and complex to develop but permit meta-theoretical analyses of the guest language – such as the influence of the reduction strategy. Indeed, considering the examples of the previous paragraph, with the shallow embedding the translation is straightforward, but only the deep embedding allows for defining and comparing alternative evaluation relations or functions.

Using the vocabulary of Chapter 6, we would also argue that shallow embeddings can address extensional considerations, why deep embeddings are required for intensional considerations. That is, in a shallow embedding a guest redex is translated into a host redex which is evaluated, whereas in a deep embedding we keep the structure of the guest expression and we are able to analyse it (for example by pattern-matching). Let us consider once more the examples of the previous paragraph. Starting from the guest redex $\dot{1}\dot{+}\dot{2}$, the shallow embedding produces 3, not memorising the fact that it is the result of an addition[1]; on the contrary with the deep embedding we preserve the structure of the expression, and can therefore apply for example factorisation or other structural optimisations.

Yet the characteristic that we are concerned with, considering our validation objective, is the accuracy of the mechanisation: a deep embedding allows for an exact representation of the syntax and semantics of the guest logic – or at least to identify and master the deviations – whereas a shallow embedding appears to us as implicitly enforcing a form of interpretation whose validity can be difficult to justify.

Indeed, it is natural in a shallow embedding to use "similar" constructs of the host logic to represent constructs of the guest logic:

Example 7.1.3 (Shallow embedding of disjunction) *We consider the following translation for the disjunction:*

$$[\![\dot{P}\dot{\vee}\dot{Q}]\!] \to [\![\dot{P}]\!] \vee [\![\dot{Q}]\!]$$

This rule may appear reasonable, but it is not always valid: $P \vee \neg P$ *for example is provable in the* B *classical logic but generally not in the* Coq *constructive logic, therefore a too straightforward translation of the guest logic into the host logic can change its nature.*

Much more devious examples can be discussed (for example the representation in a type theory of sets and functions defined in a set theory), showing how easy it would be to loose completeness (provable statements of the guest logic are not provable in the host logic) or correctness (unprovable statements of the guest logic are provable in the host logic), without even knowing it. This is especially relevant if we are questionning the guest logic, and thus cannot trust our intuition about its semantics and the validity of the translation; this chapter will definitely illustrate our point.

Of course, this discussion is only relevant because of our objective: the validation of the B logic as it is (that is as described by the B-Book and implemented in the various tools), and as applied in numerous safety or security projects. This does not appear manageable in a shallow embedding, as we do not know how to identify and manage the possible deviations resulting from a too straightforward translation.

[1] In a system such as Coq for example, once defined a function, it is not really possible to analyse the structure of this function. That is, we cannot for example test whether the function is an **if** between two alternatives, or if it indeed depends upon all its declared parameters. The only possible actions are extensional, that is we can evaluate the function for different values of its parameters – the black box vision.

On the contrary, with slightly different objectives, a shallow embedding may appear as a much more efficient approach. That is, a shallow embedding can be used for example to promote interesting concepts of a language into another one, or to allow for the cooperation of unrelated tools (such as using in cooperation different provers). It is also perfectly relevant to develop alternative tools for the B method, or even richer B theories, that can be applied to new developments.

7.1.2 Why embedding B?

Both deep and shallow embeddings of B in higher-order logics have been described in several papers, with numerous techniques, objectives and results. We just provide a short overview of a few of these works hereafter.

P. Chartier [Cha98] presents a formalisation of the GSL and the concept of B machines in ISABELLE/HOL as a foundational work toward the development of mechanically checked tools for the B method (such as a proof obligations generator). It mixes deep and shallow approaches, and is presented by its author as a semantic embedding. For example, B predicates and expressions are not defined syntactically, but are functions from states to booleans and values, respectively; similarly, the substitutions are represented as before-after operators on states, then related to the presentation given in [Abr96].

J-P. Bodeveix, *M. Filali* and *C. Muñoz* [BFM99] also formalise the GSL, this time in the higher-order logics of COQ and PVS; this work is partially based on a shallow embedding of B in PVS described in [Muñ99], describing the tool PBS which acts as a compiler for B machines producing a PVS file. Again, the approach mixes deep and shallow techniques. Substitutions of the GSL are initially described through the set transformers model of [Abr96] (that is a substitution S is described by $\mathbf{pre}(S)$ and $\mathbf{rel}(S)$); the GSL is not directly a syntactical construction, yet some meta-theoretical considerations are discussed (such as semantics for parallel composition). But one of the objectives of these embeddings is also to promote the B development methodology to COQ and PVS.

With a similar objective, *S. Boulmé* discusses in [Bou07] a formalisation of a *Hoare* logic with an associated refinement in COQ. It defines a higher-order framework enriched with non functional features, in which a specification-driven methodology can be applied thanks to the use of a GSL-like language.

J-P. Bodeveix and *M. Filali* further consider B in [BF02], describing an embedding of B in PVS, but focusing this time on the type-checking mechanisms of the B method (which are called well-formedness verification mechanisms in this memoir); a type-checker is derived from this development.

A shallow embedding of the B logic in PHOX and and COQ is also presented in [RCMP04, CM09], the main objective being the development of a prover. This prover is also used to validate the results of the B-BOOK, a few oversights being identified.

As mentionned, we consider that shallow embeddings are likely to introduce a form of interpretation, but it is fully acceptable when the objectives are to promote some of the B concepts – for example the specification and development language as well as the refinement-based methodology – in other formal methods. It is also justifiable to develop mechanically checked tools with a shallow embedding, noting however that such tools are likely to be usable only for new developments as the correspondance between the B method and its embedded version may not be managed[2].

[2]We do not want to start here a philosophical discussion about the merits and drawbacks of the various logics. From an engineering point of view, the existence of a formal tool without bugs (possibly leading to paradoxes) is much more important than the precise logic enforced by such a tool.

Our main objective is however different. The B method is frequently used for security developments and its validity has to be evaluated; it is important to know, for an entity such as the ANSSI – delivering COMMON CRITERIA certificates in France – what is the level of confidence that one can grant to a system proven using this method, and how to improve this level of confidence. Indeed, an unknown glitch in a formal method can either be a source of accidental weaknesses, or even a mean for a malicious developer to trap a system while providing a formal proof of compliance and getting a certificate. We do not want to promote a variation of B, but to validate B as it is described in the B-BOOK.

With this objective, a reasonable and controlable level of accuracy is required. The translation in a shallow embedding is difficult to define but also to defend against a skeptical independent evaluator. On the contrary a deep embedding makes the justification easier, and clearly separates the host and the guest logics: in our case, for example, excluded middle, provable in the B logic, is not promoted to the COQ logic.

To illustrate the subtle questions arising when dealing with a translation, we have mentionned in the previous paragraph the example of the disjunction between a classical logic and a constructive logic; one can also consider the tricky exercise represented by the encoding of B functions into COQ. In a very shallow approach, one can decide to represent B functions for example as COQ functions; this is however in general inappropriate, as B functions are described as relations (that is sets) and are possibly partial and undecidable. An intermediate approach, adopted for example in [CM09], is to represent B functions as COQ relations satisfying the condition ensuring that each value of the domain has at most one image. In a deep approach, such as in BICOQ, B functions are B relations, that is B subsets of B cartesian product of B sets.

Considering the objective of accuracy and validation, we should mention [BDFF04]. It is indeed based on a deep embedding of the B logic in COQ, to validate the so-called *Base Rules* used by the prover of ATELIERB. This work has allowed for the identification of several problems related to unwanted captures of names, but admits standard B results, whereas they definitively deserve some consideration, as we will see in Section 7.2.7.

Note that the validation of B is, in our view, a sufficient justification for the development of such a deep embedding. Yet it is reasonable to also try to further benefit from it, considering the important investment that it represents. This is why we also use this deep embedding to develop a mechanically checked prover for the B logic, and to prove new results about the B logic; both aspects are presented later in this chapter. For the sake of completion with regard to proven or trusted B tools, we also mention, in addition to [Cha98, BF02], the description in [CK98] of a prover for the B logic as a rewriting system encoding the inference rules.

7.2 Embedding the B logic

7.2.1 Syntax

Given a set of identifiers I, the B logic syntax described in the B-BOOK [Abr96] defines predicates P, expressions E, sets S and variables V as follows:

Definition 7.2.1 (B logic syntax)

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
P & \triangleq & P \wedge P & S & \triangleq & \mathbf{BIG} & E & \triangleq & V \\
 & | & P \Rightarrow P & & | & \wp(S) & & | & S \\
 & | & \neg P & & | & S \times S & & | & E \mapsto E \\
 & | & \forall V \cdot P & & | & \{V|P\} & & | & \mathscr{C}(S) \\
 & | & E = E & & & & & | & [V{:=}E]E \\
 & | & E \in S & V & \triangleq & I & & & \\
 & | & [V{:=}E]P & & | & V, V & & &
\end{array}
$$

Beyond the standard symbols for the logical connectors and the set operators, the following notations are used:

Notation 7.2.1 (B logical notations)

- $[V := E]T$ *represents the application of the assignment* $V := E$ *– a substitution of the* GSL– *to the term* T *(that is a predicate or an expression);*
- V_1, V_2 *is a list of variables;*
- $E_1 \mapsto E_2$ *is a pair of expressions – note that in the* B-BOOK*, pairs are often abusively denoted with a "," instead, that is the notation used for lists of variables;*
- $\mathscr{C}$ *is the choice operator, extracting an unspecified value of a set;*
- $\wp$ *is the powerset operator, that is the set of all subsets of a set;*
- **BIG** *is a (infinite) set constant;*
- $P \Leftrightarrow Q$ *denotes* $P \Rightarrow Q \wedge Q \Rightarrow P$*;*
- $P \vee Q$ *denotes* $\neg P \Rightarrow Q$*,*
- $\exists V \cdot P$ *denotes* $\neg \forall V \cdot \neg P$*;*
- $S \subseteq T$ *denotes* $S \in \wp(T)$*.*

This syntax is further constrained by well-formedness rules[3] in the B-BOOK. For example, comprehension sets, whose syntax is $\{V|P\}$ with V a variable and P a predicate, are required to be of the form $\{V|V \in S \wedge P\}$ with V not free in S, to avoid *Russell*'s paradox.

Our embedding introduces several modifications of the syntax of the B logic to correct minor deficiencies detected during the formalisation, or to optimize the representation. Whereas the accuracy is an important objective, these changes have been considered acceptable; and at least we are able to clearly trace them.

The most important one is the use of *de Bruijn* indexes to tackle the standard problems of clashes and captures of named variables. These difficulties are addressed in the B logic through numerous side conditions associated to definitions and theorems. Our initial attempts to follow the same principles were unfruitful; it was not efficient, or even manageable, in our deep embedding. Using *de Bruijn* indexes, bound variables are replaced by natural values that are pointers to their binder, and these problems do not appear. *Furthermore, the use of a* de Bruijn *representation helps to spot a few oversights in the* B

[3]These rules are called type-checking rules in the B-BOOK, but we prefer to avoid this expression in the context of the embedding, to avoid confusion with the type checking of COQ.

theory. Indeed, trying to reason on two different levels of representation leads to subtle observations about captures for example. The details and techniques associated to the management of *de Bruijn* indexes are described in the next chapter; we just provide here a short example to give the intuition about how they work.

Example 7.2.1 (*de Bruijn* indexes) *The same proposition has two representations:*

Standard representation $\forall x \cdot (x \in \{y \mid y \in \mathbb{N} \wedge y \leq 5\} \Rightarrow x \leq 5)$

de Bruijn *representation* $\underline{\forall}(\underline{0} \in \underline{\{}\underline{0} \in \mathbb{N} \wedge \underline{0} \leq 5\underline{\}} \Rightarrow \underline{0} \leq 5)$

In the de Bruijn *representation, the underlined natural values are indexes,* $\underline{\forall}$ *is the* raw de Bruijn *universal quantifier and* $\underline{\{|\}}$ *the* raw de Bruijn *comprehension quantifier. The index* $\underline{0}$ *represents a variable bound by the closest parent quantifer,* $\underline{1}$ *by the next closest parent quantifier, and so on.*

The use of a *de Bruijn* representation is nothing more than a technical approach to avoid problems related to name management. Being not relevant with the problem of the validation of the B logic, we have dedicated a specific effort to provide tools defining a form of standard representation, masking the awkward details. Except for a few exceptions we do not need to consider further the *de Bruijn* representation in this chapter.

One of the major visible consequences of using a *de Bruijn* representation is that bindings over lists of variables, often used in the B-Book, can not directly be represented in our embedding. Whereas it is possible to parameterise *de Bruijn* binders by a natural value to describe the capture of several indexes, we have not retained this approach which requires to use addition and substraction on indexes (the current version only need successor and predecessor). On the other hand, our embedding has also emphasised the fact that multiple bindings appearing in the B-Book are often invalid according to the rules of the B-Book. Consider the following example:

Example 7.2.2 (Multiple bindings in standard B)

$$\{V_1, V_2 \mid V_1, V_2 \in S_1 \times S_2 \wedge P\}$$

Indeed, well-formedness requires comprehension sets to be of the form $\{V \mid V \in S \wedge P\}$ with V not free in S. When looking at this example, the first occurrence of V_1, V_2 has to denote a list of variables (because of the syntax), but the second occurrence is in fact an abuse of notation for a pair $V_1 \mapsto V_2$. However, this analysis also means that we are not satisfying the constraints set forth by the well-formedness checking. We have not discovered a way to write such syntactically correct terms while respecting the well-formedness rules.

Fortunately, such multiple bindings can be viewed as syntactic sugar. We can indeed consider bindings on a single variable, and interpret for example $\{E \mid E \in S \wedge P\}$, with E an expression, as a friendly notation for $\{V \mid V \in S \wedge Q \wedge P\}$, where Q existentially quantifies the variables appearing free in E over the predicate $V = E$ – this is similar to the notation $\{2x+1 \mid x \in \mathbb{N}\}$ to describe the set of odd natural numbers.

Another adaptation that we have done in BiCoq is to merge B expressions, sets and variables in a single type to enrich the B syntax which is too strict.

First of all, we do not need to consider multiple variables, because their main use is related to multiple bindings – collapsing the syntactical category of variables into a single construct. Furthermore, if the distinction between sets and expressions in the syntax may appear useful, to forbid notations such as $E \in E$, the choice operator is syntactically an

expression. That is, whereas $\mathscr{C}(\wp(S))$ is clearly a subset of S, it is an expression and therefore the predicate $E \in \mathscr{C}(\wp(S))$ is syntactically ill-formed. By merging the three syntactical sorts V, E and S in a single one, we allow for variables representing sets, notations such as $E \in \mathscr{C}(\wp(S))$, and so on. Of course this also creates new syntactical terms such as $E \in E$, but these have no more semantics than the terms such as $S \in S$ allowed in standard B.

Therefore, the syntax of the B logic is embedded (with dotted notations) in Coq as follows[4]:

Definition 7.2.2 (BiCoq syntax) $\mathbb{I}$ *being the set of indexes for the* de Bruijn *notation – that is* $\mathbb{N}$ *– the syntactical sorts* $\mathbb{P}$ *(representing* B *predicates) and* $\mathbb{E}$ *(representing* B *expressions, sets and variables) are defined by:*

$$
\begin{array}{rcl@{\qquad}rcl}
\mathbb{P} & \triangleq & \mathbb{P} \dot{\wedge} \mathbb{P} & \mathbb{E} & \triangleq & \dot{\chi}\mathbb{I} \\
 & | & \mathbb{P} \Rightarrow \mathbb{P} & & | & \mathbb{E} \dot{\mapsto} \mathbb{E} \\
 & | & \dot{\neg}\mathbb{P} & & | & \dot{\mathscr{C}}(\mathbb{E}) \\
 & | & \underline{\forall}\mathbb{P} & & | & \dot{\Omega} \\
 & | & \mathbb{E} \doteq \mathbb{E} & & | & \dot{\wp}(\mathbb{E}) \\
 & | & \mathbb{E} \dot{\in} \mathbb{E} & & | & \mathbb{E} \dot{\times} \mathbb{E} \\
 & & & & | & \{\!|\, \mathbb{E} \,\underline{|}\, \mathbb{P} \,|\!\}
\end{array}
$$

$\dot{\Omega}$ *represents the* B *constant* **BIG** *and* $\dot{\chi}$ *unary* de Bruijn *variables*

Notation 7.2.2 $\mathbb{T} \triangleq \mathbb{P} \cup \mathbb{E}$ *denotes the type of* B *terms.*

Notation 7.2.3 $\dot{\chi}_i$ *denotes the application of constructor* $\dot{\chi}$ *to* $i : \mathbb{I}$*.*

The B binders $\forall V \cdot P$ and $\{V|P\}$ are represented by the constructors $\underline{\forall}$ and $\{\!|\underline{|}|\!\}$, that are raw *de Bruijn* binders: as indicated previously, they are not parameterised by a variable name and only bind a single index (variable).

The constructor $\{\!|\,\underline{|}\,|\!\}$ is further modified to take into account the associated well-formedness rule $\{V \mid V \in S \wedge P\}$ with V not free in S. This constructor therefore has two parameters, the left expression representing S and the right predicate representing P; the non-freeness condition is ensured by considering this constructor as a binder only for its right parameter. Intuitively, this corresponds to the standard notation $\lambda x : T \cdot E$ in simply-typed λ-calculus, where the λ captures x in E but not in T. This modification very efficiently bridges the gap between syntactically correct terms and well-formed ones. Intuitively, rather than forbidding V to appear free in S in $\{V \mid V \in S \wedge P\}$, we just decide that any V appearing free in S is not captured by $\{\mid\}$.

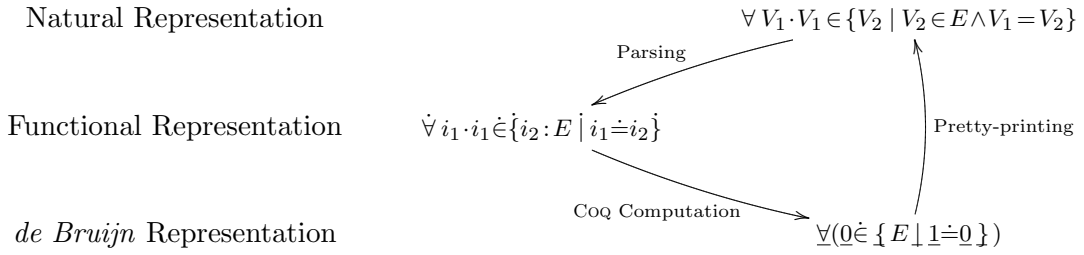
It is clear that *de Bruijn* indexes have technical advantages but are not user-friendly. In particular, the raw *de Bruijn* binders $\underline{\forall}$ and $\{\!|\underline{|}|\!\}$ implicitly captures the first *de Bruijn* index, and capturing a specific variable requires first some transformations. As detailed in Sections 8.1.2 and 8.1.5, to provide a friendly presentation for users and readers, we enrich our syntax with a form of syntactic sugar by defining Coq functions mimicking B natural notation with names.

Notation 7.2.4 (Functional universal quantification) $\dot{\forall} i \cdot P$ *denotes the invocation of the* functional universal quantification*, a function with parameters* $i : \mathbb{I}$ *and* $P : \mathbb{P}$*, computing the* de Bruijn *term representing* $\forall V \cdot P$*,* V *being encoded as the index* i

[4]This is a slightly simplified presentation focusing on aspects relevant for the validation objective.

Notation 7.2.5 (Functional comprehension quantification) $\dot{\{}i{:}S\dot{|}P\dot{\}}$ *denotes the invocation of the* functional comprehension quantification*, a function with parameters* $i:\mathbb{I}$*,* $S:\mathbb{E}$ *and* $P:\mathbb{P}$*, computing the* de Bruijn *term* $\{V \mid V \in S \wedge P\}$*,* V *being encoded as the index* i*.*

To some extent these functions can be seen as part of a parsing scheme:



Note finally that we do not represent the B syntactical construct $[V := E]T$, that denotes the application of a substitution to a term. The assignment is the unique substitution considered in the first chapters of the B-Book dealing with the B logic, whereas it is later completed to the whole Gsl – which in practice defines another syntactical category.

However, we consider that the early introduction of the application of a substitution in the B-Book is justified only by the need to express some of the inference rules, for example the elimination of the universal quantifier. In our view, there is in [Abr96] a confusion between the meta-language substitution used to describe inference rules, which is denoted $[V \backslash E]$ in this memoir, and the Gsl assignment used as a form of programming language in specifications or implementations in the B method.

We therefore define another Coq function to compute the *de Bruijn* term obtained by the meta-language substitution:

Notation 7.2.6 (Meta-language substitution) $[i \backslash E]T$ *denotes the invocation of the* meta-language substitution function *with parameters* $i:\mathbb{I}$*,* $E:\mathbb{E}$ *and* $T:\mathbb{T}$*, computing the* de Bruijn *term obtained by replacing any occurrence of the free variable* i *by* E *in* T*.*

Provided this function, we have a way to encode the inference rules, without having to deal too soon with the Gsl, and without having to introduce an additional syntactical constructor in $\mathbb{T}$.

7.2.2 Inference rules

Having formalised the syntax of the B logic as a datatype, the next step is to encode the B inference rules as the constructors of an inductive *provability* predicate defining a familily of dependent types. As indicated in Section 7.1, we are describing paths along which the terms of our logic are "true".

We denote $\Gamma \dot{\vdash} P$ the Coq type of B proofs of the predicate P under the assumptions Γ (a.k.a. proof environment, a finite set of predicates); if it is inhabited then P is provable assuming Γ. It is important to understand the difference between the two following propositions:

- $\neg(\Gamma \dot{\vdash} P)$ means that there is no B proof of P assuming Γ.
- $\Gamma \dot{\vdash} \dot{\neg} P$ means that there is a B proof of $\dot{\neg} P$ assuming Γ.

More generally, the deep embedding approach ensures a clear separation between the two logics, and the reader should take care to check whether we are considering Coq operators

or embedded B operators (denoted with dotted notations) – this is especially relevant for Section 7.2.8 comparing these operators. Note also that Coq operators, such as the negation, do not apply to B predicates but to B sequents.

Thanks to the functions mimicking B natural notation with names, and a few additional functions (for example $i \leftthreetimes T$ is the Coq function representing the B condition $V \backslash T$, checking that V does not appear free in T) these constructors look very much like the standard B rules. The translation is straightforward, merely a syntactical one, limiting the risk of error – and more convincing for an independent evaluator.

The first serie of rules is self explanatory[5]:

Definition 7.2.3 (Provability predicate, part 1) *In the following table, the standard* B *inference rules are on the left side, and their representation in* BiCoq *on the right side.*

Reading for example the second rule, the standard B *version claims that if the predicate* P *appears in* Γ *(a side condition, as it is not a sequent of the* B *logic) then* $\Gamma \vdash P$*; its translation is a constructor, whose name is left implicit here, that provided a* Coq *proof of* $P \in \Gamma$ *(where* $\in$ *is a predicate defined in* Coq*) builds a term whose type is* $\Gamma \dot{\vdash} P$*.*

$\dfrac{}{P \vdash P}$	*is encoded by*	*nothing (derivable rule)*
$\dfrac{P \text{ appears in } \Gamma}{\Gamma \vdash P}$		$P \in \Gamma \to \Gamma \dot{\vdash} P$
$\dfrac{\Gamma' \text{ includes } \Gamma \quad \Gamma \vdash P}{\Gamma' \vdash P}$		$\Gamma \subseteq \Gamma' \to \Gamma \dot{\vdash} P \to \Gamma' \dot{\vdash} P$
$\dfrac{\Gamma \vdash P \quad \Gamma, P \vdash Q}{\Gamma \vdash Q}$		*nothing (derivable rule)*
$\dfrac{\Gamma \vdash P \Rightarrow Q}{\Gamma, P \vdash Q}$		$\Gamma \dot{\vdash} P \dot{\Rightarrow} Q \to \Gamma, P \dot{\vdash} Q$
$\dfrac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}$		$\Gamma, P \dot{\vdash} Q \to \Gamma \dot{\vdash} P \dot{\Rightarrow} Q$
$\dfrac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}$		$\Gamma \dot{\vdash} P \to \Gamma \dot{\vdash} Q \to \Gamma \dot{\vdash} P \dot{\wedge} Q$
$\dfrac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P}$		$\Gamma \dot{\vdash} P \dot{\wedge} Q \to \Gamma \dot{\vdash} P$
$\dfrac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q}$		$\Gamma \dot{\vdash} P \dot{\wedge} Q \to \Gamma \dot{\vdash} Q$
$\dfrac{\Gamma, Q \vdash P \quad \Gamma, Q \vdash \neg P}{\Gamma \vdash \neg Q}$		$\Gamma, Q \dot{\vdash} P \to \Gamma, Q \dot{\vdash} \dot{\neg} P \to \Gamma \dot{\vdash} \dot{\neg} Q$
$\dfrac{\Gamma, \neg Q \vdash P \quad \Gamma, \neg Q \vdash \neg P}{\Gamma \vdash Q}$		$\Gamma, \dot{\neg} Q \dot{\vdash} P \to \Gamma, \dot{\neg} Q \dot{\vdash} \dot{\neg} P \to \Gamma \dot{\vdash} Q$
$\dfrac{}{\Gamma \vdash E = E}$		$\Gamma \dot{\vdash} E \dot{=} E$

The two rules for the universal quantifier are encoded as follows:

[5] By abuse of notation, we describe side conditions as antecedents.

Definition 7.2.4 (Provability predicate, part 2)

$$\frac{\Gamma\vdash P \quad V\backslash\Gamma}{\Gamma\vdash\forall\, V\cdot P} \quad \textit{is encoded by} \quad i\dot{\setminus}\Gamma \to \Gamma\dot{\vdash} P \to \Gamma\dot{\vdash}\dot{\forall}\, i\cdot P$$

$$\frac{\Gamma\vdash\forall\, V\cdot P}{\Gamma\vdash[V:=E]P} \qquad\qquad \Gamma\dot{\vdash}\dot{\forall} i\cdot P \to \Gamma\dot{\vdash}[i\backslash E]P$$

Of course, we use here the functional representation, which is close enough to the standard B notation to ensure a straighforward (and unquestionable) translation. Note however that these constructors of the provability predicate do not encode rules but schemas. The consequences $\Gamma\dot{\vdash}\dot{\forall} i\cdot P$ and $\Gamma\dot{\vdash}[i\backslash E]P$ of these two rules are not syntactical terms that can be matched, but denote the applications of a function, for which unification is not immediate. This is discussed in the next paragraph.

The next serie of rules applies the same principles, except for the fact that we also decompose complex rules into several simpler ones to ease their use:

Definition 7.2.5 (Provability predicate, part 3)

$$\frac{V\backslash S}{\begin{array}{c}\Gamma\vdash E\in\{V\mid V\in S\wedge P\}\\ \Leftrightarrow E\in S\wedge[V:=E]P\end{array}} \quad \textit{is encoded by} \quad \begin{array}{l}\Gamma\dot{\vdash} E\dot{\in}S\to\Gamma\dot{\vdash}[i\backslash E]P\to\Gamma\dot{\vdash}E\dot{\in}\dot{\{}i{:}S\,\dot{|}\,P\dot{\}}\\ \Gamma\dot{\vdash} E\dot{\in}\dot{\{}i{:}S\,\dot{|}\,P\dot{\}}\to\Gamma\dot{\vdash}E\dot{\in}S\\ \Gamma\dot{\vdash} E\dot{\in}\dot{\{}i{:}S\,\dot{|}\,P\dot{\}}\to\Gamma\dot{\vdash}[i\backslash E]P\end{array}$$

$$\frac{\Gamma\vdash E=F \quad \Gamma\vdash[V:=E]P}{\Gamma\vdash[V:=F]P} \qquad \Gamma\dot{\vdash} E\doteq F\to\Gamma\dot{\vdash}[i\backslash E]P\to\Gamma\dot{\vdash}[i\backslash F]P$$

$$\frac{V\backslash S}{\Gamma\vdash\exists\, V\cdot V\in S\Rightarrow\mathscr{C}(S)\in S} \qquad i\dot{\setminus}E\to\Gamma\dot{\vdash}\dot{\exists}\, i\cdot i\dot{\in}E\to\Gamma\dot{\vdash}\dot{\mathscr{C}}(E)\dot{\in}E$$

$$\frac{V\backslash S,T}{\begin{array}{c}\Gamma\vdash S\in\wp(T)\\ \Leftrightarrow\forall\, V\cdot V\in S\Rightarrow V\in T\end{array}} \qquad \begin{array}{l}i\dot{\setminus}E_1\dot{\times}E_2\to\Gamma\dot{\vdash}\dot{\forall}\, i\cdot i\dot{\in}E_1\dot{\Rightarrow}i\dot{\in}E_2\to\Gamma\dot{\vdash}E_1\dot{\in}\dot{\wp}(E_2)\\ i\dot{\setminus}E_1\dot{\times}E_2\to\Gamma\dot{\vdash}E_1\dot{\in}\dot{\wp}(E_2)\to\Gamma\dot{\vdash}\dot{\forall}\, i\cdot i\dot{\in}E_1\dot{\Rightarrow}i\dot{\in}E_2\end{array}$$

$$\frac{V\backslash S,T}{\begin{array}{c}\Gamma\vdash\forall\, V\cdot V\in S\Rightarrow V\in T\\ \wedge\forall\, V\cdot V\in T\Rightarrow V\in S\\ \Leftrightarrow S=T\end{array}} \qquad \Gamma\dot{\vdash} E_1\dot{\in}\dot{\wp}(E_2)\to\Gamma\dot{\vdash}E_2\dot{\in}\dot{\wp}(E_1)\to\Gamma\dot{\vdash}E_1\doteq E_2$$

The next B inference rule claims that the constant set **BIG** is infinite; this predicate is in fact defined in [Abr96] through a fixpoint extracting value from its parameter using $\mathscr{C}$. As it is not reasonable to unfold this complex definition to be able to express this rule, we once more deviate from the B logic by introducing a new syntactical constructor $\omega{:}\mathbb{N}\to\mathbb{E}$, representing an enumerable quantity of values belonging to **BIG**:

Definition 7.2.6 (Provability predicate, part 3)

$$\frac{}{\vdash\mathbf{infinite}(\mathbf{BIG})} \quad \textit{is encoded by} \quad \begin{array}{l}\Gamma\dot{\vdash}\omega_i\dot{\in}\Omega\\ i\neq j\to\Gamma\dot{\vdash}\dot{\neg}\,\omega_i\doteq\omega_j\end{array}$$

Finally, the last B inference rule, related to the semantics of the cartesian product, deserves specific analysis and is discussed in Section 7.2.7.

7.2.3 Raw inference rules

As indicated, we use in the embedding of the inference rules of the B logic the so-called functional representation, that is considering for example the universal quantifier we favour the expression $\dot{\forall}\, i \cdot P$, denoting the invocation of a function, instead of $\underline{\forall} P$ which is a term of our syntax. It is also true for the rules dealing with comprehension sets, and for the rules in which the meta-language substitution appears.

However, such rules being applied backward during proof, one has to note that these rules are in fact encoded as schemas. Indeed, if we have to prove a universally quantified predicate, we can decide to apply the first rule for universal quantification[6]. However, this predicate is not always of the form $\dot{\forall}\, i \cdot P$; it can be in the raw form $\underline{\forall} P$, not matching our functional pattern.

Note that in most of the cases, working within Coq that does not automatically reduces β-redexes, we often start with terms in the functional representation, and we preserve this representation by applying rules expressed in the same paradigm. In a prover using the same techniques, this would correspond to using the *de Bruijn* representation only for internal computations, whereas unification is done directly on the natural representation – that is before parsing or after pretty-printing.

Yet if we have to deal with a raw universal binder, there are several possible approaches in BiCoq. Indeed, we define in our development a functional application, denoted @, which is best described here as the reverse of the functional universal quantification:

Notation 7.2.7 (Functional application) *P@E denotes the invocation of the* functional application*, a function with parameters $P : \mathbb{P}$ and $E : \mathbb{E}$. This function is partial, applying only to universally quantified predicates (this is encoded in* Coq *by an additional proof parameter which is left implicit here); P@E computes the* de Bruijn *term representing the instantiation of the quantified predicate P at E.*

Proposition 7.2.1 (Functional application property) *The functional application is such that:*

$$(\dot{\forall}\, i \cdot P)@E \quad = \quad [i \backslash E]P$$

Proposition 7.2.2 (Functional abstraction inversion) *For a raw universally quantified term, a representation using the functional universal quantification can be obtained as follows:*

$$i \backslash \underline{\forall} P \quad \Rightarrow \quad \underline{\forall} P = \dot{\forall}\, i \cdot ((\underline{\forall} P)@\dot{\chi}_i)$$

The point is that the functional application works directly on the internal *de Bruijn* representation of universally quantified terms, without having to transform it into the functional representation. That is, whereas we have embedded the B inference rules using a straightforward translation relying on the functional notation to ensure traceability and accuracy, we also prove the equivalent raw versions in the internal *de Bruijn* representation. For example the raw version of the universal quantifier rules is as follows, noting that the first of these rules does not conclude with the invocation of a function:

[6] The situation is similar for the second rule for universal quantification, that can only be applied provided the term is of the form $[i \backslash E]P$. For this rule however, our embedding does not create additional complexity, as the problem is the same in the B logic: what we generally have is a term of the form P, and given an index i we have to build a term P' such that $P = [i \backslash E]P'$ (pattern form).

Proposition 7.2.3 (Raw universal quantifier rules)

$$i \backslash \Gamma, \underline{\forall} P \quad \Rightarrow \quad \Gamma \dot{\vdash} (\underline{\forall} P) @ \dot{\chi}_i \quad \Rightarrow \quad \Gamma \dot{\vdash} \underline{\forall} P$$

$$\Gamma \dot{\vdash} \underline{\forall} P \quad \Rightarrow \quad \Gamma \dot{\vdash} (\underline{\forall} P) @ E$$

This describes the different forms of terms and rules that can be manipulated in BiCoq – and the different associated strategies. In practice, the proofs of standard results of the B logic are easy to reproduce using only the functional representation and the encoded inference rules – in other words, it is possible to exploit BiCoq without even knowing that it is based on a *de Bruijn* representation. The other variants can be used for technical results, such as those described in Section 7.5.2 for example, but also provides a form of metatheory about our *de Bruijn* representation.

Indeed, the definition of the functional quantifications and the functional application is required to deal with all types of proofs in BiCoq, but they also emphasise that the meta-language substitution which is used here is not primitive. *As indicated by the application property herebefore, rather than defining the meta-language substitution function, we can emulate it by applying the functional universal quantification followed by the functional application.* This is further discussed in Section 8.1.6.

7.2.4 A remark about notations

We would like to mention the benefits derived from the use of appropriate notations. It is indeed possible, in Coq, to define complex notations using Utf-8 symbols. It allows for much shorter, but also much clearer definitions and statements.

This is especially important for BiCoq. In such an embedding, there is a great number of concepts to deal with, and any help to distinguish between host constructs and guest constructs is welcome.

Furthermore, with the aim of validating the B logic, the improved readability by B experts not comfortable with Coq notations or independent evaluators resulting from using the standard B presentation is important.

This is illustrated by the following comparison of the scan of a theorem as it appears in the B-Book, and the snapshot of its definition in BiCoq sources:

$$\exists x \cdot (P \Rightarrow Q) \Leftrightarrow (\forall x \cdot P \Rightarrow \exists x \cdot Q) \qquad \text{Theorem 1.3.4}$$

```
Theorem bbt_1_3_4 : forall (i:I)(p q:Π), ⊢∃(i•p⇒q)⇔(∀(i•p)⇒∃(i•q)).
```

7.2.5 Checking standard B results

Having embedded the syntax of B terms and the inference rules of the B logic, we are able to formally check the validity of B theorems and proofs. This effort has been conducted for all propositional calculus results, most of the predicate calculus results, but only some of the results dealing with more advanced set constructs.

Nearly all the theorems and proofs are validated through our review; a few significant exceptions are however identified, and discussed in the next paragraph.

To assist the proof construction, we have developed dedicated proof tactics for the B logic using the Ltac language [Del00] of Coq. Beyond the administrative tactics (e.g.

providing a fresh variable), we also provide in BiCoq more advanced tactics, for example the propositional calculus procedure described in [Abr96].

As indicated in Section 7.2.3, for some basic theorems, we also provide the so-called raw version using the internal *de Bruijn* representation and dedicated functions such as the application. These results are of course technical, but as pointed out they have some practical and metatheoretical interests. Note for example that they change the very nature of the side conditions; syntactical (administrative) side conditions disappear, and semantically relevant ones are transformed. To illustrate our point, let us reconsider the various versions of the universal quantification rules:

Example 7.2.3 (Comparing standard and raw inference rules)

$$i \dot{\leftharpoonup} \Gamma \to \Gamma \dot{\vdash} P \to \Gamma \dot{\vdash} \dot{\forall}\, i \cdot P \qquad\qquad i \dot{\leftharpoonup} \Gamma \to i \dot{\leftharpoonup} \underline{\forall} P \to \Gamma \dot{\vdash} (\underline{\forall} P)@i \to \Gamma \dot{\vdash} (\underline{\forall} P)$$

$$\Gamma \dot{\vdash} \dot{\forall} i \cdot P \to \Gamma \dot{\vdash} [i \backslash E] P \qquad\qquad \Gamma \dot{\vdash} \underline{\forall} P \to \Gamma \dot{\vdash} (\underline{\forall} P)@E$$

The standard version of the first rule for example can be applied on a term of the form $\dot{\forall}\, i \cdot P$ provided i does not appear free in Γ – that is, in practice, the rule can always be applied, but an alpha-renaming can be required first. On the contrary the raw version can be applied to any universally quantified term; the index i is not imposed by the term to which the rule is applied, but chosen (fresh).

7.2.6 About B sets constructs

We have indicated that only a few of the results of the B-Book have been checked so far for set constructs. It is indeed easy to deal with simple results, such as the properties of inclusion. However, other B constructs are much more difficult to capture.

For example, the union of two sets is defined in [Abr96] as follows:

Definition 7.2.7 (B standard union)

$$S_1 \cup S_2 \quad \triangleq \quad \{V \mid V \in S \wedge V \in S_1 \vee V \in S_2\}$$

There is a lot to say about this definition. The first immediate observation is that the set S appears only on the right side of the equality; it is a form of implicit parameter. In the B-Book, this set is the *superset* of S_1 and S_2, that is intuitively the biggest set that contains both S_1 and S_2 – this is not the type theory (disjoint) union, but the B union, which is only valid for two subsets of the same set. The expression $S_1 \cup S_2$ does not pass well-formedness checking if there is no such set.

Of course, this is not very convenient in BiCoq. There are two possible approaches:

- computing the superset S from S_1 and S_2, by analysis of the syntactical structure of S_1 and S_2, with rules such as for example $\mathbf{super}(\{V \mid V \in S \wedge P\}) = \mathbf{super}(S)$;
- embedding the union as an operation with three parameters, that is $S_1 \cup_S S_2$.

Interestingly, the first approach is probably closer to the spirit of the B-Book definition, but it is however inapplicable when dealing for example with abstract sets that can appear in the **SETS** clause of a B machine – being abstract, such sets cannot be represented in the syntax of the B logic and are in fact a form of meta-variable which is expected to be refined at a later stage into a concrete set. A possible alternative would be to enrich the B syntax with such abstract sets, using a dedicated space of names – as it is suggested by some of the well-formedness rules of the B-Book introducing the notation $\mathbf{given}(S)$ to deal with

an abstract set S. Yet we favour the second approach, expecting some enhanced parsing scheme to produce the appropriate set when analysing a standard B representation; note that we have not developped such a parsing.

A second observation, discovered thanks to the use of a *de Bruijn* representation, is that the standard B definition of union does not forbid the variable V to appear free in S_1 or S_2. That is, there is a risk of capture with unpredictable consequences on the semantics of the unions of two sets. This is something which is forbidden in our definition of union:

Definition 7.2.8 (BiCoq union) *The embedding of the* B *union is defined as follows, with $\mathcal{F}$ denoting a function computing a fresh variable with respect to its parameters*[7]*:*

$$S_1 \dot\cup_S S_2 \quad \triangleq \quad \mathbf{let}\, i := \mathcal{F}(S_1, S_2)\, \mathbf{in}\, \{i : S \mid i \in S_1 \vee i \in S_2\}$$

Note that this definition is valid also when S_1 and S_2 have no common superset; it can be modified as follows to reduce the consequences of this problem by ensuring that the union is empty if the two sets are not compatible:

$$S_1 \dot\cup_S S_2 \quad \triangleq \quad \mathbf{let}\, i := \mathcal{F}(S_1, S_2)\, \mathbf{in}\, \{i : S \mid S_1 \subseteq S \wedge S_2 \subseteq S \wedge (i \in S_1 \vee i \in S_2)\}$$

For situations in which the manipulated sets are concrete (fully defined using the B *syntax), we can define the* superset function *computing the superset and consider a version closer to the* B *union, defined by:*

$$S_1 \dot\cup S_2 \quad \triangleq \quad S_1 \dot\cup_{\mathbf{super}(S_1)} S_2$$

Similar observations can be made about other B constructs, such as the intersection, the difference, the singleton, or the λ-abstraction – that is in B a function which is represented by a set of pairs. We therefore consider that these definitions cannot be embedded accurately; instead of pursuing an important effort on these concepts, it would be more efficient to review and clarify some of the standard B definitions.

7.2.7 Validity of the B logic

As mentionned in Section 7.2.1, the formalisation of the B syntax, rules and constructions in Coq underlines some glitches, mainly related to the confusion between list of variables and ordered pairs of expressions, both being abusively denoted with the symbol "," in the *B-Book*. Section 7.2.6 also reveals inadequate definitions, using implicit parameters or allowing for accidental captures of variables. But the formalisation of the B inference rules is much more instructive; whereas it does not reveal any inconsistency, various problems have been pinpointed.

For example, some of the inference rules, the so-called *Axiom Rule* ($P \vdash P$) and the *Modus Ponens*, are provable using the other rules. Therefore the embedding formalises such rules as theorems rather than as constructors of the provability predicate.

Another deviation between the B inference rules and our formalisation is related to the correction of the definition of the cartesian product. The standard B inference rule is:

Definition 7.2.9 (B inference rule for cartesian product)

$$\vdash (E \mapsto F) \in (S \times T) \Leftrightarrow (E \in S) \wedge (F \in T)$$

[7] Working directly with raw *de Bruijn* terms, a simpler definition of union, not using $\mathcal{F}$, is possible.

However, our analysis shows that with this rule, the following results, presented in the B-Book as theorems, are in fact not provable:

$$\vdash E_1 \mapsto F_1 = E_2 \mapsto F_2 \Rightarrow E_1 = E_2$$

$$\vdash E_1 \mapsto F_1 = E_2 \mapsto F_2 \Rightarrow F_1 = F_2$$

$$\vdash S_1 \subseteq S_2 \wedge T_1 \subseteq T_2 \Rightarrow S_1 \times T_1 \subseteq S_2 \times T_2$$

The first two theorems are provided in [Abr96] with a proof that is in fact incorrect, due to a confusion between lists of variables and pairs of expressions; the third theorem is considered trivial and left as an exercise to the readers. To our knowledge, these problems were not known by the B community – whereas B implementations appear to correct the flaws, consciously or not, as the theorems can indeed be checked with the provers.

The problem with the standard B inference rule for cartesian product is that it only expresses a constraint for pairs belonging to a product. Yet it does not say that a product only contains pairs, or that two pairs with different components are different. In our view, it is unlikely that a shallow embedding approach would identify this problem, as it appears quite natural to translate the B cartesian product as the Coq product.

For the correction, we replace the inference rule by two constructors of the provability predicate in BiCoq, one dealing with the injectivity (two equal pairs have equal constituents), and the other with the surjectivity (cartesian products only contain pairs):

Definition 7.2.10 (Provability predicate, part 4)

$$\Gamma \dot{\vdash} E_1 \dot{\mapsto} E_2 \doteq E_3 \dot{\mapsto} E_4 \rightarrow \Gamma \dot{\vdash} E_1 \doteq E_3 \dot{\wedge} \Gamma \dot{\vdash} E_2 \doteq E_4$$

$$i_1, i_2 \dot{\leftarrow} E \dot{\in} (E_1 \dot{\times} E_2) \rightarrow i_1 \neq i_2 \rightarrow \Gamma \dot{\vdash} \dot{\exists}\, i_1 \cdot i_1 \dot{\in} E_1 \dot{\wedge} \dot{\exists}\, i_2 \cdot i_2 \dot{\in} E_2 \dot{\wedge} E \doteq i_1 \dot{\mapsto} i_2 \dot{\Leftrightarrow} E \dot{\in} (E_1 \dot{\times} E_2)$$

These modified rules can indeed be used to prove the expected results of the B logic.

7.2.8 Shallow embeddings revisited

Deep embeddings such as this one ensure a clear separation of the guest logic and the host logic, allowing for a study of their relations as illustrated here with the B operators on the left side and the Coq operators on the right side:

Proposition 7.2.4 (Relations between Coq and B logical operators)

$$\begin{array}{lcl}
\Gamma \dot{\vdash} P_1 \dot{\wedge} P_2 & \Leftrightarrow & (\Gamma \dot{\vdash} P_1) \wedge (\Gamma \dot{\vdash} P_2) \\
\Gamma \dot{\vdash} \dot{\forall}\, i \cdot P & \Leftrightarrow & \forall\, (E : \mathbb{E}),\ \Gamma \dot{\vdash} [i \backslash E] P \\
\Gamma \dot{\vdash} P_1 \dot{\Rightarrow} P_2 & \Rightarrow & \Gamma \dot{\vdash} P_1 \Rightarrow \Gamma \dot{\vdash} P_2 \\
\Gamma \dot{\vdash} P_1 \dot{\vee} P_2 & \Leftarrow & (\Gamma \dot{\vdash} P_1) \vee (\Gamma \dot{\vdash} P_2) \\
\Gamma \dot{\vdash} \dot{\exists}\, i \cdot P & \Leftarrow & \exists\, (E : \mathbb{E}),\ \Gamma \dot{\vdash} [i \backslash E] P \\
\Gamma \dot{\vdash} E_1 \doteq E_2 & \Leftarrow & E_1 = E_2
\end{array}$$

Of course, the interesting results are those that are not equivalences. The example of disjunction is very significant w.r.t. the difference between the classical logic of B and the constructive logic of Coq.

For example, we have mentionned that the *Excluded Middle* is provable in B. That is, it is always possible to provide a proof of $\dot{\vdash} P \dot{\vee} \dot{\neg} P$; should the disjunction being directly translated in Coq we would obtain $(\dot{\vdash} P) \vee (\dot{\vdash} \dot{\neg} P)$ for any P, that is a proof that the B logic is complete (i.e. such that it is always possible to prove or to refute any proposition), which of course is not the case.

Such results can be seen as providing a formal justification (or refutation) for the correctness and the completeness of the translation in a shallow embedding; one may wonder whether it would be possible to automatically derive a shallow embedding from a deep embedding, provided such results.

7.2.9 About the consistency of the B logic

The correspondances between B and Coq logical constructs detailed in the previous paragraph do not provide any clue about the negation. This is indeed still a work to complete.

One of the direction is quickly dealt with:

$$\neg(\Gamma \dot{\vdash} P) \quad \not\Rightarrow \quad \Gamma \dot{\vdash} \dot{\neg} P$$

Indeed, this would again be a proof of the completeness of the B logic: the fact that we know that there is no proof of $\Gamma \dot{\vdash} P$ does not ensure that there is a proof of $\Gamma \dot{\vdash} \dot{\neg} P$.

The other direction is much more interesting:

$$\Gamma \dot{\vdash} \dot{\neg} P \quad \stackrel{?}{\Rightarrow} \quad \neg(\Gamma \dot{\vdash} P)$$

This is one of the possible expressions of the consistency of the B logic: if we know that there is a proof of $\Gamma \dot{\vdash} \dot{\neg} P$, can we deduce that there is no proof of $\Gamma \dot{\vdash} P$? It would be the case if the B logic is consistent (provided the proof environment Γ is consistent as well).

Alternative formulations of the consistency of the B logic have been considered:

Definition 7.2.11 (Possible definitions for the consistency of the B logic)

$$\mathbf{B\_cons}_0 \triangleq \forall\, (P{:}\mathbb{P}),\ \neg(\dot{\vdash} P \wedge \dot{\vdash} \dot{\neg} P)$$

$$\mathbf{B\_cons}_1 \triangleq \{P{:}\mathbb{P} \mid \neg(\dot{\vdash} P)\}$$

$$\mathbf{B\_cons}_2 \triangleq \neg(\forall\, (P{:}\mathbb{P}),\ \dot{\vdash} P)$$

$$\mathbf{B\_cons}_3 \triangleq \neg\{P{:}\mathbb{P} \mid (\dot{\vdash} P) \wedge (\dot{\vdash} \dot{\neg} P)\}$$

$$\mathbf{B\_cons}_4 \triangleq \neg(\dot{\vdash} \dot{\neg}\dot{\Omega} \dot{=} \dot{\Omega})$$

$\mathbf{B\_cons}_0$ means that there is no predicate which is provable and refutable, $\mathbf{B\_cons}_1$ that there exists a predicate which is not provable, $\mathbf{B\_cons}_2$ that it is false that all predicates are provable, $\mathbf{B\_cons}_3$ that it is false that there exists a predicate which is provable and refutable, and $\mathbf{B\_cons}_4$ that $\bot$ cannot be proved in B. Fortunately, we prove:

Proposition 7.2.5

$$\mathbf{B\_cons}_0 \Rightarrow \mathbf{B\_cons}_1$$

$$\mathbf{B\_cons}_1 \Rightarrow \mathbf{B\_cons}_2$$

$$\mathbf{B\_cons}_2 \Rightarrow \mathbf{B\_cons}_3$$

$$\mathbf{B\_cons}_3 \Rightarrow \mathbf{B\_cons}_0$$

$$\mathbf{B\_cons}_0 \Rightarrow \mathbf{B\_cons}_4$$

$$\mathbf{B\_cons}_4 \Rightarrow \mathbf{B\_cons}_1$$

That is, all these definitions are equivalent in the COQ constructive logic.

The proof of the consistency of the B logic itself is still to be done, but our feeling is that most of the required tools are already available in BICOQ. Indeed, we provide a strong induction principle on B proofs, which is expected to be usable for example to show that any proof of $\dot{\vdash} \dot{\neg}\dot{\Omega} \dot{=} \dot{\Omega}$ would have to rely on a smaller proof of the same sequent. Note that the definition of the provability predicate is a form of model of the B logic, and the objective of the consistency proof would be to show that at least one of the dependent type $\Gamma \dot{\vdash} P$ is empty.

7.3 Embedding the GSL

7.3.1 Syntax

The syntax of the GSL is defined in the B-BOOK as follows:

Definition 7.3.1 (GSL **syntax**)

$$\begin{array}{rcl} G & \triangleq & \mathbf{skip} \\ & | & V := E \\ & | & P \mid G \\ & | & P \Longrightarrow G \\ & | & G [] G \\ & | & @V \cdot G \\ & | & G; G \end{array}$$

Notation 7.3.1 (B **notations**)

- **skip** *is the substitution doing nothing;*
- $V := E$ *is the elementary substitution (not to be confused with the meta-language substitution, cf. Section 7.2.1);*
- $P \mid G$ *is the precondition;*
- $P \Longrightarrow G$ *is the guard;*
- $G [] G$ *is the bounded choice;*
- $@V \cdot G$ *is the unbounded choice;*

- $G\,;\,G$ *is the sequence;*

The semantics are described in the next paragraph.

In addition, the parallel substitution is presented as a generalisation of the elementary substitution. It is described directly by its semantics of predicate transformer, that is:

Definition 7.3.2 (B parallel substitution)

$$[V, LV := E, LE]P \Leftrightarrow [LV' := LE][V := E][LV := LV']P$$

Where V is a variable, LV a list of variables, E an expression, LE a list of expressions, and LV' an ad-hoc list of fresh variables. That is, for example the substitution $V_1, V_2 := V_2, V_1$ swaps V_1 and V_2 provided $V_1 \neq V_2$. Note that the syntactical category for parallel substitutions is left implicit in the B-Book.

Our embedding, as for the logic, represents this new syntactical category with a few adaptations. For example a parallel substitution is valid only provided the variables appearing on the left side all differ, and if the list of expressions on the right side has the same length than the list of variables. For the sake of simplicity, we represent such parallel substitutions using maps in $\mathbb{M} := \mathbb{I} \to \mathbb{E}$[8], a representation which naturally ensures such well-formedness conditions and simplify the encoding of the associated semantics, as we will see in the next paragraph.

On the other hand, maps can represent infinite parallel substitutions, something not possible with the Gsl, so some care is required. Basically, maps are not finite, but we can note that for any map M and any term T, there is a finite map M' such that T transformed by M is equal to T transformed by M' – we just restrict the scope of M to the variables appearing free in T. We can also provide a well-founded way to build any interesting map with the two following operations:

Definition 7.3.3 (Maps primitive operations)

$$\odot : \mathbb{M} \quad := \quad \mathbf{fun}\, i' : \mathbb{I} \to \dot{\chi}_{i'}$$

$$(i \backslash E) \oplus m : \mathbb{M} \quad := \quad \mathbf{fun}\, i' : \mathbb{I} \to \mathbf{if}\ i = i'\ \mathbf{then}\ E\ \mathbf{else}\ m\, i'$$

The map $\odot$ associates any variable to itself, that is it can be used to describe **skip**; $\oplus$ allows to overload a map with a pair (i, E), and can be used to build any parallel substitution starting from $\odot$. In essence, $\odot$ and $\oplus$ allows for the description of a well-founded subset in $\mathbb{M}$ encompassing any parallel substitution of the B Gsl (cf. Section 7.5.2 for an associated induction principle).

These concepts being introduced, we embed the Gsl syntax as follows:

Definition 7.3.4 (BiCoq Gsl syntax)

$$\begin{array}{rcl} \mathbb{S} & \triangleq & \langle \mathbb{I} \to \mathbb{E} \rangle \\ & | & \mathbb{P} \mid \mathbb{S} \\ & | & \mathbb{P} \dot{\Longrightarrow} \mathbb{S} \\ & | & \mathbb{S} \dot{[]} \mathbb{S} \\ & | & \underline{@}\, \mathbb{S} \\ & | & \mathbb{S}\,;\,\mathbb{S} \end{array}$$

[8] In fact maps are introduced earlier in our development to prove the results presented in Section 7.5.2 and detailed in Section 8.3.4; it is sensible to reuse them for the embedding of the Gsl.

As previously, we use the dotted notations to mark guest constructs, the exception being the raw *de Bruijn* binder $\underline{@}$, which is not parameterised by a variable and only binds a single variable, as in Section 7.2.1. And as previously, we also define a function providing a form of natural notation:

Definition 7.3.5 (Functional unbounded choice) $\dot{@}\, i{\cdot}S$ *denotes the invocation of the* unbounded choice function *with parameters* $i : \mathbb{I}$ *and* $S : \mathbb{S}$*, computing the* de Bruijn *term representing the standard* B *substitution* $@\, V{\cdot}S$*,* V *being encoded as* i*.*

7.3.2 Substitutions as predicate transformers

To define the semantics of the GSL constructs, we still use the same strategy of representing application as an external operation. That is, we do not enrich the syntax of terms with constructs like $[S]T$, preferring to define a function building the resulting term instead.

This function, defined only for predicates, is as follows:

Definition 7.3.6 (Functional substitution application)

$$\begin{array}{lcll} \text{Transform}(P:\mathbb{P}):\mathbb{S}\to\mathbb{P} & \triangleq & \langle m'\rangle & \mapsto & [[m']]P \\ & | & P' \mid s' & \mapsto & P' \mathbin{\dot\wedge} \text{Transform } P\ s' \\ & | & P' \dot\Longrightarrow s' & \mapsto & P' \dot\Rightarrow \text{Transform } P\ s' \\ & | & s_1 \mathbin{\dot[]} s_2 & \mapsto & \text{Transform } P\ s_1 \mathbin{\dot\wedge} \text{Transform } P\ s_2 \\ & | & \underline{@}\, s' & \mapsto & \underline{\forall}(\text{Transform } (\uparrow P)\ s') \\ & | & s_1 \,;\, s_2 & \mapsto & \text{Transform } (\text{Transform } P\ s_2)\ s_1 \end{array}$$

Here $[[m]]T$ denotes the application of the parallel substitution defined by the map m to the term T, and $\uparrow T$ the lifting of the term T – a standard operation on *de Bruijn* terms discussed in the next chapter. In fact the B semantics of the unbounded choice in associated with a side conditions about non freeness; in our embedding, thanks to the use of the *de Bruijn* representation, this is replaced by appropriate operations on the terms such as lifting, preventing automatically unwanted captures of free variables.

7.3.3 About refinement

We discuss now our attempts to capture the concept of refinement in our embedding. As indicated, the substitutions defined by GSL are encoded as constructors of a specific syntactical type denoted $\mathbb{S}$, and the semantics of a substitution as a COQ function in $\mathbb{P}\to\mathbb{P}$, that is a predicate tranformer.

The straightforward embedding of the higher-order definition of the substitution refinement provided in the B-BOOK (cf. Section 3.1) is then the following:

Definition 7.3.7 (B higher-order refinement for substitutions)

$$S_A \sqsubseteq S_C \quad \triangleq \quad \forall\, (P{:}\mathbb{P}),\ [S_A]P \Rightarrow [S_C]P$$

Yet our objective is also to validate the transformation of this higher-order definition into a first-order one, as it is described in the B-BOOK. This relies for example on the concepts of *aborting substitutions* and *terminating substitutions*. Quoting [Abr96], the predicate **abt** characterises the aborting substitutions, *i.e.* those which cannot establish anything, and its negation **trm** characterises the terminating substitutions. This description is associated to the following definitions:

Definition 7.3.8 (B aborting and terminating substitutions)

$$\frac{\neg[S]P \text{ for any predicate } P}{\mathbf{abt}(S)} \qquad \frac{\neg\mathbf{abt}(S)}{\mathbf{trm}(S)}$$

At this stage, the formalisation in our embedding is definitively less straightforward. The universal quantification on any predicate is trivial, but makes **abt** a predicate of the meta-logic (or according to the B-Book a second order predicate). And its exact meaning is unclear, even given the "formal" definition.

Let us consider more explicit notations (Coq being not so fond of implicit ones): $[S]P$ is a predicate, that is a term; yet to speak about true (provable) statements it is necessary to consider instead the sequent $\dot{\vdash}[S]P$. So the following definitions are possible candidates to capture in our embedding the concept of aborting substitution:

Definition 7.3.9 (Possible definitions for abortion in BiCoq)

$$\mathbf{abt}_0(S{:}\mathbb{S}) \triangleq \forall\,(P{:}\mathbb{P}),\ \dot{\vdash} \dot{\neg}[S]P$$

$$\mathbf{abt}_1(S{:}\mathbb{S}) \triangleq \forall\,(P{:}\mathbb{P}),\ \neg \dot{\vdash} [S]P$$

$$\mathbf{abt}_2(S{:}\mathbb{S}) \triangleq \neg\exists\,(P{:}\mathbb{P}),\ \dot{\vdash} [S]P$$

The formula $\mathbf{abt}_0(S)$ claims that for any predicate P, it is possible to refute $[S]P$ in the B logic. The formula $\mathbf{abt}_1(S)$ claims that for any predicate P it is impossible to prove $[S]P$ in the B logic. And $\mathbf{abt}_2(S)$ claims that there is no predicate P such that $[S]P$ is provable in the B logic. The two first definitions could represent the formal definition of **abt**, the alternatives being in using the negation of the B logic or the negation of the meta-logic (*i.e.* the negation of the Coq logic), whereas the last definition better captures the textual description. And the situation for defining termination is even worse.

We prove the following results:

Proposition 7.3.1

$$\mathbf{abt}_0(S) \Rightarrow \mathbf{abt}_1(S)$$

$$\mathbf{abt}_1(S) \Rightarrow \mathbf{abt}_2(S)$$

However we have not been able to prove any equivalence using the constructive Coq logic; so it is still an open question to know which one to choose.

The B-Book definitions have therefore to be clarified. Additional sources regarding the refinement theory, such as [BAW98, Bou07], should be considered to prove the equivalence between higher-order definitions and first-order definitions, such as $\mathbf{trm}(S) \Leftrightarrow [S]\mathbf{True}$. This is still a work to do, and we will not consider further these questions in this memoir.

7.4 A proven prover

Up to now, the deep embedding has been justified by the objective of verifying the B method, or more precisely at the current stage the B logic. Indeed, we are still missing the representation of various set constructs and their associated theorems, as well as the concept of refinement. It should also be noted that we have not yet attempted to represent B machines, or the generation of proof obligations, and so on.

This is however sufficient, at this stage, in demonstrating the interest of such a work, and its feasibility. It is rather successful, pinpointing various unknown difficulties with the theory itself, such as insufficient inference rules, invalid proofs, ambiguous or invalid definitions – some of these difficulties preventing straightforward progress of the embedding.

We consider this to be a sufficient justification for such a long and complex development. Yet, it is easy to have further benefits with reasonable additional works, optimising the gain to effort ratio. For example, deep embeddings can be used to develop mechanically checked tools, as illustrated here by the development of a first-order prover for the B logic which is validated in Coq (see also [RM05] for a similar approach).

This prover can be extracted from the embedding for example as an OCaml code that can be compiled to be used independently of Coq. That is, we favour here the development of a prover (or proof checker) which can be integrated in various environments, without having to tackle with Coq or BiCoq.

7.4.1 Implementing decidable properties

Inference rules and theorems of the B-Book are often associated to various side conditions, for example related to non freeness or the fact that a predicate appears in (belongs to) a proof environment. These conditions can for example be described in Coq by inductive relations mimicking precisely the definition of the B-Book.

With the objective of developping a proven prover, we need to provide functions computing such conditions. A possible approach, in Coq, is to prove that a given predicate is decidable, and to extract from this proof a program deciding the predicate. Yet this is not the approach that we adopt in BiCoq, as the produced code – always correct – can be rather surprising (cf. the example of division in Section 3.2). To favour a better control of the code of these functions in terms of conciseness, efficiency, and readability, we therefore adopt a different approach.

Indeed, it is also possible, using the **Extraction** command, to extract programs not from proofs but directly by exporting functions (described in the internal Coq Ml language) for example as an OCaml code. The idea is therefore to explicitly provide Ml functions deciding predicates, and to extract these functions.

It is more efficient, adopting this approach, to formalize the generic relationship existing between a predicate and a function deciding this predicate. For P a predicate over a type T and f a function in $T \to \mathbb{B}$, $P \sim f$ means that f decides P, that is:

Definition 7.4.1 (Implementation relation)

$$P \sim f \quad \triangleq \quad \forall (t{:}T), (f\ t = \top \Rightarrow P\ t) \wedge (f\ t = \bot \Rightarrow \neg P\ t)$$

$P \sim f$ of course guarantees that P is decidable; furthermore the following properties hold:

Proposition 7.4.1

$$P \sim f \Rightarrow \forall (t{:}T), P\ t \Rightarrow f(t) = \top$$

$$P \sim f \Rightarrow \forall (t{:}T), \neg P\ t \Rightarrow f(t) = \bot$$

We also define folding as the conjunctive extension to lists as follows:

Definition 7.4.2 (Conjunctive folding)

$$\mathbf{Fold}_p(P) := \mathbf{fun}\ (l{:}\mathbf{list}\ T) \mapsto \forall\ (t{:}T),\ t \in l \to P(t)$$

$$\mathbf{Fold}_f(f) := \mathbf{fun}\ (l{:}\mathbf{list}\ T) \mapsto \mathbf{match}\ l\ \mathbf{with}\ [\,] \mapsto \top \mid h :: t \mapsto f(h) \wedge \mathbf{Fold}_f\ f\ t$$

We can then show that if f decides P then the folding of f decides the folding of P:

Definition 7.4.3 (Monotony of implementation by folding)

$$(P \sim f) \quad \Rightarrow \quad (\mathbf{Fold}_p(P) \sim \mathbf{Fold}_\mathbf{f}(f))$$

That is, provided a function deciding freeness of a variable in a predicate, for example, we can use this result to know that the folding of this function decides freeness in a proof environment.

7.4.2 A proven prover

BiCoq includes dedicated functions written in the internal Coq Ml, thereafter named B *tactics*, to emulate the application of B inference rules or theorems to sequents.

By providing such a dedicated piece of code for each of the inference rules of the B logic, and by proving them correct, we got a correct and complete prover – that is, a result can be proven with the prover iff it can be proven in the B logic. Of course additional B tactics can be provided to represent the application of more complex theorems.

B tactics operate over a sequent, *i.e.* in a pair $(\Gamma, P) : \mathbf{list}\ \mathbb{P} \times \mathbb{P}$ – not to be confused with $\Gamma \dot{\vdash} P$ which is a type of B proofs. They return a list of such pairs. The sequent passed as a parameter represents the goal, the returned sequents represent the subgoals; if this list is empty that means that the B tactic has succeeded to prove the goal. In case the tactic cannot be applied to a goal (because it has not the expected pattern), the tactic does nothing and returns the list containing only the goal passed as a parameter (the alternative is to return an option type, **none** indicating the inadequacy of the tactic for the considered goal, or the absence of progress).

The following examples illustrate the concepts:

Example 7.4.1 (Axiom and conjunction tactics)

$$T_\in(\Gamma : \mathbf{list}\ \mathbb{P})(P : \mathbb{P}) := \mathbf{if}\ P \in \Gamma\ \mathbf{then}\ [\,]\ \mathbf{else}\ [(\Gamma, P)]$$

$$T^i_\wedge(\Gamma : \mathbf{list}\ \mathbb{P})(P : \mathbb{P}) := \mathbf{match}\ P\ \mathbf{with}\ P_1 \dot{\wedge} P_2 \to [(\Gamma, P_1), (\Gamma, P_2)] \mid \_ \to [(\Gamma, P)]$$

Following the same principles, numerous (much more complex) B tactics are provided in our embedding, implementing theorems or even strategies, such as the decision procedure for propositional calculus described in the *B-Book*.

For each B tactic T, the correctness is ensured by a proof that the goal is indeed derivable from the subgoals, *i.e.* that if $T(\Gamma, P) = [(\Gamma_1, P_1), \ldots, (\Gamma_n, P_n)]$, then:

Definition 7.4.4 (Tactic correctness)

$$\Gamma_1 \dot{\vdash} P_1 \Rightarrow \ldots \Rightarrow \Gamma_n \dot{\vdash} P_n \Rightarrow \Gamma \dot{\vdash} P$$

This prover is not really usable in its current form, lacking unification algorithms, automation and human-machine interface. Yet it can be coupled with other tools, for example a B parser using the platform Brillant [CPR+05], or used as a proof checker to validate B proofs produced by other provers, provided a common proof description language. This last use is again of special interest in the context of security evaluations, as a flawed prover (e.g. in which a paradox can be derived) is as dangerous as an inconsistent theory. Separating the proof generation with advanced automation but relaxed constraints, and the proof checking with a strict respect of the theory, appears to be a relevant strategy.

7.4.3 Toward a certifying prover

There are for now two ways to build B proofs using BiCoq:

- Directly within Coq, using B provability predicate constructors and B theorems, that represent in fact B proof terms (terms whose type is of the form $\Gamma \vdash P$);

- Using the prover, that is a set of Ml functions extracted from BiCoq and operating purely at the syntactical level.

The second approach has the merit to be usable without knowing or even using Coq, as the core engine of the prover is just a set of OCaml functions that happen to be mechanically checked in Coq. Being correct, any result proven with it is a valid theorem of the B logic. But how do we record a proof produced with this prover?

The simplest approach, maintaining the independency with Coq and BiCoq, is to build a tree of identifiers and parameters to memorize function calls during proof construction. Such a tree is a proof script, that can be replayed – that is recomputed, something which is not always immediate if advanced tactics are used (for example the decision procedure for propositional calculus). In fact, this can even result in portability problems, if the machine on which we replay the proof is not as powerful as the one on which the proof was developped; time out or memory overflow can then appear during verification. This is a problem if an independent evaluator has to check the provided proofs.

The alternative is to record not a proof script but a proof term, in the sense defined by the first approach herebefore (as for the Zenon prover [BDD07]). This does not change the core engine of the prover, that has still to provide tactics operating at the syntactical level to help the developer during proof construction; but instead of memorising the actions of the developer, it builds a proof term. Such a proof term can then be easily checked using BiCoq (by a simple invocation of the Coq compiler), therefore satisfying the *de Bruijn* criterion.

This approach has several merits:

- The prover, in fact, does not need anymore to be correct, as any invalid proof will produce an invalid proof term rejected during checking. That is it allows for the development of prover focusing on automation and efficiency, correctness being tackled by a different tool.

- It allows for an efficient cooperation between provers as it is more easy to compose proof terms in the common language without having to question the validity.

- An independent evaluator can check any proof by a simple invocation of the Coq compiler; he does not need to replay (recompute) the proof script, nor to use the prover, nor even to know which prover was used. This can be important in industrial contexts, when the access to the tool may not be guaranteed to all the actors of the certification process, for example due to cost or intellectual property rights.

It would be straightforward to produce a branch of BiCoq only providing the elements required for the definition of the provability predicate. Such a branch would act for BiCoq as the core type-checker of Coq: a relatively small piece of code, trusted and sufficient to check any proof. To have a more compact language however, the inclusion in this branch of some theorems – intuitively acting as macros – can also be considered.

7.5 New results for the B logic

Another possible use of a deep embedding such as BiCoq, beyond the validation of the theory and the development of mechanically checked tools, is to prove new (complex) results. In fact, we can even prove results that would not be expressible in the guest logic, by using features of the host logic.

This section illustrates this approach with the presentation of several theorems, that in our view are interesting from a theoretical point of view, but also from a practical point of view, as they offer the opportunity to define powerful B tactics for a prover easing human-led proof construction or optimising automation. Furthermore, they seem to provide a justification of current practices in existing B provers.

These results describe new forms of congruence rules, and allow for the replacement of equal expressions or equivalent predicates. The B logic primitively includes an inference rule for the *Leibniz*' law:

Definition 7.5.1 (B inference rule for equality)

$$\frac{\Gamma \vdash E_1 = E_2 \quad \Gamma \vdash [V := E_1]P}{\Gamma \vdash [V := E_2]P}$$

That is, if $E_1 = E_2$, we can replace occurrences of E_1 in a term by E_2. This is for example the justification for replacing computations by their result (e.g. $1+2$ by 3), or for unfolding definitions – at least in first approach. Yet this inference rule has two limitations: it is only dealing with expressions, and it cannot be used to replace expressions having a free variable bound by the context in which they appear.

Let us consider a first illustration of these limitations:

Example 7.5.1 (Predicate replacement)

$$P_1 \Leftrightarrow P_2 \vdash P_1 \wedge Q \Leftrightarrow P_2 \wedge Q$$

In the absence of predicate replacement in B, there is no way to express a generic proof rule applicable here. Of course, this result can be proven with an *ad hoc* proof, but the intuition claims that it is trivially true and does not deserve such a dedicated effort.

But our problem is more generic; consider these additional examples:

Example 7.5.2 (Bound terms replacement)

$$\vdash \forall x \cdot x * 0 = 0$$

$$\vdash \forall x \cdot P \Leftrightarrow \forall x \cdot \neg\neg P$$

In this case, even provided *Leibniz*' rules for replacing expressions and predicates, these rules cannot be applied here. Indeed, we cannot for example replace $x*0$ by 0, as it contains a variable x which is bound; similarly, the variable x may appear free in P, preventing a straightforward conclusion for the second result.

Our last example introduces an additional level of complexity:

Example 7.5.3 (Conditional bound terms replacement)

$$y \in \mathbb{N}, y \neq 0 \vdash \forall x \cdot x \in \mathbb{N} \Rightarrow x/y \leq x$$

This example uses an expression x/y which is in fact a definition. As previously, we cannot justify the unfolding of this definition in its context, because the variable x is bound. But the interesting aspect here is that this definition is conditional, that is the unfolding is only justified provided $y \neq 0$ (which is the case here).

7.5.1 Substituting predicates

Our embedding first extends the *Leibniz'* rule to equivalent predicates, but relies on an extension of the B syntax. Indeed, the intuition of the congruence rules is to replace an expression E_1 appearing in a term T by an equal expression E_2, but this is not done directly: one has first to rewrite T as a pattern, that is to find T' such that $T = [x\backslash E]T'$. Variables therefore appear to act as placeholders for expressions that are manipulated through the meta-language substitution, and we have no similar construct for predicates.

To allow for the replacement of predicates in B terms, we introduces *propositional variables* in the syntactical type of B predicates as follows:

Definition 7.5.2 (Syntax extension for the B logic)

$$\begin{array}{rcll} \mathbb{P} & := & \mathbb{P} \dot{\wedge} \mathbb{P} & \\ & | & \mathbb{P} \dot{\Rightarrow} \mathbb{P} & \\ & | & \dot{\neg} \mathbb{P} & \\ & | & \underline{\forall} \mathbb{P} & \\ & | & \mathbb{E} \doteq \mathbb{E} & \\ & | & \mathbb{E} \dot{\in} \mathbb{E} & \\ & | & \pi \mathbb{A} & \textit{(Propositional variables)} \end{array}$$

where $\mathbb{A}$ is a set of names. Propositional variables are never bound (there are no binders for them) and have no associated inference rule: we limit the impact of their introduction by avoiding as far as possible the creation of new theorems. Note however that we are generating new instances of previous results of the propositional calculus, for example $\vdash \pi_a \dot{\Leftrightarrow} \pi_a$ – theorems such as $\vdash P \dot{\Leftrightarrow} P$ being in fact schemas in the B first order logic (for any P, it is provable, but there is no proof for all P).

Propositional variables are only manipulated through a dedicated form of meta-language substitution, $\langle a\backslash P\rangle T$ denoting the term obtained by replacing any occurrence of the propositional variable π_a in the term T by the predicate P. It is important to insist on the fact that, again, no specific semantics are given to the propositional variables or even to the new substitution: it is just a manipulation of terms which is possibly meaningless.

We however prove the following results, the first one being very similar to the inference rule for equal expressions:

Proposition 7.5.1 (Congruences for unbound predicates)

$$\Gamma \dot{\vdash} P_1 \dot{\Leftrightarrow} P_2 \quad \Rightarrow \quad \Gamma \dot{\vdash} \langle a\backslash P_1\rangle Q \quad \Rightarrow \quad \Gamma \dot{\vdash} \langle a\backslash P_2\rangle Q$$

$$\Gamma \dot{\vdash} P_1 \dot{\Leftrightarrow} P_2 \quad \Rightarrow \quad \Gamma \dot{\vdash} \langle a\backslash P_1\rangle Q \dot{\Leftrightarrow} \langle a\backslash P_2\rangle Q$$

$$\Gamma \dot{\vdash} P_1 \dot{\Leftrightarrow} P_2 \quad \Rightarrow \quad \Gamma \dot{\vdash} \langle a\backslash P_1\rangle E \doteq \langle a\backslash P_2\rangle E$$

These results are very intuitive and interesting, especially when dealing with formal proofs in such an embedding: they can considerably shorten proofs by avoiding numerous tedious steps for deconstructing terms.

However, they suffer the same important limitation than the inference rule for equality: *none of these results can be used to replace bound subterms.* Substitutions $[i\backslash E]$ and $\langle a\backslash P\rangle$ can cross binders, yet they mechanically prevent capture of free variables appearing in E or in P. This is a typical constraint for substitution in any logic, including the B logic, and not in any way a specific limitation of our embedding. The consequences are important; for example, it is not possible to use the previous congruence result to prove:

Example 7.5.4

$$\forall (i{:}\mathbb{I})(P{:}\mathbb{P}),\ \dot{\vdash} (\dot{\forall}\, i \cdot \dot{\neg}\dot{\neg} P) \dot{\Leftrightarrow} (\dot{\forall}\, i \cdot P)$$

7.5.2 Grafting

We therefore still want more generic congruence results, to cope with bound subterms. To be able to write such advanced theorems, it is clear that we need a new tool, a form of meta-language substitution able to interact with bound subterms – an operation unlikely to be descriptible in a standard B environment.

We want to be able to replace bound subterms, even under conditions, as in:

Example 7.5.5

$$y \in \mathbb{N}, y \neq 0 \vdash \forall\, x \cdot x \in \mathbb{N} \Rightarrow x/y \leq x$$

Indeed, the definition $(x/y) = \mathbf{max}\{z : \mathbb{N} \mid x \geq y * z\}$ is only valid provided $y \neq 0$. This has justified several iterations of our proofs and techniques in the embedding, detailed in the next chapter; we just present here our conclusions.

We define new functions for *grafting* expressions and predicates. These functions are a form of substitution allowing for the capture of free variables. They have no equivalent in standard B, yet they do not constitute an extension of the B logic: as for predicate substitution they are just operators to represent complex operations on B terms, that are not associated to inference rules or new syntactical constructs (except for the introduction of propositional variables as far as the grafting of predicates is used).

We denote $[i \lhd E]T$ the term obtained by grafting the expression E at any occurence of the free variable $\dot{\chi}_i$ in the term T, and $\langle a \lhd P \rangle T$ the similar function for grafting the predicate P at any occurrence of a propositional variable π_a. The intuitive behaviour of grafting is illustrated by this example:

Example 7.5.6 (Illustration of grafting)

$$[j \lhd i](\dot{\forall}\, i \cdot i \dot{\in} \mathbb{N} \dot{\Rightarrow} 0 \dot{\leq} j) \quad = \quad \dot{\forall}\, i \cdot i \dot{\in} \mathbb{N} \dot{\Rightarrow} 0 \dot{\leq} i \qquad \textit{provided } i \neq j$$

Note that grafting does not replace bound variables, but free variables; however variables in the replacing term can be captured. The precise definition of the grafting functions is described using the internal *de Bruijn* representation, and is presented in Section 8.3.2.

The associated congruence results are the following ones[9]:

Proposition 7.5.2 (Congruence for bound terms)

$$\Gamma \dot{\vdash} E_1 \dot{=} E_2 \quad \Rightarrow \quad \Gamma \perp_E E_1 \dot{=} E_2 \quad \Rightarrow \quad \Gamma \dot{\vdash} [i \lhd E_1]E \dot{=} [i \lhd E_2]E$$

$$\Gamma \dot{\vdash} E_1 \dot{=} E_2 \quad \Rightarrow \quad \Gamma \perp_P E_1 \dot{=} E_2 \quad \Rightarrow \quad \Gamma \dot{\vdash} [i \lhd E_1]P \dot{\Leftrightarrow} [i \lhd E_2]P$$

$$\Gamma \dot{\vdash} P_1 \dot{\Leftrightarrow} P_2 \quad \Rightarrow \quad \Gamma \perp_E P_1 \dot{\Leftrightarrow} P_2 \quad \Rightarrow \quad \Gamma \dot{\vdash} \langle n \lhd P_1 \rangle E \dot{=} \langle n \lhd P_2 \rangle E$$

$$\Gamma \dot{\vdash} P_1 \dot{\Leftrightarrow} P_2 \quad \Rightarrow \quad \Gamma \perp_P P_1 \dot{\Leftrightarrow} P_2 \quad \Rightarrow \quad \Gamma \dot{\vdash} \langle n \lhd P_1 \rangle P \dot{\Leftrightarrow} \langle n \lhd P_2 \rangle P$$

[9] Only a weaker form of the predicate grafting results have been proven at this stage, as they have not yet been reintroduced in the new version of the embedding; yet we consider safe to admit them for now.

$\Gamma \perp_F T$ is called the *orthogonality condition*, and is formally defined in Sections 8.3.2 and 8.3.3. Intuitively, it requires any variable appearing free in both Γ and T not to be captured in F (or at least not to be captured at the positions to which T is grafted). Its role is easily understood with the following counterexample where y is finally captured instead of staying free:

Example 7.5.7 (Role of the orthogonality condition)

$$x\dot{\in}\mathbb{N}, y\dot{\in}\mathbb{N}, \dot{\neg}y\dot{=}0 \dot{\vdash} x/y\dot{\leq}x \dot{\Leftrightarrow} \top$$

but $$x\dot{\in}\mathbb{N}, y\dot{\in}\mathbb{N}, \dot{\neg}y\dot{=}0 \dot{\vdash} \langle a \lhd x/y\dot{\leq}x\rangle(\dot{\forall}\, y\cdot\pi_a) \dot{\not\Leftrightarrow} \langle a \lhd \top\rangle(\dot{\forall}\, y\cdot\pi_a)$$

that is $$x\dot{\in}\mathbb{N}, y\dot{\in}\mathbb{N}, \dot{\neg}y\dot{=}0 \dot{\vdash} (\dot{\forall}\, y\cdot x/y\dot{\leq}x) \dot{\not\Leftrightarrow} (\dot{\forall}\, y\cdot\top)$$

We are just providing here the intuition about new and complex results derived in our embedding. The precise definition of the grafting functions and the orthogonality condition, as well as the exact congruence results, are provided in the next chapter, as some technical considerations about *de Bruijn* representations are required.

These congruence theorems have to be compared with the delta-lemma approach discussed in Section 5.3.3. This approach, described in [Bur00, BBM98], aims at tackling non defined terms, such as $\mathbf{min}(\emptyset)$ or $1/0$, in set theory and more specifically in the B logic. On the basis of three valued semantics, a minimal amount of verifications ensuring the validity of a proof is identified, the aim being to ease automation of proof. Our congruence results bring a very different vision. Indeed, we do not consider three valued semantics, nor are we concerned with possibly non defined expressions; but we propose a strategy to deal with equalities or equivalences under hypothesis (including conditional definitions), a strategy which is furthermore able to operate even with bound subterms.

7.5.3 Validity of the new results

The question of the applicability of such new results in the standard B logic appears to be a legitimate one for a B expert, or an independent evaluator. Indeed, these results are derived using higher-order logic and inductive proofs in Coq, which can appear as "alien" tricks. In other words, *are those results truly valid, or are they just artefacts resulting of the embedding, and consequently only valid in* BiCoq*?* Would the implementation of these rules in a prover for the B logic represents an implicit enrichment, possibly endangering the correctness of the prover?

The intuitive justification is provided by the *Curry-Howard* isomorphism (cf. Section 3.2). Let us consider the Coq proof using the BiCoq definitions of the first congruence result in the previous paragraph:

$$\Gamma \dot{\vdash} E_1\dot{=}E_2 \quad \Rightarrow \quad \Gamma \perp_E E_1\dot{=}E_2 \quad \Rightarrow \quad \Gamma \dot{\vdash} [i\lhd E_1]E\dot{=}[i\lhd E_2]E$$

The interpretation of this theorem in Type Theory is that the proof is a program that given a B proof of $\Gamma \dot{\vdash} E_1\dot{=}E_2$ is able to compute a B proof of $\Gamma \dot{\vdash} [i\lhd E_1]E\dot{=}[i\lhd E_2]E$ provided that $\Gamma \perp_F E_1\dot{=}E_2$. That is, we implicitly have a function whose signature is:

$$\mathit{Graft\_Congr}_1(\Gamma : \mathbf{list}\ \mathbb{P})(E_1\ E_2 : \mathbb{E})(H : \Gamma \dot{\vdash} E_1\dot{=}E_2)(i : \mathbb{I})(E : \mathbb{P}) : \Gamma \dot{\vdash} [i\lhd E_1]E\dot{=}[i\lhd E_2]E$$

Extracting this function – a purely intellectual exercise in this case, as this function's signature uses dependent types – and only using it when $\Gamma \perp_E E_1\dot{=}E_2$ (acting as a precondition, but easy to transform into a guard), we could indeed produce the appropriate B

proof terms. Or in other words, if a B proof of $\Gamma \vdash E_1 \doteq E_2$ exists, and if $\Gamma \perp_F E_1 \doteq E_2$, then we know that a B proof of $\Gamma \vdash [i \lhd E_1]E \doteq [i \lhd E_2]E$ exists as well, and we are even able to exhibit one.

The structure of the Coq proof is also very interesting: as we will see in the next chapter, this result is obtained by induction over E, and does not analyse the structure of the predicate $E_1 \doteq E_2$ or of the proof $\Gamma \vdash E_1 \doteq E_2$. That means that the corresponding program is a recursion over the structure of the term E, propagating a parameter $\Gamma \vdash E_1 \doteq E_2$ which is never decomposed (that is it is an opaque parameter used in a form of cut).

Note that this type of justifications is only possible thanks to the clear separation between the guest logic and the host logic in a deep embedding. Furthermore, as indicated, grafting is an operation on terms that cannot be described in the B logic or in a B environment, without even speaking about proving its properties.

Chapter 8

A Technical Review of the Embedding

This chapter explores some of the technical aspects of the deep embedding of B in Coq used in the previous chapter for the validation of the B logic. There are indeed numerous difficulties with such a development, such as designing appropriate representations, adapting proof methodologies or ensuring some form of optimisations. The techniques discussed in this chapter may be relevant for other types of language mechanisations, and may have wider applications beyond the admittedly quite narrow domain of deep embeddings, for example when dealing with compilers, analysers, etc.

As mentioned, we have implemented several versions of BiCoq, which in total amount to several tenths of thousand of lines of Coq. The justification for a full redevelopment after the initial version was to study technical alternatives, quickly compared here, but also to allow for the derivation of the full congruence results described in Section 7.5.2. The first version of BiCoq uses *de Bruijn* indexes, and includes propositional variables, the proven prover, and a limited version of the congruence results; two other versions have been redeveloped, in parallel, one using *de Bruijn* indexes and the other *de Bruijn* levels, with an improved calculus (leading to the full version of the congruence results); on the other hand these versions do not include a prover or results about propositional variables. There is however no difficulty into porting these aspects to the new version.

The rest of this chapter is organised as follows. Section 8.1 discusses various representations of terms. It starts with a few reminders about *de Bruijn* representations, comparing indexes and levels, before describing the notation used in BiCoq, in which we introduce context awareness. Section 8.2 presents a generic induction principle used in BiCoq proofs. Section 8.3 introduces a new operation for BiCoq terms – named *grafting* – and proposes to enrich the BiCoq representation with *namespaces* to tackle complex results. Finally Section 8.4 summarises some of the concepts introduced in the previous sections regarding term representation by describing a simply typed λ-calculus not requiring any typing context.

8.1 *De Bruin* representations

One of the problems to deal with when mechanising a language (cf. [ABF+05, CPW06]) is the representation of variables, and more specifically of bound variables. The standard approach, denoted λ_V in this chapter, uses names for bound and free variables. However, whereas it is easy to read, it suffers from various flaws.

Indeed, two terms differing only by the names of their bound variables (the so-called α-renaming), such as $\lambda x{\cdot}\lambda y{\cdot}x-y$ and $\lambda z{\cdot}\lambda x{\cdot}z-x$, should be considered as equal but are not when using a notation with names . One may also wonder how to compute the reduction of the substitution for the following cases:

Example 8.1.1 (Problems of capture in substitution)

$[x\backslash E]\lambda x{\cdot}T$ $\qquad$ $[x\backslash y]\lambda y{\cdot}T$

The so-called *de Bruijn* representations (cf. [dB72, Gor93, NV07]) address these problems by encoding bound variables as natural values acting as pointers to their binder; they define an α-quotiented representation, *i.e.* terms equivalent modulo α-renaming are indeed equal. They also provide clear and straightforward semantics to deal with capture phenomena applicable for example when considering substitutions.

We first introduce classical notions about *de Bruijn* representations in Sections 8.1.1-8.1.3 using a standard λ-calculus for the sake of clarity, before presenting the adaptation of these concepts to our embedding, and some improvements that are required to tackle advanced results.

8.1.1 Using indexes in λ-calculus: The $\lambda_{\mathbf{dBI}}$ notation

The most popular *de Bruijn* representation uses indexes, that are relative pointers to their binder (counting from the variables, that is the leaves in the tree representing the term). The value 0 represents the variable bound by the closest parent binder, as illustrated hereafter (raw *de Bruijn* binders and indexes are underlined for the sake of clarity):

Example 8.1.2 (*de Bruijn* indexes)

λ_V *notation* $\qquad$ $(\lambda x{\cdot}\lambda y{\cdot}(X_0{+}x{-}y))\,X_0$

λ_{dBI} *notation* $\qquad$ $\underline{\lambda}\,\underline{\lambda}(\underline{2}{+}\underline{1}{-}\underline{0})\,\underline{0}$

We have chosen here to use the *pure nameless notation*: the free variable X_0 is represented by the value 2, assuming it is the first free variable in the context (left implicit here). Such a pointer is said to be *dangling* as its value exceeds the number of parent binders.

This notion of *context* is important, and is generally managed explicitly in a λ-calculus, in various forms. In simply typed λ-calculus, for example, it is the typing context, indicating the type of the free variables (cf. Section 8.4.1 for an illustration). The context can also be seen as a side product of the parsing of a λ_{V} term and its transformation into λ_{dBI} term, recording the name of free variables and allowing for pretty-printing (that is the reverse transformation from λ_{dBI} to λ_{V}); let us reconsider our previous example:

Example 8.1.3 (*de Bruijn* indexes with naming context)

λ_V *notation* $\qquad$ $(\lambda x{\cdot}\lambda y{\cdot}(X_0{+}x{-}y))\,X_0$

λ_{dBI} *notation* $\qquad$ $\underline{\lambda}\,\underline{\lambda}(\underline{2}{+}\underline{1}{-}\underline{0})\,\underline{0}$ $\qquad$ *Context:* $[0{\mapsto}X_0]$

Whereas we often mention the notion of context in the rest of this chapter, we do not explicitly manage such a context (it is however abstracted through λ-height).

Another possible alternative *de Bruijn* representation is to use the *locally nameless notation*; in this case, there are also named variables which are syntactically distinct of indexes, such named variables being always free. It is then possible to choose two main strategies with regard to the nature of the variables represented by indexes:

- Indexes only represent bound variables, and dangling indexes are not allowed; the type of terms is then (possibly implicitly) a dependent type – or a nested datatype, as in [BP99] – the maximum value of an index being constrained by the λ-height, that is the context of the index.

- Indexes can represent bound or free variables; in such a situation there are therefore two different ways to represent free variables, that can be distinguished by a form of specialisation – for example considering named variables as unification variables and dangling indexes as free variables manipulated only through binding and application, as in [DHK00].

We will not consider further the locally nameless notation approach. It was not really explored in any version of BiCoq, or more precisely it was not retained at the beginning of the development – to avoid further complexity of the syntactical type that has already 15 distinct constructors – and it was not required later. Note however that our final solution, with namespaces (cf. Section 8.3), can be seen as a generalisation of a locally-nameless representations, that are also considered in Section 8.4.8.

8.1.2 Managing indexes in $\lambda_{\mathbf{dBI}}$

The index representing a given variable, either bound or free, changes with its λ-*height*, *i.e.* the number of parent binders, as illustrated hereafter:

Example 8.1.4 (Influence of the λ-height)

λ_V *notation*	$(\lambda x{\cdot}(x{+}\lambda y{\cdot}(x{-}y)X_0))\ X_0$
λ_{dBI} *notation*	$\underline{\lambda}(\underline{0}{+}\underline{\lambda}(\underline{1}{-}\underline{0})\underline{2})\ \underline{0}$

The first occurrence of x in the λ_V notation is bound by the closest parent binder and is therefore denoted $\underline{0}$ in the λ_{dBI} notation. The second occurrence of x appears under a second binder (capturing y); it is therefore denoted $\underline{1}$.

This makes reading or manipulating λ_{dBI} terms by hand rather awkward.

It is therefore customary to provide standard operators to support index management, either technical such as *lifting* (denoted $\uparrow$) or user-relevant such as substitution. The former is used by the latter to adapt terms when crossing a binder, as illustrated here:

Definition 8.1.1 (Standard *de Bruijn* lifting)

$$
\begin{array}{l}
\uparrow_h : \lambda_{dBI} \to \lambda_{dBI} \quad \triangleq \\
\quad \mid \underline{\lambda} T' \mapsto \underline{\lambda}(\uparrow_{h+1} T') \\
\quad \mid i' \mapsto \textbf{if } h \leq i' \textbf{ then } i'+1 \textbf{ else } i' \\
\quad \mid \ldots
\end{array}
$$

Definition 8.1.2 (Standard *de Bruijn* substitution)

$$
\begin{array}{l}
[i \backslash E] : \lambda_{dBI} \to \lambda_{dBI} \quad \triangleq \\
\quad \mid \underline{\lambda} T' \mapsto \underline{\lambda}[x{+}1 \backslash \uparrow E] T') \\
\quad \mid i' \mapsto \textbf{if } i = i' \textbf{ then } E \textbf{ else } i' \\
\quad \mid \ldots
\end{array}
$$

Indeed, crossing a binder modifies the λ-height, so the index i has to be incremented to represent the same variable, and similarly dangling indexes of E have to be incremented to maintain their semantics as well as to avoid their capture – this is the role of lifting, that makes the required adaptations to the representation of a term to preserve its semantics when crossing a binder.

To identify dangling indexes, lifting is parameterised by the contextual information h recording the current λ-height, left implicit in our notations when $h=0$ (other values of h resulting only from recursive calls for bound subterms). This is the only form of context that we need to manage to ensure the validity of all our operations.

This toolbox for λ-calculus can be completed with operators defining a user-friendly representation (introduced without details in Section 7.2.1), as in [Gor93]. Indeed, the λ_{V} abstraction in λ_{dBI} is not a simple transformation, as illustrated here:

Example 8.1.5 (Abstraction for *de Bruijn* terms)

λ_V *notation*	$X_0+X_1+X_2$	$\rightarrow$	$\lambda x\cdot(X_0+x+X_2)$	*(binding* X_1*)*
λ_{dBI} *notation*	$\underline{0}+\underline{1}+\underline{2}$	$\rightarrow$	$\underline{\lambda}(\underline{1}+\underline{0}+\underline{3})$	*(binding* $\underline{1}$*)*

To assist the user, we provide term functions able to mimick standard operations (and therefore standard representations) by computing the required *de Bruijn* terms. Together, these functions define what we call the *functional representation*; this is not a new language or a new notation but a different way to denote terms using functional expressions of the meta-language.

For example, $\dot{\lambda}\, i \cdot T$ does not denote the λ_{V} abstraction, but the invocation of the *function* $\dot{\lambda}$ with parameters $i : \mathbb{I}$ and $T : \lambda_{\mathrm{dBI}}$ and returning a term in λ_{dBI}. This functional abstraction is defined as follows:

Definition 8.1.3 (*de Bruijn* functional abstraction)

$$\dot{\lambda}\, i \cdot T \quad \triangleq \quad \underline{\lambda}(\mathrm{Abstr}_0\, i\, T)$$

where Abstr *is the following function:*

$$\begin{array}{l} \mathrm{Abstr}_h(i:\mathbb{I}) : \lambda_{dBI} \rightarrow \lambda_{dBI} \quad \triangleq \\ \quad \mid \underline{\lambda} T' \mapsto \underline{\lambda}(\mathrm{Abstr}_{h+1}\ (i+1)\ T') \\ \quad \mid i' \mapsto \left\{ \begin{array}{l} i' \ \textit{if}\ i' < h \\ h \ \textit{if}\ i' \geq h \ \textit{and}\ i' = i \\ i'+1 \ \textit{if}\ i' \geq h \ \textit{and}\ i' \neq i \end{array} \right. \\ \quad \mid \ldots \end{array}$$

Note that using the functional representation, i is an index and T a λ_{dBI} term, so this is not a named representation; however this approach is interesting to describe terms or to support parsing, as illustrated thereafter (cf. also the figure in Section 7.2.1):

Example 8.1.6 (Use of functional abstraction)

λ_V *notation*	$\lambda\, x_2 \cdot (x_2 + \lambda\, x_3 \cdot (x_3 + x_2 + X_0)\ X_1)$
Functional representation	$\dot{\lambda}\, \underline{2} \cdot (\underline{2} + \dot{\lambda}\, \underline{3} \cdot (\underline{3}+\underline{2}+\underline{0})\ \underline{1})$
λ_{dBI} *notation*	$\underline{\lambda}(\underline{0}+\underline{\lambda}(\underline{1}+\underline{0}+\underline{2})\ \underline{2})$

*From the standard representation on the first line, the functional representation on the second line is obtained by a straightforward parsing replacing the binder λ of λ_V by the invocation of the functional abstraction $\dot{\lambda}$, and any variable by an index (without any computation related to the λ-height). This expression can then be evaluated (*Coq *β-reduced) to obtain the raw* de Bruijn *term of the third line.*

Except for the fact that we are using natural values instead of names, the functional representation is indeed much more user-friendly. Being close to the natural representation, this also justifies why we have favoured the use of the functional representation for example to embed the B inference rules in Section 7.2.2.

8.1.3 Comparing indexes and levels in λ-calculus

The alternative to indexes is to use levels, discussed for example in [HAF01]; this is the so-called λ_{dBL} notation. Levels (denoted by hatted natural values) are absolute pointers counting binders from the root of the term; the value $\hat{0}$ then represents the variable bound by the farest parent binder, as illustrated here:

Example 8.1.7 (*de Bruijn* levels)

λ_V *notation*	$\lambda x{\cdot}\lambda y{\cdot}(X_0{+}x{-}y)$
λ_{dBL} *notation*	$\hat{\lambda}\,\hat{\lambda}(\hat{2}{+}\hat{0}{-}\hat{1})$

Index and level notations only differ in the representation of bound variables. Levels ensure a unique representation in a term of a bound variable, whereas with indexes this representation depends on the variable position (its context); on the other hand, bound levels need frequent renumbering during abstraction or substitution whereas bound indexes are never modified (except of course when eliminating the corresponding binder).

Both indexes and levels have been investigated in our embedding: nearly two full versions have been developed, yet without reaching a general conclusion. Indeed for most of our needs, levels are more efficient; they are easier to deal with, theorems tend to be more generic and proofs simpler. Consider as a typical example the lifting functions for indexes (left code) and levels (right code):

Example 8.1.8 (Comparing lifting for indexes and levels)

$$
\begin{array}{l}
\uparrow_h : \lambda_{dBI} \to \lambda_{dBI} \quad \triangleq \\
\quad \mid \underline{\lambda} T' \mapsto \underline{\lambda}(\uparrow_{h+1} T') \\
\quad \mid i' \mapsto \textbf{if } h \leq i' \textbf{ then } i'{+}1 \textbf{ else } i' \\
\quad \mid \ldots
\end{array}
\qquad
\begin{array}{l}
\hat{\uparrow}^L : \lambda_{dBL} \to \lambda_{dBL} \quad \triangleq \\
\quad \mid \hat{\lambda} T' \mapsto \hat{\lambda}(\hat{\uparrow}^L T') \\
\quad \mid i' \mapsto i'{+}1 \\
\quad \mid \ldots
\end{array}
$$

As mentioned in the previous paragraph, in λ_{dBI} the lifting operation $\uparrow_h$ requires a contextual parameter h to identify dangling indexes, bound indexes being never modified. On the contrary the λ_{dBL} equivalent operation $\hat{\uparrow}^{\mathrm{L}}$ increments all levels, regardless of their context, and the λ-height parameter is therefore not required. This has numerous consequences; for example when using levels theorems about lifting are not specialised according to the value of this parameter.

Yet our final choice, as well as our recommendation for other developments, is to use de Bruijn *indexes.* Indeed the congruence results about grafting (presented in Sections 7.5.2, 8.3.2 and 8.3.3) are proven using complex methods, and in particular parallel substitutions (cf. Sections 7.3.1 and 8.3.4). The important point in these proofs is that

parallel substitutions provide an alternative encoding of standard operations on terms (such as lifting, abstraction, etc.). Whereas it is indeed possible to emulate most operations using parallel substitutions in the λ_{dBI} representation, as those operation are similar to parallel substitutions in never modifying bound indexes, this is not the case in the λ_{dBL} representation[1].

We therefore consider that whereas *de Bruijn* levels are simpler to use and to tackle, there is a clear advantage for *de Bruijn* indexes when dealing with advanced techniques required for example when manipulating subterms whose free variables are bound by the context.

Having noted this analysis, one can wonder whether an appropriate language mechanisation toolbox should not offer the support of several forms of notations, provided with translation operators and homomorphisms proofs. As mentioned, formally dealing with names in the λ_{V} notation is difficult, but this is the only user-friendly representation; the λ_{dBL} notation is efficient and easy to tackle in proofs, but more subtle results are out of reach; finally the λ_{dBI} notation is powerful, but also often awkward. In a framework combining several of these representations, it would be possible to express and prove a standard result by choosing the most appropriate representation, and then to translate this result into the other representations by using homomorphism properties.

8.1.4 Operations on B terms in BiCoq

The notations and functions described up to now in this paragraph are standard and well known – except may be for the functional representation approach which provides a *form* of λ_{V} notation. What follows is the version used in our embedding, based on *de Bruijn* indexes, adapted to the B syntax introduced in Section 7.2.1.

In essence, the translation should be straightforward, just requiring an adaptation to the term constructors of BiCoq: $\underline{\forall}$ is a binder for which term operations should behave as for $\underline{\lambda}$ in λ_{dBI}, $\{\underline{\_}|\underline{\_}\}$ is also a binder but only for its right parameter, etc. However, we also proceed to additional modifications, to improve the management of the context.

We present the full details of the implementation only for one specific operation, lifting, before adopting a more compact presentation. The details are indeed rather long and technical, for two main reasons. The first one is that the B syntax is associated to many constructors – making the detailed presentation of functions rather long, without even speaking about representing the associated proofs by induction. The second reason is that in BiCoq, we deal with two sorts of syntactical constructs, $\mathbb{E}$ for the expressions and $\mathbb{P}$ for the predicates, and with their union $\mathbb{T}$ (cf. Section 7.2.1). To ensure type-checking, each term operation is in practice defined through three functions, one per syntactical sort; that is, for example, for lifting:

- we have a lifting function for expressions, returning an expression ($\uparrow^E : \mathbb{E} \to \mathbb{E}$);
- we have a lifting function for predicates, returning a predicate ($\uparrow^P : \mathbb{P} \to \mathbb{P}$);
- we have a lifting function for terms, returning a term ($\uparrow : \mathbb{T} \to \mathbb{T}$).

It is important to have this distinction, to know for example that if $P \dot{\wedge} E \dot{\in} S$ is syntactically valid (that is P is a predicate, E and S are expressions), then $\uparrow^P P \dot{\wedge} \uparrow^E E \dot{\in} \uparrow^E S$ is syntactically valid as well – this would not be possible using only a single lifting function for terms. Where possible, of course, we however use Coq coercion mechanisms, indicating

[1] On the other hand, one could wonder if a whole different theory is not possible for levels, for example considering a form of parallel grafting instead of parallel substitution to emulate standard operations.

that it is always acceptable to automatically cast $\mathbb{E}$ and $\mathbb{P}$ into $\mathbb{T}$ where necessary. But for the discussion in this memoir, such a level of details is basically irrelevant.

The lifting function is modified as follows to deal with the syntactical constructors of our embedding – noting that as $\{\!|\,|\!\}$ does not bind its left parameter, the left λ-height is not incremented:

Definition 8.1.4 (Lifting BiCoq predicates – detailed version)

$$\begin{array}{rcl}
\uparrow_h^P\!:\mathbb{P}\to\mathbb{P} \triangleq & \pi_a & \mapsto \pi_a \\
| & \dot{\neg}P' & \mapsto \dot{\neg}(\uparrow_h^P P') \\
| & \underline{\forall}P' & \mapsto \underline{\forall}(\uparrow_{h+1}^P P') \\
| & P_1 \dot{\wedge} P_2 & \mapsto (\uparrow_h^P P_1)\dot{\wedge}(\uparrow_h^P P_2) \\
| & P_1 \dot{\Rightarrow} P_2 & \mapsto (\uparrow_h^P P_1)\dot{\Rightarrow}(\uparrow_h^P P_2) \\
| & E_1 \doteq E_2 & \mapsto (\uparrow_h^E E_1)\doteq(\uparrow_h^E E_2) \\
| & E_1 \dot{\in} E_2 & \mapsto (\uparrow_h^E E_1)\dot{\in}(\uparrow_h^E E_2)
\end{array}$$

Definition 8.1.5 (Lifting BiCoq expressions – detailed version)

$$\begin{array}{rcl}
\uparrow_h^E\!:\mathbb{E}\to\mathbb{E} \triangleq & \dot{\Omega} & \mapsto \dot{\Omega} \\
| & \omega_{i'} & \mapsto \omega_{i'} \\
| & \dot{\chi}_{i'} & \mapsto \dot{\chi}(\textbf{if } h\leq i' \textbf{ then } i'+1 \textbf{ else } i') \\
| & \dot{\mathscr{C}}\left(E'\right) & \mapsto \dot{\mathscr{C}}\left(\uparrow_h^E E'\right) \\
| & \dot{\wp}\left(E'\right) & \mapsto \dot{\wp}\left(\uparrow_h^E E'\right) \\
| & E_1 \dot{\times} E_2 & \mapsto (\uparrow_h^E E_1)\dot{\times}(\uparrow_h^E E_2) \\
| & E_1 \dot{\mapsto} E_2 & \mapsto (\uparrow_h^E E_1)\dot{\mapsto}(\uparrow_h^E E_1) \\
| & \{\!|\, E' \,|\!|\, P' \,|\!\} & \mapsto \{\!|\, \uparrow_h^E E' \,|\!|\, \uparrow_{h+1}^P P' \,|\!\}
\end{array}$$

Definition 8.1.6 (Lifting BiCoq terms – detailed version)

$$\begin{array}{rcl}
\uparrow_h\!:\mathbb{T}\to\mathbb{T} \triangleq & \mathrm{Trm_of_Prd}\, P' & \mapsto \uparrow_h^P P' \\
| & \mathrm{Trm_of_Exp}\, E' & \mapsto \uparrow_h^E E'
\end{array}$$

Clearly, this last definition of lifting for terms is not well-typed, as it returns an expression or a predicate whereas a term is expected; yet Coq automatically uses the defined coercions to cast the values.

As expected, the only interesting cases are those for the binders and the variables (that is $\underline{\forall}$, $\{\!|\,|\!\}$ and $\dot{\chi}$); the other cases are just straightforward recursion, and will not be detailed further. Combined with the use of coercions, we therefore summarise lifting as follows:

Definition 8.1.7 (Lifting BiCoq terms – compact version)

$$\begin{array}{rcl}
\uparrow_h\!:\mathbb{T}\to\mathbb{T} \triangleq & \underline{\forall}P' & \mapsto \underline{\forall}(\uparrow_{h+1} P') \\
| & \{\!|\, E' \,|\!|\, P' \,|\!\} & \mapsto \{\!|\, \uparrow_h E' \,|\!|\, \uparrow_{h+1} P' \,|\!\} \\
| & \dot{\chi}_{i'} & \mapsto \dot{\chi}(\textbf{if } h\leq i' \textbf{ then } i'+1 \textbf{ else } i') \\
| & \ldots & \textit{(straightforward recursion)}
\end{array}$$

The same principles are applied for all the other functions on terms.

8.1.5 Context awareness

Beyond the adaptation to the B *syntax, we have also improved the operations with a better management of the λ-height parameter.* For example the BiCoq functional abstractions are defined as follows:

Definition 8.1.8 (BiCoq functional abstractions)

$$\begin{array}{rcl} \dot\forall\, i\cdot P & \triangleq & \underline{\forall}(\mathrm{Abstr}_0\, i\, P) \\ \dot\exists\, i\cdot P & \triangleq & \dot\neg(\dot\forall i\cdot \dot\neg P) \\ \{i{:}E \mid P\}^{\cdot} & \triangleq & \{\!| E \downarrow \mathrm{Abstr}_0\, i\, P |\!\} \end{array}$$

where Abstr is the following function:

$$\begin{array}{rcl} \mathrm{Abstr}_h(i{:}\mathbb{I}){:}\mathbb{T}\to\mathbb{T} \triangleq \underline{\forall}P' & \mapsto & \underline{\forall}(\mathrm{Abstr}_{h+1}\,(\uparrow_h i)\,P') \\ \mid \{\!| E' \downarrow P' |\!\} & \mapsto & \{\!| \mathrm{Abstr}_h\, i\, E' \downarrow \mathrm{Abstr}_{h+1}\,(\uparrow_h i)\,P' |\!\} \\ \mid \dot\chi_{i'} & \mapsto & \dot\chi(\textbf{if } h{\le}i' \wedge i{=}i' \textbf{ then } h \textbf{ else } \uparrow_h i') \\ \mid \ldots & & \textit{(straightforward recursion)} \end{array}$$

Compared with the definition of *Abstr* for λ_{dBI} given in Section 8.1.2, we have of course adapted the *Abstr* to the B syntax. But there are a few additional changes to note, related to the use of the λ-height parameter h.

It is not easy to justify these changes in a few words; they summarise several attempts to optimise the representation but also the associated proofs in the various versions of BiCoq. Originally, they result from a simple observation: any call of the form $Abstr_h\, i\, T$ is expected to be the result of a recursion starting from $Abstr_0\, i'\, T'$, *i.e.* from the root of the term, before crossing any binder. Therefore normal uses ensure that the condition $h{\le}i$ is a recursion invariant. Whereas this condition appears normally as a precondition in most theorems (that is we have propositions of the form $h{\le}i \Rightarrow P$) the general idea of our changes is to capture this invariant directly in the code of all the operations on term as a guard, for example using **if** $h{\le}i$ **then** ….

First, when crossing a binder we do not increment indexes anymore, but we lift them from the current λ-height, in other words we introduce a condition on the incrementation. Therefore, in the function *Abstr*, we replace the computation $i{+}1$ by $\uparrow_h i$, that is by **if** $h{\le}i$ **then** $i{+}1$ **else** i[2].

Similarly, when dealing with a variable, we check whether or not this variable is the one we are looking for, but also that $h{\le}i'$ – intuitively to prevent capture of already bound indexes. In both cases we are enforcing the fact that when using *de Bruijn* indexes bound variables should never be modified (except when eliminating the corresponding binder).

When the invariant $h \le i$ is broken because of an ill-formed call, the functional abstraction for λ_{dBI} (cf. Section 8.1.2) returns a term which is meaningless – or at least semantically unrelated to its parameters; indeed, it can for example replace occurrences of the bound index i by index h which is (re)captured by the new head binder, in an unmonitored way. On the contrary, our modified functional abstractions are such that when the invariant $h{\le}i$ is broken they return the term lifted and quantified, but without capture – returning this term may not be useful, but at least it is not meaningless.

The benefits of our modified version are not computational but logical. Both versions of *Abstr*, the standard one for λ_{dBI} and the context aware one for BiCoq, are equivalent

[2]This is an abuse of notation; in practice we we define a lifting function for indexes, but we do not distinguish this function from the lifting function for variables in this memoir.

for any well-formed invocation (that is ensuring that $h \leq i$), which is automatically the case when starting from the root of the term (and therefore for the functional abstractions). But we have learned that a stricter discipline in managing contexts is a very good practice, easing the expression of theorems as well as their proofs. Using the λ-height parameter h to express conditions in term operations ensures an explicit management of the context, a form of weak typing useful for complex proofs.

More generally, in any function manipulating terms, we exploit as far as possible the λ-height parameter h with its precise value – including in our code a form of context calculus. This discipline even leads to generalise the λ-height parameter to functions that don't need it, to explicitly record the current λ-height.

Let us illustrate this with the trivial function deciding whether a variable appears free in a term; the first version is the standard one, and the second version the context aware one used in BiCoq:

Definition 8.1.9 (Standard freeness)

$$\begin{array}{lcll} \mathrm{Free}^S(i{:}\mathbb{I}){:}\mathbb{T}\to\mathbb{B} & \triangleq & \underline{\forall} P' & \mapsto \mathrm{Free}^S\ (i{+}1)\ P' \\ & | & \{ E' \bot P' \} & \mapsto \mathrm{Free}^S\ i\ E' \vee \mathrm{Free}^S\ (i{+}1)\ P' \\ & | & \dot{\chi}_{i'} & \mapsto i'{=}i \\ & | & \ldots & \textit{(straightforward recursion)} \end{array}$$

Definition 8.1.10 (BiCoq freeness)

$$\begin{array}{lcll} \mathrm{Free}_h(i{:}\mathbb{I}){:}\mathbb{T}\to\mathbb{B} & \triangleq & \underline{\forall} P' & \mapsto \mathrm{Free}_{h+1}\ (\uparrow_h i)\ P' \\ & | & \{ E' \bot P' \} & \mapsto \mathrm{Free}_h\ i\ E' \vee \mathrm{Free}_{h+1}\ (\uparrow_h i)\ P' \\ & | & \dot{\chi}_{i'} & \mapsto h{\leq}i' \wedge i'{=}i \\ & | & \ldots & \textit{(straightforward recursion)} \end{array}$$

The standard version *Free*^{S} just looks for a dangling index i, representing the i^{th} free variable, in a term T, and is expected to be called from the root of the term. When crossing a binder, as the λ-height increases, the index is incremented to represent the same free variable. But our modified version *Free*$_h$ uses the λ-height h as an additional (useless) parameter. These two versions differ only for ill-formed uses; indeed, if the invocation *Free*$_h\ i\ T$ is such that $h{<}i$, then it returns false – quite reasonably, we would say, as this means that the index i is in fact bound.

The generalisation of the λ-height parameter h is also illustrated for the substitution function; again we provide the standard version in first place, and the context aware version in second place:

Definition 8.1.11 (Standard substitution)

$$\begin{array}{lcll} [i\backslash E]^S{:}\mathbb{T}\to\mathbb{T} & \triangleq & \underline{\forall} P' & \mapsto \underline{\forall}([i{+}1\backslash\uparrow E]^S P') \\ & | & \{ E' \bot P' \} & \mapsto \{ [i\backslash E]^S E' \bot [i{+}1\backslash\uparrow E]^S P' \} \\ & | & \dot{\chi}_{i'} & \mapsto \textbf{if } i{=}i' \textbf{ then } E \textbf{ else } \dot{\chi}_{i'} \\ & | & \ldots & \textit{(straightforward recursion)} \end{array}$$

Definition 8.1.12 (BiCoq substitution)

$$\begin{array}{lcll} [i\backslash E]_h{:}\mathbb{T}\to\mathbb{T} & \triangleq & \underline{\forall} P' & \mapsto \underline{\forall}([\uparrow_h i\backslash\uparrow_h E]_{h+1} P') \\ & | & \{ E' \bot P' \} & \mapsto \{ [i\backslash E]_h E' \bot [\uparrow_h i\backslash\uparrow_h E]_{h+1} P' \} \\ & | & \dot{\chi}_{i'} & \mapsto \textbf{if } h{\leq}i \wedge i{=}i' \textbf{ then } E \textbf{ else } \dot{\chi}_{i'} \\ & | & \ldots & \textit{(straightforward recursion)} \end{array}$$

Again, the addition of the parameter h permits to use lifting instead of incrementation for the index i. Furthermore, if for both versions of substitution we lift the expression parameter E when crossing a binder, in the context aware version we use the exact λ-height parameter value h instead of the default value 0.

To understand why, let us consider the two approaches on an example:

Example 8.1.9 (Standard substitution and lifting)

$$[i\backslash E]^S \underline{\forall}\underline{\forall} P \quad = \quad \underline{\forall}[i{+}1\backslash \uparrow E]^S \underline{\forall} P \quad = \quad \underline{\forall}\underline{\forall}[i{+}2\backslash \uparrow\uparrow E]^S P$$

Example 8.1.10 (BiCoq substitution and lifting)

$$[i\backslash E]_0 \underline{\forall}\underline{\forall} P \quad = \quad \underline{\forall}[\uparrow_0 i\backslash \uparrow_0 E]_1 \underline{\forall} P \quad = \quad \underline{\forall}\underline{\forall}[\uparrow_1\uparrow_0 i\backslash \uparrow_1\uparrow_0 E]_2 P$$

As we have the following result:

Proposition 8.1.1 (Lifting composition)

$$h' \leq h \Rightarrow \uparrow_{h'} (\uparrow_h T) = \uparrow_{h+1} (\uparrow_{h'} T)$$

that is with $h{=}h'$ $\quad \uparrow_h (\uparrow_h T) = \uparrow_{h+1} (\uparrow_h T)$

We can prove that both versions of substitution are extensionally equivalent for any well-formed invocation:

$$\begin{array}{rcl} \uparrow\uparrow\uparrow \ldots \uparrow E & = & \uparrow_0 (\uparrow_0 (\uparrow_0 \ldots (\uparrow_0 E))) \\ & = & \uparrow_1 (\uparrow_0 (\uparrow_0 \ldots (\uparrow_0 E))) \\ & = & \uparrow_1 (\uparrow_1 (\uparrow_0 \ldots (\uparrow_0 E))) \\ & = & \uparrow_2 (\uparrow_1 (\uparrow_0 \ldots (\uparrow_0 E))) \\ & = & \ldots \end{array}$$

In other words, applying h times the function $\uparrow$ yields exactly the same result as applying successively $\uparrow_0$, followed by $\uparrow_1$, ..., $\uparrow_{h-1}$. Therefore lifting the expression E from the λ-height h instead of 0 during recursion in the context aware version of the substitution does not change the final result.

We have indicated in Section 8.1.3 that *de Bruijn* levels appear to be simpler to use because the λ-height parameter is never required. *To some extent, our work on* de Bruijn *indexes shows that they are easier to deal with by ensuring that the λ-height parameter is always present.* This is why we systematically use the exact λ-height h instead of the default value 0 for example in the function *Abstr*, and why we generalize the λ-height h parameter to all operations, even those that do not require it, such as the substitution.

The intuitive justification about this statement is provided by considering typical commutation lemmas required in such developments, whose generic form is $f(g(T)){=}h(T)$ with f, g and h functions such as freeness, abstraction, lifting, substitution and so on. *With the standard version of such functions, problems arise when the λ-height parameter appears in some but not all of these three functions, and require side conditions as well as technical meaningless lemmas to be proven.* Our modified versions on the contrary explicitly introduces the λ-height parameter in all of them, ensuring for example its consistency between the two sides of the equality during induction steps.

Our estimate using this approach is that about 90% of the side conditions to avoid ill-formed calls can be removed. This is further explored and illustrated in Section 8.4.5, dealing with the same concepts and optimisations, but applied to a simpler calculus.

8.1.6 Representing application

In a deep embedding of a λ-calculus it is standard to represent the application directly in the syntax as a term constructor; the embedded syntax therefore includes terms such as $(\lambda\, x{\cdot}T)\, T_a$. The β-reduction can then for example be represented as a relation between syntactical terms, relying on external operations:

Example 8.1.11 (Syntactical application and functional reduction)

$$(\lambda\, x{\cdot}T)\, T_a \rightarrow_\beta [x\backslash T_a]T$$

It is indeed an external operation, as $[x\backslash T_a]T$ does not denote a syntactical term of the language but the invocation of an external operation, the meta-language substitution, whose computation produces a term of the language.

The β-reduction can also be defined as a relation between syntactical terms without relying on external operations. The *explicit substitution* approach [ACCL91, CHL96] for example defines a syntactical sort for substitutions (such as $x\backslash E$) and a construct $[S]T$, S being a syntactical substitution, T and $[S]T$ being syntactical terms of the language.

Both types of approaches are interesting for example to study the influence of normalisation strategies, yet this is not relevant in our case, as our objectives with regard to the B are not related to such meta-theoretical studies.

Therefore, we choose to represent application not as a constructor of the language, and in fact more generally we do not provide any way to represent application. The immediate consequence is that our language for the syntax of the B logic only represents terms in normal form ; furthermore we can not compare various reduction strategies for the guest language, as we have to rely on the reduction strategy of the host language – here Coq.

Instead of having an application constructor, we encode the binder elimination rules as external operations that represent the application followed by β-reduction – a form of normalization by evaluation [Lin05, AHN08, Boe10]. *It is interesting to note that we are mixing deep and shallow approaches*: we represent B terms and B proofs as terms in Coq, but B applications are in fact embedded as the invocations of Coq functions on B terms, that is we transform a B redex into a Coq redex whose reduction produces B terms.

We define a functional application for each B binder as follows:

Definition 8.1.13 (BiCoq functional applications)

$$T @_{\forall} E \quad \triangleq \quad \mathbf{match}\, T\, \mathbf{with}\, \underline{\forall}\, T' \mapsto App_0\, E\, T'$$

$$T @_{\{\}} E \quad \triangleq \quad \mathbf{match}\, T\, \mathbf{with}\, \{E' \mid T'\} \mapsto E \dot{\in} E' \dot{\wedge} App_0\, E\, T'$$

where App *is the following function:*

$$\begin{array}{lll}
\mathrm{App}_h(E{:}\mathbb{E}){:}\mathbb{T}\rightarrow\mathbb{T} \triangleq & & \\
\quad \mid \underline{\forall} P' & \mapsto & \underline{\forall}(\mathrm{App}_{h+1}\, (\uparrow_h E)\, P') \\
\quad \mid \{ E' \mid P' \} & \mapsto & \{ \mathrm{App}_h\, E\, E' \mid \mathrm{App}_{h+1}\, (\uparrow_h E)\, P' \} \\
\quad \mid \dot{\chi}_{i'} & \mapsto & \begin{cases} \dot{\chi}_{i'-1} & \text{if } h < i' \\ E & \text{if } h = i' \\ \dot{\chi}_{i'} & \text{if } h > i' \end{cases} \\
\quad \mid \ldots & & \textit{(straightforward recursion)}
\end{array}$$

The functional applications only apply to terms starting with the appropriate binder – the partiality is encoded in Coq *by an additional proof parameter left implicit here. The interesting aspect is that we factorise the common underlying process in the function* App.

The functional abstractions, the substitution function and the functional applications are such that the following properties hold (the first one being valid only after generalising the λ-height parameter to the substitution function):

Proposition 8.1.2 (Substitution as a composite operation)

$$[i\backslash E]_h T = App_h\ E\ (Abstr_h\ i\ T)$$

or more simply $[i\backslash E]T = (\dot\forall i\cdot P)@_\forall E$

Thanks to this property of our calculus, the elimination rules of the B logic can be rewritten with a slightly different presentation:

Proposition 8.1.3 (Alternative presentation of binder elimination rules)

$$\frac{\Gamma \dot\vdash \dot\forall i\cdot P}{\Gamma \dot\vdash (\dot\forall i\cdot P)@_\forall E} \qquad\qquad \frac{\Gamma \dot\vdash E \dot\in \{i:S \mid P\}}{\Gamma \dot\vdash (\{i:S \mid P\})@_{\dot\cup} E}$$

As indicated, the standard definition of β-reduction in λ-calculus is $\lambda x\cdot T@E \rightarrow_\beta [x\backslash E]T$; it describes the semantics of application using substitution. In our embedding, on the contrary, application is directly defined (as an external operation) whereas the substitution is a composite operation. *The point is that beyond considering substitution as an external operation, it is not even primitive in our calculus.*

Note also that we can write $App_h\ \dot\chi_i\ (Abstr_h\ i\ T) = T$, or more simply at λ-height 0 and using the functional universal quantification $(\dot\forall i\cdot P)@_\forall \dot\chi_i = P$, to emphasise that application at $\dot\chi_i$ is the "inverse" of abstraction at i. Furthermore this result commutes, that is:

Proposition 8.1.4

$$h \leq i \quad\Rightarrow\quad \mathrm{Free}_h\ i\ (\underline\forall T) = \bot \quad\Rightarrow\quad \mathrm{Abstr}_h\ i\ (\mathrm{App}_h\ \dot\chi_i\ T) = T$$

This last property gives a method to build a term in the functional representation which is equal to a term in the raw representation, that is for universal quantification we have:

Proposition 8.1.5 (Inverse universal quantification)

$$\mathrm{Free}_0\ i\ (\underline\forall P) = \bot \quad\Rightarrow\quad \underline\forall P = \dot\forall i\cdot(\mathrm{App}_0\ \dot\chi_i\ P)$$

The combination of the two properties about functional application and functional abstraction also justifies the fact that our *de Bruijn* representation is indeed α-quotiented, that is for universal quantification:

Proposition 8.1.6 (α-renaming for universal quantification) *Two α-equivalent terms are structurally equal in the* de Bruijn *representation:*

$$\mathrm{Free}\ i\ P = \bot \quad\Rightarrow\quad \dot\forall i\cdot[j\backslash i]P = \dot\forall j\cdot P$$

Indeed, we can develop the expression:

$$\dot\forall i\cdot[j\backslash i]P \quad=\quad \dot\forall i\cdot(\mathrm{App}_0\ \dot\chi_i\ (\mathrm{Abstr}_0\ j\ P)) \quad=\quad \underline\forall \mathrm{Abstr}_0\ i\ (\mathrm{App}_0\ \dot\chi_i\ (\mathrm{Abstr}_0\ j\ P))$$

Then, noting that $\mathrm{Free}\ i\ P = \bot \Rightarrow \mathrm{Free}_0\ i\ (\underline\forall \mathrm{Abstr}_0\ j\ P) = \bot$*, we simplify as follows:*

$$\underline\forall \mathrm{Abstr}_0\ i\ (\mathrm{App}_0\ \dot\chi_i\ (\mathrm{Abstr}_0\ j\ P)) \quad=\quad \underline\forall \mathrm{Abstr}_0\ j\ P \quad=\quad \dot\forall j\cdot P$$

8.2 Proving by induction

As mentioned in Section 3.2, the definition of an inductive datatype in Coq yields automatically the associated structural induction principle.

When considering B terms (that is more precisely $\mathbb{P}$ and $\mathbb{E}$), this structural induction principle is relevant to prove structural properties such as those about freeness (cf. the definition of *Free* in the previous paragraph), but it is hopelessly inadequate to prove semantic results, *i.e.* results dealing with statements of the B logic instead of predicates and expressions. To be accurate, all induction principles are equivalent, being derivable from each other; yet in practice some induction principles are better than other for proving specific properties.

Indeed, the inductive definition of $\mathbb{P}$ presented in Section 7.2.1 includes the constructor definition $\underline{\forall}:\mathbb{P}\rightarrow\mathbb{P}$. It indicates how to build a new term $\underline{\forall}P:\mathbb{P}$ using an existing term $P:\mathbb{P}$, but also identifies P as the structural predecessor of $\underline{\forall}P$ when using structural induction in a proof. In other words, proving that a property $\mathcal{Q}$ over $\mathbb{P}$ is valid for any predicate by structural induction requires proving a subgoal of the form $\forall\,(P:\mathbb{P}),\ \mathcal{Q}(P)\Rightarrow\mathcal{Q}(\underline{\forall}P)$.

However, it should be clear that with a *de Bruijn* representation this approach is not always appropriate, as illustrated thereafter:

Example 8.2.1 (Structural predecessors in *de Bruijn* representation)

de Bruijn *representation*	$\underline{\exists}(1*0>2)$	$\underline{\forall}(\underline{\exists}(1*0>2))$
Natural representation	$\exists\, z\cdot X_0*z>X_1$	$\forall\, y\cdot\exists\, z\cdot y*z>X_0$

The two de Bruijn *terms are related structurally, the term on the right side being the quantified version of the term of the left side. However, considering the associated natural representations, they are not related semantically – intuitively because of the changes of the λ-height, causing unmonitored shifts of the context modifying free variables representation.*

To address this problem and some others, numerous induction principles have been derived in the first version of our embedding: (weak) structural induction, semantic induction, strong induction based on a measure for a given type or for mutually recursive types. And this was not yet sufficient for induction on B proofs, because the predecessors – that is the sub-proofs – of a step in a proof have different (dependent) types. This was not considered as a proper approach, due to the number of principles to be expressed and proved as well as the absence of genericity of the proof method.

We have therefore designed a more general approach, which is furthermore relatively intuitive. It may also have the potential to help for a better automation of proofs in BiCoq, e.g. using Ltac, the Coq tactic languages. It combines a single strong induction principle based on a measure in $\mathbb{N}$ (this is the intuitive part) with a strategy for conducting the proof defined through an inductive relation (the so-called *accessibility relation*).

The strong induction principle on $\mathbb{N}$ is unique and very generic. Indeed D is any family of types indexed by any type T, (*i.e.* for any $t:T$, $D(t)$ is a type), M any measure on this family and $\mathcal{Q}$ any predicate on this family:

Proposition 8.2.1 (Strong induction principle on a measure)

$$\forall\,(T:\mathbf{Type})(D:T\rightarrow\mathbf{Type})(M:\forall\,(t:T),\ D\,t\rightarrow\mathbb{N})(\mathcal{Q}:\forall\,(t:T),\ D\,t\rightarrow\mathbf{Prop}),$$

$$(\forall\,(t:T)(d:D\,t),\ (\forall\,(t':T)(d':D\,t'),\ M(t')<M(t)\Rightarrow\mathcal{Q}(t')\Rightarrow\mathcal{Q}(t)))\Rightarrow\forall\,(t:T),\ \mathcal{Q}(t)$$

For any inductive proof in BiCoq, the idea is to select an appropriate accessibility relation $\mathcal{A}$ and to select or define a measure M compatible with this relation, that is a measure which is such that the following property holds:

$$Pred_{\mathcal{A}}\,(t_1, t_2, \ldots, t_n)\,t' \Rightarrow M(t_i) < M(t')$$

In other words, the measure M has to be such that $t_1, t_2, \ldots, t_n$ (the predecessors of t' according to the accessibility relation $\mathcal{A}$) are strictly smaller than t'. The accessibility relation $\mathcal{A}$ is therefore only used to make a case reasonning, and the strong induction principle on $\mathbb{N}$ allows for the derivation of a result for a term by using the result for all its predecessors.

Choosing the relation $\mathcal{A}$ is choosing the strategy, the cases in a proof by cases, the predecessors for the entity that we are considering. Intuitively, it defines paths to reach terms in D, and provided the measure M is compatible with the relation it allows to derive proofs along these paths.

The accessibility relation can be surjective or not in D; in the later case it defines a strict subset of accessible terms and can be used to prove that any term of this subset satisfies a property. Note that we do not really need an accessibility relation as such, but rather a *destruction principle*, to express properties of the form:

$$\forall\,(t{:}T)(d{:}D(t)),\ \mathcal{A}(d) \Rightarrow d = e_1 \vee \ldots \vee d = e_n$$

$\mathcal{A}(d)$ defining the subset of accessible terms in the type $D(t)$ and $e_1, \ldots, e_n$ all the possible forms for a term in this subset, defining the cases for a proof by cases.

For example a semantically relevant strategy for the B logic has to show that any term is accessible following natural operators such as the function $\dot{\forall}$ instead of the raw *de Bruijn* universal quantifier $\underline{\forall}$. Such a strategy can be defined as follows:

Definition 8.2.1 (Semantic accessibility relation)

$$\begin{array}{l}
\textbf{Inductive}\ \Sigma_{Sem}{:}\mathbb{T}\to\textbf{Type}\triangleq \\
\mid \Sigma_{\chi}{:}\forall\,(i{:}\mathbb{I}),\ \Sigma_{Sem}\,\dot{\chi}_i \\
\mid \Sigma_{\forall}{:}\forall\,(P{:}\mathbb{P})(i{:}\mathbb{I}),\ \Sigma_{Sem}\,P\to\Sigma_{Sem}\,\dot{\forall}\,i{\cdot}P \\
\mid \Sigma_{\{\}}{:}\forall\,(P{:}\mathbb{P})(E{:}\mathbb{E})(i{:}\mathbb{I}),\ \Sigma_{Sem}\,P\to\Sigma_{Sem}\,E\to\Sigma_{Sem}\,\dot{\{}i{:}E \mid P\dot{\}} \\
\mid \ldots \textit{(straightforward induction)}
\end{array}$$

This relation is indeed surjective, *i.e.* we prove:

Proposition 8.2.2 (Semantic construction of $\mathbb{T}$)

$$\forall\,(T{:}\mathbb{T}),\ \Sigma_{Sem}(T)$$

To prove a property $\mathcal{Q}$ for any term T, it is possible to apply the generic induction principle (with M the standard depth function on B terms) and then to use this relation to make a proof by cases by destructing the Coq term $\Sigma_{\mathrm{Sem}}(T)$. The generated subgoals are then semantically relevant, that is considering for example the binder cases, we have to prove the following induction steps:

$$\forall\,(E'{:}\mathbb{E})(P'{:}\mathbb{P}),\ \mathcal{Q}\,E' \Rightarrow \mathcal{Q}\,P' \Rightarrow \mathcal{Q}\,\dot{\{}i'{:}E' \mid P'\dot{\}}$$

$$\forall\,(P'{:}\mathbb{P})(i'{:}\mathbb{I}),\ \mathcal{Q}\,P' \Rightarrow \mathcal{Q}\,(\dot{\forall}\,i'{\cdot}P')$$

8.3 Grafting, congruence and namespaces

In Section 7.5.2, we present new congruence results proven to be valid for the B logic, related to the replacement of bound subterms. Using the standard concepts of the B logic, e.g. the meta-language substitution, such a replacement is not possible. Indeed, substitution prevents captures of free variables.

In the natural notation λ_V of the B method, the absence of captures is ensured by requiring appropriate α-renamings when substitution crosses a binder. In the *de Bruijn* representation used in BiCoq, the absence of capture is a mechanical consequence of the lifting applied to the replaced variable and the replacing expression when crossing a binder (cf. Sections 8.1.2 and 8.1.5).

This is a standard approach, but it also strongly limits the interest of the associated standard congruence rules. Consider the *Leibniz*' rule:

Example 8.3.1 (*Leibniz*' rule)

$$\frac{\Gamma \vdash E = F \quad \Gamma \vdash [V := E]P}{\Gamma \vdash [V := F]P}$$

We cannot use this rule directly to prove results such as the following ones:

Example 8.3.2 (Bound subterms)

$$\vdash \forall x \cdot x \in \mathbb{N} \Rightarrow x * 0 \leq 0 \qquad \vdash \forall x \cdot x \leq y \Leftrightarrow \neg\neg(x \leq y)$$

In the first case, knowing that $x*0=0$ (for any x), we have no way to substitute $x*0$ by 0 because x is bound. In the second case, knowing that $P \Leftrightarrow \neg\neg P$, similarly we cannot use the Leibniz*' rule for predicates (cf. Section 7.5.1) because P is here instantiated by $x \leq y$ in which x is bound.*

Of course, these results can be proven, but this requires at some point an elimination of the binders to be able to apply the Leibniz*' rules using capture-avoiding substitution.*

To derive more powerful congruence rules applicable to our two examples, we therefore need a special operator representing a form of substitution allowing for captures. The idea is to create an arbitrary Coq function $f : \mathbb{T} \to \mathbb{T}$ – in the sense that we provide the code of a function making unjustified modifications to a term – and to prove that a term T and its transformation $f(T)$ have a semantic relationship. That is, as we do not add any syntactical construct or any inference rule, we do not modify in any way the B logic, but we provide new tools for more efficient operations.

Note that the operation for replacing subterms whose free variables are bound by the context is a form of modified substitution; it does not aim at replacing a bound variable by an expression, but to replace a free variable by an expression in which some of the free variables can be captured by a parent binder. To understand why, consider once more the *Leibniz*' rule given herebefore. This rule, in practice, justify the replacement of the expression E_1 appearing in a term by the expression E_2, provided we have a proof that they are equal. The fact that we present this replacement through an intermediate pattern form $[V := \ldots]P$ is just a trick, and the variable V can be arbitrarily chosen. The situation is similar for the extended form of substitution that we want to define, and we can therefore restrict ourselves to the overloading of free variables.

8.3.1 A missed attempt: collapsing terms

Several attempts have been conducted to develop interesting congruence results, before finding an appropriate solution. We describe in this paragraph a nearly successful attempt, for the sake of completion; it was in fact extensively developped, before we were led to some tricky observations.

The idea was to define operations on terms as simple and atomic as possible – beyond a form of elegance, such an approach can considerably simplify associated proofs. Therefore, instead of defining a new capture-allowing substitution, we have tried to define a simpler function which, composed with other functions, was able to represent capture-allowing substitution. Our main experimentation with this approach was called *collapsing*, and denoted $\lightning$. In essence, it is a function replacing any occurrence of a given free variable by a given bound variable, or more precisely in our representation replacing any occurrence of a dangling index by a bound index. Such a function can be seen as a form of reverse of *Skolemisation*.

Whereas unusual, collapsing does not seem meaningless and is relatively simple. When trying to use this function to prove semantic results, for example the expected congruence rules, we identify interesting lemmas, such as this one:

$$(\Gamma \vdash E_1 \doteq E_2) \wedge i \backslash \Gamma \stackrel{?}{\Rightarrow} \Gamma \vdash \lightning_d^i(E_1) \doteq \lightning_d^i(E_2)$$

This does not seem unreasonable; indeed, the condition $i\backslash\Gamma$ requires the variable i to not appear (free) in Γ. It is therefore equivalent to a universally quantified variable, and collapsing it, that is in a way instantiating it so that it is captured, should not be a problem. Being unable to prove this lemma (in the absence of any convincing strategy), we have generalized it to the following form:

$$(\Gamma \vdash P) \wedge i \backslash \Gamma \stackrel{?}{\Rightarrow} \Gamma \vdash \lightning_d^i P$$

Unfortunately, we have identified for this last lemma different forms of counter-examples. Consider the following trivial result:

$$\vdash \forall x \cdot \exists y \cdot x \neq y$$

Eliminating the head binder, we got the equivalent statement $\vdash \exists y \cdot x \neq y$. Unfortunately, collapsing the free variable x so that it is captured by the existential quantifier leads to the false statement $\vdash \exists y \cdot y \neq y$. In fact, whereas x is indeed universally quantified, collapsing in this situation results into an inversion of the quantifiers, and intuitively breaks the causal dependencies between x and y.

Albeit the important investments into studying this solution, and the fact that such examples are refutation of one of the sufficient lemma but not of the congruence result itself, we have therefore decided to stop our investigations on collapsing, considering such subtle problems as non-intuitive and therefore misleading.

8.3.2 A (nearly) successful attempt: grafting

The collapsing approach being unsuccessful, we describe here a different approach, this time successful with regard to our objectives, that is the development of a specific operator for the replacement of subterms whose free variables are bound by the context and the proof of associated semantic results.

We define a special operator, called *grafting*, which compared to the standard substitution allows for the capture of variables in its parameter E by never lifting it:

Definition 8.3.1 (Grafting)

$$
\begin{array}{lcll}
[i\lhd E]_h : \mathbb{T}\to\mathbb{T} & \triangleq & \underline{\forall} P' & \mapsto \ \underline{\forall}([\uparrow_h i\lhd E]_{h+1}P') \\
& | & \{\!| E' \perp T' |\!\} & \mapsto \ \{\!| [i\lhd E]_h E' \perp [\uparrow_h i\lhd E]_{h+1}P' |\!\} \\
& | & \dot{\chi}_{i'} & \mapsto \ \textbf{if } h\leq i' \wedge i'=i \textbf{ then } E \textbf{ else } \dot{\chi}_{i'} \\
& | & \ldots & \textit{(straightforward recursion)}
\end{array}
$$

As mentioned, grafting does not permit to replace a bound variable – we have kept in the grafting function the guard $h \leq i'$ when applying to a variable, inherited from the substitution function. Grafting however allows for the replacement of a free variable by an expression whose free variables can be captured.

Note that, compared to the standard substitution, for grafting the transformation of a term T into a pattern, that is the construction of terms S and T' such that $[i\lhd S]T'=T$, is very simple. Indeed, the subterm S in this pattern being never lifted during the computation of the result of grafting, it has exactly the form that it has in the context of T; there is no transformation required on its bound or dangling indexes.

This arbitrary function however causes rather strange transformations with regards to the indexes, that is the variables; consider the following example:

Example 8.3.3 (Illustration of grafting in $\lambda_{\mathbf{dBI}}$ and $\lambda_{\mathbf{V}}$)

$$
\begin{array}{lllll}
\lambda_{dBI}\ \textit{notation} & [\underline{3}\lhd\underline{0}+\underline{1}+\underline{2}]_0\,\underline{\forall}(\underline{4}\geq\underline{0}) & \to_\beta & \underline{\forall}(\underline{0}+\underline{1}+\underline{2}\geq\underline{0}) \\
\lambda_V\ \textit{notation} & [x_3\lhd x_0+x_1+x_2](\forall\, z\cdot x_3\geq z) & & \forall\, z\cdot z+x_0+x_1\geq x_0
\end{array}
$$

The first line illustrates the use of grafting, and the second line is the pretty-printing of the λ_{dBI} terms in λ_V. This shows that when computing grafting, the variable x_1 is transformed into x_0 and x_2 into x_1 – without any control.

Indeed, not lifting the expression when crossing a binder also means in our *de Bruijn* representation not managing the variable representations consistently with the context. We have therefore typical unmonitored shifts in the representations of variables, loosing trace of the semantics of the indexes.

We can however prove very interesting semantic properties (the proof is sketched in Section 8.3.4) using grafting as a description trick. These results uses a specific condition:

Definition 8.3.2 (Orthogonality condition) $\Gamma \perp T$ *is a condition requiring the proof environment* Γ *to have no common free variable with the term* T*.*

The congruence result for the grafting of expressions is:

Proposition 8.3.1 (Congruence rules for grafted expressions)

$$
\Gamma \dot{\vdash} E_1 \doteq E_2 \quad\Rightarrow\quad \Gamma \perp E_1 \doteq E_2 \quad\Rightarrow\quad \Gamma \dot{\vdash} [i\lhd E_1]P \Leftrightarrow [i\lhd E_2]P
$$

$$
\Gamma \dot{\vdash} E_1 \doteq E_2 \quad\Rightarrow\quad \Gamma \perp E_1 \doteq E_2 \quad\Rightarrow\quad \Gamma \dot{\vdash} [i\lhd E_1]E \doteq [i\lhd E_2]E
$$

Similarly provided the extension of the B syntax with propositional variables discussed in Section 7.5.1, it is also possible to define a predicate grafting function, and to derive similar congruence results for equivalent predicates:

Proposition 8.3.2 (Congruence rules for grafted predicates)

$$\Gamma \dot{\vdash} P_1 \Leftrightarrow P_2 \quad \Rightarrow \quad \Gamma \perp P_1 \Leftrightarrow P_2 \quad \Rightarrow \quad \Gamma \dot{\vdash} \langle a \lhd P_1 \rangle P \Leftrightarrow \langle a \lhd P_2 \rangle P$$

$$\Gamma \dot{\vdash} P_1 \Leftrightarrow P_2 \quad \Rightarrow \quad \Gamma \perp P_1 \Leftrightarrow P_2 \quad \Rightarrow \quad \Gamma \dot{\vdash} \langle a \lhd P_1 \rangle E \doteq \langle a \lhd P_1 \rangle E$$

The orthogonality condition directly results from the unmonitored shifts of context mentioned herebefore. It ensures that the unpredictable changes of the variables have no effect on the validity of the equality or of the equivalence; in essence, if the free variables for example of $E_1 \doteq E_2$ do not appear in Γ, that means that they are intuitively universally quantified, and therefore their capture should not cause a problem[3]. It is easy to exhibit a counterexample when not respecting the orthogonality condition:

Example 8.3.4 (Role of the orthogonality condition)

$$\dot{\chi}_0 \doteq \Omega, \dot{\chi}_1 \doteq \Omega \dot{\vdash} \dot{\chi}_0 \doteq \dot{\chi}_1 \quad \not\Rightarrow \quad \dot{\chi}_0 \doteq \Omega, \dot{\chi}_1 \doteq \Omega \dot{\vdash} [\underline{1} \lhd \dot{\chi}_0](\underline{\forall}\, \dot{\chi}_2 \doteq \dot{\chi}_0) \Leftrightarrow [\underline{1} \lhd \dot{\chi}_1](\underline{\forall}\, \dot{\chi}_2 \doteq \dot{\chi}_0)$$

that is $\dot{\chi}_0 \doteq \Omega, \dot{\chi}_1 \doteq \Omega \dot{\vdash} \dot{\chi}_0 \doteq \dot{\chi}_1 \quad \not\Rightarrow \quad \dot{\chi}_0 \doteq \Omega, \dot{\chi}_1 \doteq \Omega \dot{\vdash} (\underline{\forall}\, \dot{\chi}_0 \doteq \dot{\chi}_0) \Leftrightarrow (\underline{\forall}\, \dot{\chi}_1 \doteq \dot{\chi}_0)$

However this orthogonality condition limits the interest of these results. In their current form, they help to justify for example the simplification of propositional equivalences, e.g. $\dot{\neg}\dot{\neg} P$ into P, or the unfolding of a standard definitions. But we are not yet able to deal with predicate equivalences or conditional definitions such as:

Example 8.3.5 (Conditional definition for division)

$$y \neq 0 \Rightarrow (x/y) = \mathbf{max}\{z : \mathbb{N} \mid x \geq y * z\}$$

8.3.3 Introducing namespaces

To improve the congruence results and avoid this limitation, several approaches have been considered, such as using names (to adopt a locally nameless representation), marking *de Bruijn* indexes during grafting, and so on. We have finally designed a simpler solution, using *parameterised* de Bruijn *indexes*.

In its most general form, this representation describes free and bound variables by pairs (n, x), the first component $n : \mathcal{N}$ being the *namespace* and the second component $i : \mathbb{I}$ the index. Binders of the language are themselves parameterised by a namespace $n : \mathcal{N}$ in which they capture variables.

Namespaces can be seen as sorts, used to mark binders and indexes[4]. This has limited consequences on the complexity of the code of the various operations on terms, *e.g.* lifting is as well parameterised by a namespace and only modifies indexes in this namespace.

Note that this representation defines in fact a form of names: if there is no binder in a namespace n, a pair (n, x) always represents a free variable and can be considered as a name, being never subject to computations. The only required operation on such pairs is an equality check. The application of these principles is considered for a generic λ-calculus in Section 8.4.

We just discuss in the rest of this section the simplified approach adopted for BiCoq. We use the namespace set $\mathcal{N} \triangleq \mathbb{N}$, and all the B binders act implicitly in the dedicated namespace 0, the other namespaces $n{+}1$ being used for eternally free variables. Therefore, we define a new type for our indexes:

[3] The same type of intuitive explanation is invalid for collapsing, as illustrated in the previous paragraph.

[4] Two sorts of *de Bruijn* indexes are considered in [DL07] but for different reasons, each of the two binders of the defined language using its own space of *de Bruijn* indexes.

Definition 8.3.3 (BiCoq parameterised *de Bruijn* indexes)

$$\mathbb{I}_P = \mathbb{N} \times \mathbb{N}$$

The first natural value is the namespace, the second natural value is the index in this namespace.

We do not modify the binders $\underline{\forall}$ and $\underline{\{|\}}$: they do not need to be parameterised by a namespace because they only capture indexes in the namespace 0. The parameterised *de Bruijn* index $(n,i):\mathbb{I}_P$ is therefore dangling at the λ-height h iff $n \neq 0 \vee h \leq i$; consistently, lifting only modifies pairs of the form $(0,i)$ in a term, etc.

With these minor modifications, we can reconsider the expression and the proof of the congruence results of the previous paragraph. The new versions are as follows[5]:

Proposition 8.3.3 (Congruence rules for grafted expressions)

$$\Gamma \dot{\vdash} E_1 \doteq E_2 \quad \Rightarrow \quad \Gamma \perp_0 E_1 \doteq E_2 \quad \Rightarrow \quad \Gamma \dot{\vdash} [i \lhd E_1]P \dot{\Leftrightarrow} [i \lhd E_2]P$$

$$\Gamma \dot{\vdash} E_1 \doteq E_2 \quad \Rightarrow \quad \Gamma \perp_0 E_1 \doteq E_2 \quad \Rightarrow \quad \Gamma \dot{\vdash} [i \lhd E_1]E \doteq [i \lhd E_2]E$$

The difference with the similar congruence results of the previous paragraph is that the side condition $\perp_0$ is modified, and now precisely requires Γ and $E_1 = E_2$ to have no common free variable *in the namespace* 0.

The previous technical difficulty of unmonitored shifts is still there, but is now limited to the namespace 0. Remember that grafting, by not lifting its expression parameter when crossing a binder, does not maintain consistency between the λ-height and the variable representation. However, as all binders only operate in the namespace 0, we can guarantee the preservation of the semantics of the indexes in the other namespaces.

This is a purely technical trick: namespaces have no semantics. Yet provided we avoid using the namespace 0 for free variables, through an extended form of α-conversion, changing the name of the free variables, we got the full expressiveness of our results. In their most general form, they allow for *strong β-reduction*, unfolding of (conditional) definitions under a binder... or more generally any form of first-order rewriting.

It is interesting to note that [DHK00] addresses similar considerations, using explicit substitutions to finely describe and manage various phenomenas such as reductions and captures; thanks to the expressiveness of the chosen representation, precise and exact algorithms are developed. In our case on the contrary, the algorithms are very simple but their validity is limited by side conditions. The introduction of namespaces reduces the scope of these conditions, to the point that they become purely technical and trivial to satisfy. The simplicity of the algorithm also leads to rather long and complex proofs, as illustrated in the next paragraph.

8.3.4 Sketch of the congruence proof

Introduction

The different versions of the proofs of the congruence results, that is for grafting expressions or predicates, and with or without namespaces, are not significantly different. They are however rather long and technical, and we just present here a sketch of their structure.

[5]Parameterised *de Bruijn* indexes are only defined in the latest version of BiCoq, which at the date of redaction is not yet as complete as the first version. The results for expression grafting are proven, but not the results for predicate grafting. However we do not foresee any technical difficulty to adapt them.

We use semantic induction, that is induction over a measure in $\mathbb{T}\to\mathbb{N}$ associated with a destruction principle derived from the relation Σ_{Sem}, which is semantically relevant (cf. Section 8.2). However, working directly on the congruence result is doomed to fail; consider the induction step for the universal quantifier:

$$\Gamma \stackrel{\cdot}{\vdash} e_1 \doteq e_2 \quad \Rightarrow$$
$$\Gamma \perp_0 e_1 \doteq e_2 \quad \Rightarrow$$
$$\Gamma \stackrel{\cdot}{\vdash} [j\lhd e_1]p \dot{\Leftrightarrow} [j\lhd e_2]p \quad \Rightarrow \quad \Gamma \stackrel{\cdot}{\vdash} [j\lhd e_1](\dot\forall\, i\cdot p) \dot{\Leftrightarrow} [j\lhd e_2](\dot\forall\, i\cdot p)$$

To use the induction hypothesis $\Gamma \stackrel{\cdot}{\vdash} [i\lhd e_1]p \dot{\Leftrightarrow} [i\lhd e_2]p$ we need to be able to commute grafting and universal quantification, that is grafting and abstraction (cf. Section 8.1.5). In other words, we need to transform a term such as $[j\lhd e_1](\dot\forall\, i\cdot p)$ into a term of the form $\dot\forall i'\cdot[j'\lhd e_1']p'$. However, we have not been able to discover any relevant way to commute these two operations without unacceptable side effects.

As it is often the case using induction, we have therefore decided to prove a more generic result – whose expression requires the definition and the formalisation of parallel substitutions, similar to those of λ-calculus.

A theory of parallel substitutions

There are two main properties justifying the introduction of parallel substitution. First, lifting and abstraction as well as any other standard operation on terms – with the notable exception of grafting – can be emulated by the application of specific parallel substitutions. Second, applying two parallel substitutions to a term is equivalent to applying a third parallel substitution, computable from the two other ones. Such properties, of course, have to be proven, and are in fact associated to a rather extensive formalisation of the parallel substitutions in BiCoq – even if they have no real operational interest, being mainly used as a proof tool for the congruence results[6].

In the first version of our embedding, parallel substitutions were defined as lists of pairs $(i, E) : \mathbb{I}_P \times \mathbb{E}$ and associated to a dedicated functional application of such parallel substitutions to terms. Note that such lists are well-formed only provided that there are never two pairs (i, E) and (i, E') s.t. $E \neq E'$, that would represent divergent substitutions.

On the contrary, our latest version uses maps to represent parallel substitutions, that is we define the type of parallel substitutions as follows:

Definition 8.3.4 (Parallel substitutions)

$$\mathbb{M} \quad \triangleq \quad \mathbb{I}_P \to \mathbb{E}$$

This approach has several merits that deserve consideration. The first one is that this representation of parallel substitutions does not contains ill-formed entities – the problem of divergent substitutions simply does not exist here – nor does it introduce artificial properties, such as a meaningless order between the pairs in an association list. Furthermore, we do not have to deal with functions looking for a value or the numerous theorems about manipulations of such lists.

But the most interesting property is that maps represent infinite parallel substitutions. Consider for example lifting: applied to a term, it increments all dangling indexes. Using (finite) lists of pairs in $\mathbb{I}_P \times \mathbb{E}$, we cannot define a parallel substitution whose application emulate lifting for any term; we can only prove that for any term there is a parallel substitution emulating lifting. On the contrary, parallel substitutions represented by (infinite) maps allow for a generic substitution emulating lifting for any term.

[6] However we use the same tools to embed the Gsl in Section 7.3.2.

We first define the following constants and functions[7]:

Definition 8.3.5 (Neutral parallel substitution)

$$\odot : \mathbb{M} \quad \triangleq \quad \mathbf{fun}\, i' : \mathbb{I}_P \mapsto i'$$

Definition 8.3.6 (Overloading of a parallel substitution)

$$m \oplus (i \backslash E) : \mathbb{M} \quad \triangleq \quad \mathbf{fun}\, i' : \mathbb{I}_P \mapsto \mathbf{if}\, i = i'\, \mathbf{then}\, E\, \mathbf{else}\, m\, i'$$

Definition 8.3.7 (Lifting of a parallel substitution)

$$\begin{array}{lll} \uparrow_h m : \mathbb{M} & \triangleq & \mathbf{fun}\, i' : \mathbb{I}_P \mapsto \mathbf{match}\, i'\, \mathbf{with} \\ & & \quad \mid (0,0) \mapsto i' \\ & & \quad \mid (0, j'+1) \mapsto \mathbf{if}\, h \leq j'\, \mathbf{then}\ \uparrow_h (m\,(0,j'))\, \mathbf{else}\, i' \\ & & \quad \mid \_ \mapsto \uparrow_h (m\, i') \end{array}$$

Definition 8.3.8 (Application of a parallel substitution to a term)

$$\begin{array}{lllll} [[m]]_h : \mathbb{T} \to \mathbb{T} & \triangleq & \underline{\forall} P' & \mapsto & \underline{\forall}([[\uparrow_h m]]_{h+1} P') \\ & \mid & \{\!\!\{ E' \bot T' \}\!\!\} & \mapsto & \{\!\!\{ [[m]]_h E' \bot [[\uparrow_h m]]_{h+1} P' \}\!\!\} \\ & \mid & \dot{\chi}_{(n',i')} & \mapsto & \mathbf{if}\, n' \neq 0 \vee h \leq i'\, \mathbf{then}\, m\, i'\, \mathbf{else}\, \dot{\chi}_{(n',i')} \\ & \mid & \ldots & & \textit{(straightforward recursion)} \end{array}$$

The last function represents the application of a parallel substitution to a term; as for any other operations in our *de Bruijn* representation, we systematically explicit the λ-height parameter h, and we use it as precisely as possible (for example to lift the parallel substitution parameter in the recursive calls when crossing a binder).

As indicated, it is then possible to define parallel substitutions whose application to a term emulate standard terms operations:

Definition 8.3.9 (Parallel substitution emulating substitution)

$$M_h^{\backslash}(i : \mathbb{I}_P)(E : \mathbb{E}) : \mathbb{M} \quad \triangleq \quad \mathbf{fun}\, (n', i') : \mathbb{I}_P \mapsto \mathbf{if}\, (n' \neq 0 \vee h \leq i') \wedge i = (n', i')\, \mathbf{then}\, E\, \mathbf{else}\, (n', i')$$

Proposition 8.3.4 (Substitution emulation)

$$[i \backslash E]_h\, T \quad = \quad [[M_h^{\backslash}\, i\, E]]_h T$$

Definition 8.3.10 (Parallel substitution emulating lifting)

$$M_h^{\uparrow} : \mathbb{M} \quad \triangleq \quad \mathbf{fun}\, i' : \mathbb{I}_P \mapsto \uparrow_h i'$$

Proposition 8.3.5 (Lifting emulation)

$$\uparrow_h T \quad = \quad [[M_h^{\uparrow}]]_h T$$

Definition 8.3.11 (Parallel substitution emulating abstraction)

$$M_h^{\mathrm{Abstr}}(i : \mathbb{I}_P) : \mathbb{M} \quad \triangleq \quad \mathbf{fun}\, (n', i') : \mathbb{I}_P \mapsto \mathbf{if}\, (n' \neq 0 \vee h \leq i') \wedge i = (n', i')\, \mathbf{then}\, (0, h)\, \mathbf{else}\ \uparrow_h\ (n', i')$$

[7] For the sake of readability we do not distinguish here the index i and the variable $\dot{\chi}_i$.

Proposition 8.3.6 (Abstraction emulation)

$$\mathrm{Abstr}_h\, i\, T \quad = \quad [[M_h^{\mathrm{Abstr}}\, i]]_h T$$

Definition 8.3.12 (Parallel substitution emulating application)

$$M_h^{\mathrm{App}}(E{:}\mathbb{E}){:}\mathbb{M} \quad \triangleq \quad \mathbf{fun}\, i'{:}\mathbb{I}_P \mapsto \begin{cases} (0, j'-1) \ \textit{if}\ i=(0,j') \ \textit{with}\ h<j' \\ E \ \textit{if}\ i=(0,h) \\ i' \ \textit{in any other case} \end{cases}$$

Proposition 8.3.7 (Application emulation)

$$\mathrm{App}_h\, E\, T \quad = \quad [[M_h^{\mathrm{App}}\, E]]_h T$$

Note that the λ_{dBL} representation does not enjoy these properties. Indeed lifting for example in λ_{dBL} increments all levels, bound or dangling, and therefore cannot be emulated by the application of a parallel substitution (only affecting free variables, that is dangling levels, and unable to change a bound level). This is the reason why we have preferred to use the λ_{dBI} representation (cf. Section 8.1.3).

We have also numerous associated properties, for example the distribution of lifting over the application of a parallel substitution:

Proposition 8.3.8 (Distribution of lifting)

$$\uparrow_h ([[m]]_h T) \quad = \quad [[\uparrow_h m]]_{h+1} \uparrow_h T$$

Such properties are of course reminescent of results of the explicit substitution calculus.

The second important aspect for the congruence proof about parallel substitutions is that there is a composition law, that is multiple applications of parallel substitutions is equivalent to the application of a single parallel substitution:

Definition 8.3.13 (Composition of parallel substitutions)

$$m_1 \circledcirc_h m_2{:}\mathbb{M} \quad \triangleq \quad \mathbf{fun}\, i'{:}\mathbb{I}_P \mapsto [[m_1]]_h (m_2\, i')$$

Proposition 8.3.9 (Validity of composition)

$$[[m_1]]_h([[m_2]]_h T) \quad = \quad [[m_1 \circledcirc_h m_2]]_h T$$

Note again that even for composition, we explicit and use the λ-height parameter.

Tackling infinity

As indicated, one of the main interests of maps is the ability to represent infinite substitutions, for example to have a generic parallel substitution emulating lifting. On the other hand, maps require additional theorems that may be complex to deal with as $\mathbb{I}\to\mathbb{E}$ is not well-founded. This for example forbids the use of an induction principle.

Fortunately, we are not interested in BiCoq by properties of maps, but by properties about the application of parallel substitutions to terms – a point that makes a significant difference. For example, the following semantic property indeed deals with the application of a parallel substitution, and is an important lemma for our congruence proof:

Proposition 8.3.10 (Semantic lemma for parallel substitutions)

$$\Gamma \dot{\vdash} P \quad \Rightarrow \quad \forall\,(m{:}\mathbb{M}),\ (\forall\,(i{:}\mathbb{I}_P),\ m\,i{=}i \vee i\backslash(\Gamma \dot{\vdash} P)) \quad \Rightarrow \quad \Gamma \dot{\vdash} [[m]]P$$

That is, if P is provable under the assumptions Γ in the B *logic, and if m is a parallel substitution that only modifies variables that do not appear free in both P and Γ, then $[[m]]P$ is provable as well under the same assumptions.*

Intuitively, if a variable does not appear free in both P and Γ, it can be universally quantified and then instantiated for any value, that is any change by a parallel substitution is acceptable. This condition is called the *compatibility condition* between m and $\Gamma \dot{\vdash} P$.

When considering the application of parallel substitutions to terms, the infiniteness of maps is irrelevant because there is only a finite number of dangling indexes in a term that can be affected. Using this idea, we define in BiCoq scoped application of a parallel substitution as an application whose effects are limited to indexes listed in a list. It is then possible to prove a form of induction principle on these lists, and to finally derive the following result:

Proposition 8.3.11 (Induction-like principle for parallel substitutions)

$$\begin{aligned}&\forall\,(T{:}\mathbb{T})(P{:}\mathbb{T}\rightarrow\mathbf{Prop}),\quad P(T) \Rightarrow \\ &\qquad (\forall\,(m:\mathbb{M}),\ P([[m]]T) \Rightarrow \forall\,(i:\mathbb{I})(E:\mathbb{E}), P([[m \oplus (i\backslash E)]]T)) \Rightarrow \\ &\qquad (\forall\,(m:\mathbb{M}),\ P([[m]]T).\end{aligned}$$

If a property is true for a term T (in other words for $[[\odot]]T$), and if provided the property is true for $[[m]]T$ then we can prove that it is also true for $[[m \oplus (i\backslash E)]]T$, then the property is true for $[[m]]T$, for any m.

This looks like an induction principle, but it is not complete with regard to maps. Using only $\odot$ and $\oplus$ as constructors, the only accessible maps are those that have a finite scope, that is those having a finite (yet arbitrary high) number of indexes for which the substitution is not the identity. Maps such as $M_h^{\uparrow}$ are therefore not accessible. Yet we are not trying to prove a property on maps but on parallel substitutions applied to a term; the principle is valid because for any term T and any map M, there is a map M' whose scope is finite and such that $[[M']]T = [[M]]T$. In other words, the set of maps is not well-founded, but the set of relevant maps is.

We just discuss a few last remarks about maps. First, whereas maps are implementations for example in ML-like languages, they are generally considered to be inefficient. But here maps are a representation for parallel substitutions that are just used as a proof tool in BiCoq. That is, for example, the proven prover discussed in Section 7.4 may have B tactics for the application of the congruence results without having to implement parallel substitutions. Second, before choosing to use maps all the consequences should be carefully analysed. For example, they cannot be analysed extensionally; it is not possible, given $m{:}\mathbb{M}$, to check whether it has a finite scope or not.

Generalisation of the congruence results

Now that we have a theory about parallel substitutions and some semantic results, we can return to our initial objective, that is the proof of the congruence results. In its local and generic form for the grafting of expressions, the extended congruence result with parallel substitutions is as follows:

Proposition 8.3.12

$$\forall\,(\Gamma : \mathbf{list}\,\mathbb{P})(E_1\ E_2 : \mathbb{E}),\ \Gamma \dot{\vdash} E_1 \doteq E_2 \quad \Rightarrow \quad \Gamma \perp_0 E_1 \doteq E_2 \quad \Rightarrow$$

$$\forall\,(T : \mathbb{T})(i : \mathbb{I}_P)(m : \mathbb{M}),\ (\forall\,(n'\ i' : \mathbb{N}),\ m\,(n'+1, i') = (n+1, i') \vee (n'+1, i') \backslash (\Gamma \dot{\vdash} E_1 \doteq E_2)) \quad \Rightarrow$$

match T **with**
$|\ \mathrm{Trm\_of\_Prd}\ P' \rightarrow \Gamma \dot{\vdash} [[m]]([i \lhd E_1]P' \dot{\Leftrightarrow} [i \lhd E_2]P')$
$|\ \mathrm{Trm\_of\_Exp}\ E' \rightarrow \Gamma \dot{\vdash} [[m]]([i \lhd E_1]E' \doteq [i \lhd E_2]E')$

The main difference with the simpler congruence result that we were initially trying to prove is the universal quantification over $\mathbb{M}$. The former can be derived from the latter using the parallel substitution $\odot$ which satisfies the compatibility condition with $\Gamma \dot{\vdash} E_1 \doteq E_2$.

The proof is done by using the semantic induction principle derived from Σ_{Sem}. We only discuss two illustrative cases. When the term is a variable, and more specifically the grafted variable, the goal becomes:

$$\Gamma \dot{\vdash} [[m]](E_1 \doteq E_2)$$

This is proven using the semantic result about parallel substitutions – which is indeed applicable because of the compatibility condition on m.

The universal quantification case is much more tricky; we have to prove:

$$\Gamma \dot{\vdash} [[m]]([i \lhd E_1]\dot{\forall}\, i' \cdot P \dot{\Leftrightarrow} [i \lhd E_2]\dot{\forall}\, i' \cdot P) \qquad \text{that is}$$

$$\Gamma \dot{\vdash} \underline{\forall}\,[[\uparrow_0 m]]_1[\uparrow_0 i \lhd E_1]_1 Abstr_0\, i'\ P \quad \dot{\Leftrightarrow} \quad \underline{\forall}\,[[\uparrow_0 m]]_1[\uparrow_0 i \lhd E_2]_1 Abstr_0\, i'\ P$$

We can use the inverse universal quantification result at the end of Section 8.1.6 to construct a functional representation for this term; the goal then becomes:

$$\Gamma \dot{\vdash} \dot{\forall}\, i'' \cdot App_0\, i''\ [[\uparrow_0 m]]_1[\uparrow_0 i \lhd E_1]_1 Abstr_0\, i'\ P \dot{\Leftrightarrow} \dot{\forall}\, i'' \cdot App_0\, i''\ [[\uparrow_0 m]]_1[\uparrow_0 i \lhd E_2]_1 Abstr_0\, i'\ P$$

Thanks to the reintroduction of a form of natural representation – the functional representation – we can apply a standard B semantic result to purely and simply eliminate the universal quantification:

$$\Gamma \dot{\vdash} App_0\, i''\ [[\uparrow_0 m]]_1[\uparrow_0 i \lhd E_1]_1 Abstr_0\, i'\ P \dot{\Leftrightarrow} App_0\, i''\ [[\uparrow_0 m]]_1[\uparrow_0 i \lhd E_2]_1 Abstr_0\, i'\ P$$

This is an important step, the disparition of the quantifier corresponding to the reduction of the depth (the measure) of our term, and therefore allowing for a later application of the induction hypothesis. Through various simplifications and equalities, as well as the transformation of the application into a parallel substitution followed by its composition with m, we build a parallel substitution m' (not detailed here) and have to prove:

$$\Gamma \dot{\vdash} [[m']]([\uparrow_0 i \lhd E_1] Abstr_0\, i'\ P \dot{\Leftrightarrow} [\uparrow_0 i \lhd E_2] Abstr_0\, i'\ P)$$

We then apply the induction hypothesis, and check that indeed we satisfy the associated conditions to conclude for this case.

This is a rather long and technical proof – its complexity resulting from the use of a *de Bruijn* representation combined with an oversimplistic vision of grafting – whose validity would be difficult to assess without the support of proof mechanisation. Yet once proven, this theorem can be used to build tactics in a prover without having to care about parallel substitutions; the use of a very simple grafting function then becomes relevant.

8.4 The $\lambda_{\mathbf{TdB}}$ notation

We have introduced in Section 8.1 numerous concepts and optimisations related to the representation of terms in BiCoq. We summarise in this section these principles and apply them to the formalisation of a simpler λ-calculus, denoted λ_{TdB} (for Typed *de Bruijn*).

There are two very different forms of adaptations to consider. The first one is structural, and corresponds to the generalisation of the concept of namespaces, introduced in BiCoq to tackle complex problems when dealing with *de Bruijn* indexes for unusual operations such as grafting. This section explores a variation of this idea, using types, before reintroducing namespaces later.

The second one is just a form of optimisation of the theory by an explicit and precise management of the context in which term transformations operate. It is presented here as well, but this time with less justifications; please refer to the previous chapter about BiCoq to have the full details.

Our description is, at this stage, limited to simple results; the interest of this calculus is yet to be explored. In essence, this new representation (with its associated operations) does not bring any new result but seems to provide simpler ways to deal with complex situations. Note that unless indicated otherwise, the following definitions and propositions have been formalised in Coq.

8.4.1 Using types as part of variable identifiers

The classical notations of the λ-calculus are often described as *Church*'s or *Curry*'s representations, and differ in the way types are associated to terms; in both cases, variables are represented by a name. One of the underlying ideas of this section is that a variable can also be represented by the ordered pair composed of a name and a type.

The formal details of this representation (denoted λ_{TV}, for typed variables) are not given, as we are introducing later another representation with the same principles, but using *de Bruijn* indexes. We just provide here a few illustrations to explain some of the principles before introducing additional layers of complexity. In λ_{TV} a variable is a pair (σ, n) where σ is a type and n a name. That is, if $\sigma_1 \neq \sigma_2$, then (σ_1, n) and (σ_2, n) denote two different variables.

As usual, binders are parameterised by the variable they bind, that is here a pair. It is important to distinguish between the standard notation $\lambda x{:}\sigma{\cdot}L$, that represents the term L in which (free) occurrences of x are captured, the parameter σ indicating the assumed type for x, and our notation $\lambda\,(\sigma, x){\cdot}L$, that represents the term L in which (free) occurrences of (σ, x) are captured:

Example 8.4.1 (Comparing binders in $\lambda_{\mathbf{V}}$ and $\lambda_{\mathbf{TV}}$)

$$\begin{array}{lrcl} \lambda_V \textit{ notation} & X_0{:}\sigma' & \vdash & \lambda x{:}\sigma{\cdot}(x{+}\lambda y{:}\sigma'{\cdot}(x{-}y)\ X_0) \\ \\ \lambda_{TV} \textit{ notation} & & \vdash & \lambda(\sigma,x){\cdot}((\sigma,x){+}\lambda(\sigma',y){\cdot}((\sigma,x){-}(\sigma',y))\ (\sigma',X_0)) \end{array}$$

The representation of variables as pairs has several consequences. The most evident is that we do not need anymore to manage a typing context, as it is now embedded in the term itself[8]. But the real justification for this vision is provided in the rest of this section, when considering *de Bruijn* indexes; it is sufficient to say at this stage that we are in fact

[8] A similar approach is discussed in [GMW09], not yet published; it is apparently a recent and independent development, using a locally nameless representation with standard *de Bruijn* indexes, only named free variables being associated to a type.

providing a set of names per type, and that we are implicitly managing one context of names per type.

Note that the approaches presented here are expected to be used as internal representations. Indeed, an OCaml code adopting this type of convention would be considered as misleading – without even using a *de Bruijn* representation.

Example 8.4.2 (Typed identifiers in OCaml) *We can first consider a version in which type information would be inferred:*

```
let x = 1;;
let x = 'a';;
if 1<x then x else 'b';;
```

Interestingly, this code would be accepted, being well-typed. It is indeed equivalent to the following one, in which the types are explicit:

```
let (int,x) = 1;;
let (char,x) = 'a';;
if 1<(int,x) then (char,x) else 'b';;
```

8.4.2 The $\lambda_{\mathbf{TdB}}$ syntax

We now adapt the principle of typed variable identifiers to a *de Bruijn* representation with indexes; this can be seen as a generalisation of the concept of namespaces introduced in Section 8.3.

Provided a set Φ of sorts (with a decidable equality), the types Σ and the terms Λ are defined as follows:

Definition 8.4.1 ($\lambda_{\mathbf{TdB}}$ syntax)

$$\begin{array}{rcll} \Sigma & \triangleq & (\Phi) & \textit{Sorts} \\ & | & \Sigma \blacktriangleright \Sigma & \textit{Functional types} \end{array} \qquad \begin{array}{rcll} \Lambda & \triangleq & \delta(\Sigma, \mathbb{N}) & \text{de Bruijn } \textit{indexes} \\ & | & \underline{\lambda}(\Sigma, \Lambda) & \textit{Abstractions} \\ & | & \Lambda @ \Lambda & \textit{Applications} \end{array}$$

Any index is associated to a type to describe a free or bound variable. As in the previous subsection, the idea is not to infer the type of a variable before decorating the term, but to enforce the type to be part of the identifier. If $\sigma_1 \neq \sigma_2$ then $\chi(\sigma_1, i)$ and $\chi(\sigma_2, i)$ denote different variables, and both can appear in the same term at the same λ-height.

The abstraction, being a raw *de Bruijn* binder, is denoted $\underline{\lambda}$; as previously we reserve the notation λ for a more user-friendly notation. It is parameterised by a type in which indexes are captured, but not by a name or a value, as it is usual in *de Bruijn* notations.

Notation 8.4.1 *$\underline{\lambda}^{\sigma}\, l$ denotes the constructor $\underline{\lambda}$ applied to the type σ and the term l.*

As mentioned, the most visible impact of our modification is that there is no syntactical category for typing contexts, that is compared with standard representations of λ-calculus in which typing judgements are of the form $\Gamma \vdash l : \sigma$, we do not need to define a syntactical category for Γ, nor even to have such a Γ to express a typing judgement.

In contrast with the BiCoq calculus, we embed the application as a constructor of the language. This is done to adopt the standard vision, and ease comparison with other calculi. The consequences are that we now have in the formal language terms that are not in normal form, or that are ill-typed (cf. the next subsection); in BiCoq such terms were only described in the meta-language.

8.4.3 Typing

Without surprise, we define the typing relation $l{:}\sigma$, l is of type σ, as follows:

Definition 8.4.2 (Typing relation)

$$\frac{}{\delta(\sigma,i){:}\sigma} \qquad \frac{l{:}\sigma_1}{\underline{\lambda}^{\sigma_2}\, l{:}\sigma_2 \blacktriangleright \sigma_1} \qquad \frac{l_1{:}\sigma_2 \blacktriangleright \sigma_1 \quad l_2{:}\sigma_2}{l_1@l_2{:}\sigma_1}$$

It is obviously decidable, deterministic but not complete: any term has at most one type, but some terms are not typable. We can therefore implement a function *typeof*, not detailed here, that for any term l, returns an option type, such that:

Proposition 8.4.1 (Decision procedure for typing)

$$l{:}\sigma \quad \Leftrightarrow \quad \text{typeof}(l) = \mathbf{some}\,\sigma$$

$$(\forall\,(\sigma{:}\Sigma),\, \neg l{:}\sigma) \quad \Leftrightarrow \quad \text{typeof}(l) = \mathbf{none}$$

8.4.4 Standard operations for $\lambda_{\mathbf{TdB}}$

We now define a few usual operations on terms, adapted to our representation. We first describe a standard implementation of these operations – denoted here with an S exponent – before introducing in the next paragraph changes similar to those done in BiCoq, enforcing a precise context management. We keep the two versions to allow for a smoother introduction of the notations and associated concepts, but also to permit comparisons.

To start with a trivial example, we consider the function deciding whether a variable appears free in a term, that is more precisely whether an index appears dangling:

Definition 8.4.3 ($\lambda_{\mathbf{TdB}}$ standard freeness)

$$\begin{array}{lcll} \text{Free}^S(\sigma,i){:}\Lambda\to\mathbb{B} & \triangleq & \delta(\sigma',i') & \mapsto \quad \sigma{=}\sigma' \wedge i{=}i' \\ & | & \underline{\lambda}^{\sigma'}\, l' & \mapsto \quad \text{Free}^S(\sigma, \mathbf{if}\ \sigma{=}\sigma'\ \mathbf{then}\ i{+}1\ \mathbf{else}\ i)\, l' \\ & | & l_1@l_2 & \mapsto \quad \text{Free}^S(\sigma,i)\, l_1 \vee \text{Free}^S(\sigma,i)\, l_2 \end{array}$$

The index parameter i is conditionnaly incremented when a binder is crossed: we only increment the index if the variable is in the scope of the binder, that is if they have the same type. We implicitly manage one λ-height per type, as this will be clarified later. Indeed in λ_{TdB} crossing a binder of a given type has no influence on the representation of variables of other types, as illustrated by this example:

Example 8.4.3 (Contexts and variable representations in $\lambda_{\mathbf{TdB}}$)

$$\begin{array}{lll} \lambda_V \textit{ notation} & X_0{:}\sigma \ \vdash & \lambda x{:}\sigma{\cdot}(x{+}\lambda y{:}\sigma'{\cdot}(x{-}y)\, X_0) \\ \lambda_{TdB} \textit{ notation} & & \underline{\lambda}^{\sigma}((\sigma,0){+}\underline{\lambda}^{\sigma'}((\sigma,0){-}(\sigma',0))(\sigma,1)) \quad \textit{if} \quad \sigma{\neq}\sigma' \\ & & \underline{\lambda}^{\sigma}((\sigma,0){+}\underline{\lambda}^{\sigma}\,((\sigma,1){-}(\sigma\,,0))(\sigma,2)) \quad \textit{if} \quad \sigma{=}\sigma' \end{array}$$

The next operation is lifting, that increments dangling indexes in a term. It is used to adapt terms when crossing a binder in order to avoid captures of free variables. In our case, binders are parameterised by a type, therefore the lifting operation is parameterised as well by a type, and only increments indexes of this type; it also needs a contextual parameter, the current λ-height (for the considered type):

Definition 8.4.4 ($\lambda_{\mathbf{TdB}}$ standard lifting)

$$\begin{array}{rcccl}
\uparrow^S_{\sigma,h}\colon \Lambda\to\Lambda & \triangleq & \delta(\sigma',i') & \mapsto & \delta(\sigma', \textbf{if } \sigma=\sigma' \wedge h\leq i' \textbf{ then } i'+1 \textbf{ else } i') \\
& | & \underline{\lambda}^{\sigma'}\, l' & \mapsto & \underline{\lambda}^{\sigma'}\, (\uparrow^S_{\sigma,(\textbf{if } \sigma=\sigma' \textbf{ then } h+1 \textbf{ else } h)}\ l') \\
& | & l_1@l_2 & \mapsto & \uparrow^S_{\sigma,h}\ l_1@ \uparrow^S_{\sigma,h}\ l_2
\end{array}$$

As usual, the parameter h is left implicit when lifting from λ-height 0 – which is the only valid one from the user perspective, as other values of the λ-height may only result from recursive calls when progressing in the term.

The next operation is called here elimination, and is denoted $[\downarrow]^S$. It represents BiCoq functional application[9], that is it is such that $(\underline{\lambda}^\sigma\, l)@l_a$ reduces into $[\downarrow l_a]^S_{\sigma,0}\, l$ – it computes the reduction of a redex. It is a form of higher-order substitution, as dangling indexes of the considered type need to be decremented:

Definition 8.4.5 ($\lambda_{\mathbf{TdB}}$ standard elimination)

$$\begin{array}{rcccl}
[\downarrow l_e]^S_{\sigma,h}\colon \Lambda\to\Lambda & \triangleq & \delta(\sigma',i') & \mapsto & \left\{\begin{array}{lll}
\delta(\sigma',i') & \textit{if} \quad \sigma\neq\sigma' & \textit{(extern)} \\
\delta(\sigma',i') & \textit{if} \quad \sigma=\sigma' \wedge h>i' & \textit{(bound)} \\
l_e & \textit{if} \quad \sigma=\sigma' \wedge h=i & \textit{(instantiate)} \\
\delta(\sigma',i'-1) & \textit{if} \quad \sigma=\sigma' \wedge h<i' & \textit{(dangling)}
\end{array}\right. \\
& | & \underline{\lambda}^{\sigma'}\, l' & \mapsto & \underline{\lambda}^{\sigma'}\, [\downarrow\uparrow^S_{\sigma'}\ l_e]^E_{\sigma,(\textbf{if } \sigma=\sigma' \textbf{ then } h+1 \textbf{ else } h)}\, l' \\
& | & l_1@l_2 & \mapsto & [\downarrow l_e]^S_{\sigma,h}\, l_1@[\downarrow l_e]^S_{\sigma,h}\, l_2
\end{array}$$

Regarding the elimination on a variable, the cases are as follows:

- *extern: elimination in a different type (no effect, as there is one context per type);*
- *bound: the index is bound by a binder which is not the one being eliminated;*
- *instantiate: the index is bound by the binder which is being eliminated, and is therefore replaced by the parameter expression;*
- *dangling: the index is dangling and needs to be decremented to represent the same variable after binder elimination.*

In our representation, elimination does not need to be parameterised by a variable or an index, but by a type – the one of the binder that we are eliminating.

These operations are sufficient for example to discuss strategies of reduction. Yet, we can consider as for BiCoq a few additional operations, such as functional abstraction and substitution. Functional abstraction offers a form of user-friendly notation, and can also be used as part of a translation operator between λ_{TV} (the representation with typed names) and λ_{TdB}:

Definition 8.4.6 ($\lambda_{\mathbf{TdB}}$ standard functional abstraction)

$$\lambda^S\, (\sigma,i)\cdot l \quad \triangleq \quad \underline{\lambda}^\sigma\, \mathrm{Abstr}^S_0\, (\sigma,i)\, l$$

where Abstr^S *is the following function:*

$$\begin{array}{rcccl}
\mathrm{Abstr}^S_h(\sigma,i)\colon \Lambda\to\Lambda & \triangleq & \delta(\sigma',i') & \mapsto & \delta\left(\ \sigma', \left\{\begin{array}{lll}
h & \textit{if} \quad \sigma=\sigma' \wedge h\leq i' \wedge i=i' \\
i'+1 & \textit{if} \quad \sigma=\sigma' \wedge h\leq i' \wedge i\neq i' \\
i' & \textit{if} \quad \sigma\neq\sigma' \vee h>i'
\end{array}\right.\ \right) \\
& | & \underline{\lambda}(\sigma',l') & \mapsto & \underline{\lambda}^{\sigma'}\, \mathrm{Abstr}^S_{\textbf{if } \sigma=\sigma' \textbf{ then } h+1 \textbf{ else } h}\, (\sigma, \textbf{if } \sigma=\sigma' \textbf{ then } i+1 \textbf{ else } i)\, l' \\
& | & l_1@l_2 & \mapsto & (\mathrm{Abstr}^S_h\, (\sigma,i)\, l_1)@(\mathrm{Abstr}^S_h\, (\sigma,i)\, l_2)
\end{array}$$

[9] We have changed the name of this operation to avoid the confusion with the term constructor in λ_{TdB}.

Based on the same principles, the encoding of substitution is straightforward:

Definition 8.4.7 ($\lambda_{\mathbf{TdB}}$ standard substitution)

$$
\begin{array}{rcccl}
[(\sigma,i)\backslash l_s]^S : \Lambda \to \Lambda & \triangleq & \delta(\sigma',i') & \mapsto & \left\{ \begin{array}{ll} l_s & \textit{if} \quad \sigma = \sigma' \wedge i = i \\ \delta(\sigma',i') & \textit{if} \quad \sigma \neq \sigma' \vee i \neq i' \end{array} \right. \\
 & | & \underline{\lambda}^{\sigma'}\, l' & \mapsto & \underline{\lambda}^{\sigma'}\, [(\sigma, \textbf{if } \sigma = \sigma' \textbf{ then } i+1 \textbf{ else } i)\backslash \uparrow^S_{\sigma'}\, l_s]^S l' \\
 & | & l_1 @ l_2 & \mapsto & [(\sigma,i)\backslash l_s]^S l_1 @ [(\sigma,i)\backslash l_s]^S l_2
\end{array}
$$

8.4.5 Context aware operations for $\lambda_{\mathbf{TdB}}$

The previous definitions of the operations on terms are fully valid, but as indicated, not entirely satisfactory from our perspective. Indeed, we want also to validate these definitions through the proof of various properties, for example commutation results, as well as to examine properties of our calculus, such as confluence.

As explained in Section 8.1.5, our experimentations in BiCoq have led us to favour a form of detailed and precise context management for all the operations. In essence, when using operations such as elimination or substitution, that lift their parameters when crossing a binder, we want to use the correct λ-height parameter instead of the default value; this leads us to explicit this parameter, and to generalize it to all operations, even to those that do not require it, such as freeness or substitution. Having this parameter explicit and generalised, we also use it to condition the behaviour of the operations, for example not making anything if the call is ill-formed. These changes simplify many of our theorems, that have less side conditions and more natural expression, but also the proof of these theorems.

Making the same adaptations for λ_{TdB} is relatively straightforward, with the exception of the nature of the contextual parameter. Indeed, because we have implicitly one context per type, a single natural value is not sufficient; we need instead a map, that is a function in $\Sigma \to \mathbb{N}$, returning for any type the current λ-height. Let us illustrate the principle by expliciting the evolution of the context when progressing in a term:

Example 8.4.4 (Evolution of context)

$$
\left[\begin{array}{l} \sigma_1 : 0 \\ \sigma_2 : 0 \\ \sigma_3 : 0 \end{array} \right] \underline{\lambda}^{\sigma_1}\, \underline{\lambda}^{\sigma_2}\, \underline{\lambda}^{\sigma_1}\, l \to \underline{\lambda}^{\sigma_1} \left[\begin{array}{l} \sigma_1 : 1 \\ \sigma_2 : 0 \\ \sigma_3 : 0 \end{array} \right] \underline{\lambda}^{\sigma_2}\, \underline{\lambda}^{\sigma_1}\, l \to \underline{\lambda}^{\sigma_1}\, \underline{\lambda}^{\sigma_2} \left[\begin{array}{l} \sigma_1 : 1 \\ \sigma_2 : 1 \\ \sigma_3 : 0 \end{array} \right] \underline{\lambda}^{\sigma_1}\, l \to \underline{\lambda}^{\sigma_1}\, \underline{\lambda}^{\sigma_2}\, \underline{\lambda}^{\sigma_1} \left[\begin{array}{l} \sigma_1 : 2 \\ \sigma_2 : 1 \\ \sigma_3 : 0 \end{array} \right] l
$$

The reason why we want to keep trace of this context are the same as for BiCoq: remembering for example that in the substitution $[i\backslash l]_h$, l is such that it has already been lifted h times, and has therefore no dangling index $i < h$. In λ_{TdB}, this is similar, except that this information is detailed for each type; the cost may appear important, but we can remember that a mapping is similar to the context encountered in other representations, and that our calculus only use such mapping for internal computations during recursion. That is, this level of complexity is hidden to the user.

This being said, we should remember that these changes have no computational justification, and are not economical in terms of memory – but they give a very different flavour to the theorems and proofs. Note also that the λ_{TdB} representation does not enforce such a context management, it is just our optimised version of the operations that requires it.

Using maps, we first introduce a constant μ representing the context for the root of a term, a common condition $\mathcal{D}$ checking whether an index is dangling in given type and context, and an operation $\oplus$ to increment the λ-height for a specific type:

Definition 8.4.8 (Root context)

$$\mu : \Sigma \to \mathbb{N} \quad \triangleq \quad \mathbf{fun}\, \sigma' : \Sigma \mapsto 0$$

Definition 8.4.9 (Dangling index characterisation)

$$\mathcal{D}_{\sigma,m}(\sigma', i') : \mathbb{B} \quad \triangleq \quad \sigma = \sigma' \wedge (m\ \sigma') \leq i'$$

Definition 8.4.10 (Context incrementation)

$$m \oplus \sigma : \Sigma \to \mathbb{N} \quad \triangleq \quad \mathbf{fun}\, \sigma' \mapsto \mathbf{if}\ \sigma = \sigma'\ \mathbf{then}\ (m\ \sigma') + 1\ \mathbf{else}\ (m\ \sigma')$$

Applying these principle first to freeness, we add such a contextual parameter:

Definition 8.4.11 ($\lambda_{\mathbf{TdB}}$ context aware freeness)

$$\begin{array}{rcccl} \mathrm{Free}_m(\sigma, i) : \Lambda \to \mathbb{B} & \triangleq & \delta(\sigma', i') & \mapsto & \mathbf{if}\ \mathcal{D}_{\sigma,m}(\sigma', i')\ \mathbf{then}\ i = i'\ \mathbf{else}\ \bot \\ & | & \underline{\lambda}^{\sigma'}\ l' & \mapsto & \mathrm{Free}_{m \oplus \sigma'}\ (\sigma, \mathbf{if}\ \mathcal{D}_{\sigma',m}(\sigma, i)\ \mathbf{then}\ i+1\ \mathbf{else}\ i)\ l' \\ & | & l_1 @ l_2 & \mapsto & \mathrm{Free}_m(\sigma, i)\ l_1 \vee \mathrm{Free}_m(\sigma, i)\ l_2 \end{array}$$

Note that the contextual parameter is also used to condition the answer. That is, for an index, instead of returning directly $i = i'$ as previously, we first check that i is dangling in the context. Intuitively, the condition $\mathcal{D}_{\sigma,m}(\sigma', i')$, which defines a well-formed call, is explicited and internalised as a guard in the code.

In the case of lifting, the λ-height parameter exists in both versions, but is here generalised as a map. This is sufficient to be able to identify dangling indexes in all types, even if lifting still increment only the dangling indexes of the type σ passed as a parameter:

Definition 8.4.12 ($\lambda_{\mathbf{TdB}}$ context aware lifting)

$$\begin{array}{rcccl} \uparrow^{\sigma}_{m} : \Lambda \to \Lambda & \triangleq & \delta(\sigma', i') & \mapsto & \delta(\sigma', \mathbf{if}\ \mathcal{D}_{\sigma,m}(\sigma', i')\ \mathbf{then}\ i+1\ \mathbf{else}\ i) \\ & | & \underline{\lambda}^{\sigma'}\ l' & \mapsto & \underline{\lambda}^{\sigma'}\ \uparrow^{\sigma}_{m \oplus \sigma'}\ l' \\ & | & l_1 @ l_2 & \mapsto & \uparrow^{\sigma}_{m}\ l_1 @ \uparrow^{\sigma}_{m}\ l_2 \end{array}$$

For elimination, the contextual parameter is also used with the internal liftings:

Definition 8.4.13 ($\lambda_{\mathbf{TdB}}$ context aware elimination)

$$\begin{array}{rcccl} [\downarrow l_e]^{\sigma}_{m} : \Lambda \to \Lambda & \triangleq & \delta(\sigma', i') & \mapsto & \left\{ \begin{array}{ll} l_e & \mathit{if}\ \ \mathcal{D}_{\sigma,m}(\sigma', i') \wedge (m\ \sigma) = i' \\ \delta(\sigma', i'-1) & \mathit{if}\ \ \mathcal{D}_{\sigma,m}(\sigma', i') \wedge (m\ \sigma) \neq i' \\ \delta(\sigma', i') & \mathit{if}\ \ \neg \mathcal{D}_{\sigma,m}(\sigma', i') \end{array} \right. \\ & | & \underline{\lambda}^{\sigma'}\ l' & \mapsto & \underline{\lambda}^{\sigma'} [\downarrow \uparrow^{\sigma'}_{m}\ l_e]^{\sigma}_{m \oplus \sigma'} l' \\ & | & l_1 @ l_2 & \mapsto & [\downarrow l_e]^{\sigma}_{m} l_1 @ [\downarrow l_e]^{\sigma}_{m} l_2 \end{array}$$

Similarly for functional abstraction and substitution, we got[10]:

Definition 8.4.14 ($\lambda_{\mathbf{TdB}}$ context aware functional abstraction)

$$\lambda\, (\sigma, i) \cdot l \quad \triangleq \quad \underline{\lambda}^{\sigma}\ \mathrm{Abstr}_{\mu}(\sigma, i)\ l$$

[10] For the sake of clarity, we make an abuse of our notations by using the expression $\uparrow_{\sigma',m}\ (\sigma, i)$ to denote $(\sigma, \mathbf{if}\ \mathcal{D}_{\sigma,m}(\sigma', i')\ \mathbf{then}\ i+1\ \mathbf{else}\ i)$.

where Abstr *is the following function:*

$$\begin{array}{rcccl}
\mathrm{Abstr}_m(\sigma,i):\Lambda\to\Lambda & \triangleq & \delta(\sigma',i') & \mapsto & \delta\left(\sigma',\left\{\begin{array}{lll}(m\,\sigma') & \mathit{if} & \mathcal{D}_{\sigma,m}(\sigma',i')\wedge i=i'\\ i'+1 & \mathit{if} & \mathcal{D}_{\sigma,m}(\sigma',i')\wedge i\neq i'\\ i' & \mathit{if} & \neg\mathcal{D}_{\sigma,m}(\sigma',i')\end{array}\right.\right)\\
 & | & \underline{\lambda}^{\sigma'}\,l' & \mapsto & \underline{\lambda}^{\sigma'}\,\mathrm{Abstr}_{m\oplus\sigma'}\uparrow_{\sigma',m}(\sigma,i)\,l'\\
 & | & l_1@l_2 & \mapsto & \mathrm{Abstr}_m(\sigma,i)\,l_1@\mathrm{Abstr}_m(\sigma,i)\,l_2
\end{array}$$

Definition 8.4.15 ($\lambda_{\mathbf{TdB}}$ context aware substitution)

$$\begin{array}{rcccl}
[(\sigma,i)\backslash l_s]_m:\Lambda\to\Lambda & \triangleq & \delta(\sigma',i') & \mapsto & \textbf{if } \mathcal{D}_{\sigma,m}(\sigma',i')\wedge i=i' \textbf{ then } l_s \textbf{ else } \delta(\sigma',i')\\
 & | & \underline{\lambda}^{\sigma'}\,l' & \mapsto & \underline{\lambda}^{\sigma'}[\uparrow_{\sigma',m}(\sigma,i)\backslash\uparrow_{\sigma',m} l_s]_{m\oplus\sigma'}l'\\
 & | & l_1@l_2 & \mapsto & [(\sigma,i)\backslash l_s]_m l_1@[(\sigma,i)\backslash l_s]_m l_2
\end{array}$$

8.4.6 A quick analysis

To illustrate the fact that the λ_{TdB} notation indeed introduces implicitly a context per type, we can consider the following properties:

Proposition 8.4.2

$$\begin{array}{lcccl}
 & & \mathrm{typeof}(\uparrow_{\sigma,m} l) & = & \mathrm{typeof}(l)\\
 & & \mathrm{typeof}(\mathrm{Abstr}_m(\sigma,i)\,l) & = & \mathrm{typeof}(l)\\
l_e:\sigma & \Rightarrow & \mathrm{typeof}([\downarrow l_e]^{\sigma}_m l) & = & \mathrm{typeof}(l)\\
l_s:\sigma & \Rightarrow & \mathrm{typeof}([(\sigma,i)\backslash l_s]_m l) & = & \mathrm{typeof}(l)\\
l_b:\sigma & \Rightarrow & \lambda\,(\sigma',i)\cdot l_b:\sigma\blacktriangleright\sigma' & &
\end{array}$$

The first one is very significative, as lifting does not preserve typing in standard *de Bruijn* notation. Similar results exist, but require shifting the typing context in parallel with lifting the term. This form of context calculus is internalised in our term representation.

We can also quickly compare the two versions of our operations detailed herebefore by considering the expression of various properties. Let us consider the very simple example of the commutation lemmas of two liftings (such lemmas are often required, for example to prove the results of the next paragraph). Using the two versions of lifting, we have the following properties:

Proposition 8.4.3 (Commutation of standard liftings)

$$h_1\leq h2 \quad\Rightarrow\quad \uparrow^S_{\sigma,h_1}(\uparrow^S_{\sigma,h_2} l) = \uparrow^S_{\sigma,h_2+1}(\uparrow^S_{\sigma,h_1} l)$$

$$\sigma_1\neq\sigma_2 \quad\Rightarrow\quad \uparrow^S_{\sigma_1,h_1}(\uparrow^S_{\sigma_2,h_2} l) = \uparrow^S_{\sigma_2,h_2}(\uparrow^S_{\sigma_1,h_1} l)$$

Proposition 8.4.4 (Commutation of context aware liftings)

$$\uparrow^{\sigma_1}_{m\oplus\sigma_2}(\uparrow^{\sigma_2}_m l) = \uparrow^{\sigma_2}_{m\oplus\sigma_1}(\uparrow^{\sigma_1}_m l)$$

There is no miracle: in essence, these results are similar, and the side conditions associated to the results for the first version of lifting are just embedded as notations and computations in the second version. Note also that the result for the improved version of lifting presented here can only be used when there is a common context m for the two liftings. This is of course restrictive, but sufficient for some of our purposes. But the interesting observation is that this constraint about a common context cannot be written at all with the first version of lifting.

The consequences of these differences are not to underestimate: we have (a little) more lemmas with the standard versions of the operations, that are (a little) more difficult to express, and (a little) more difficult to prove. By this last statement, we mean that for example during proofs by induction, the standard version leads to a rewriting accompanied by a proof that the condition is satisfied, whereas the improved version is just a rewriting; the difference can have strong impact for example on the efficiency of proof automation.

We just consider another example, a property discussed for BICOQ and still valid here:

Proposition 8.4.5 (Standard substitution as a composite operation)

$$h \leq i \quad \Rightarrow \quad [\downarrow l_s]^S_{\sigma,h} \mathrm{Abstr}^S_h(\sigma, i)\, l = [(\sigma, i) \backslash l_s]^S l$$

Proposition 8.4.6 (Context aware substitution as a composite operation)

$$[\downarrow l_s]^\sigma_m \mathrm{Abstr}_m(\sigma, i)\, l = [(\sigma, i) \backslash l_s]_m l$$

These results both show that substitution does not need to be primitive in our calculus, as it is equivalent to abstraction followed by elimination. Yet again the second version is easier to handle in the absence of side condition.

8.4.7 β-reduction and normal form

The β-reduction is defined using the operations introduced in the previous subsection:

Definition 8.4.16 (β-reduction relation)

$$(\underline{\lambda}^\sigma\, l) @ l_a \quad \rightarrow_\beta \quad [\downarrow l_a]^\sigma_\mu\, l \qquad (\beta\text{-}red)$$

$$l \rightarrow_\beta l' \quad \Rightarrow \quad \underline{\lambda}^\sigma\, l \quad \rightarrow_\beta \quad \underline{\lambda}^\sigma\, l' \qquad (\beta\text{-}abstr)$$

$$l_1 \rightarrow_\beta l_1' \quad \Rightarrow \quad l_1 @ l_2 \quad \rightarrow_\beta \quad l_1' @ l_2 \qquad (\beta\text{-}appl)$$

$$l_2 \rightarrow_\beta l_2' \quad \Rightarrow \quad l_1 @ l_2 \quad \rightarrow_\beta \quad l_1 @ l_2' \qquad (\beta\text{-}appr)$$

We also define the function *normal*, not detailed here, checking whether a term is in a normal form, and prove the consistency with β-reduction:

Proposition 8.4.7 (Characterisation of normal forms)

$$\mathrm{normal}(l) \Leftrightarrow (\forall\, (l' : \Lambda),\, \neg(l \rightarrow_\beta l'))$$

$$\mathrm{normal}(l) \vee \exists\, l',\, l \rightarrow_\beta l'$$

The constructive proof of the second result is a program that decide whether or not a term is in normal form, and in the latter case returns a reduction.

β-reduction, as expected, preserves typing; we need just to ensure that the initial term is well-typed, as an ill-typed term can become well-typed after a reduction:

Proposition 8.4.8 (Preservation lemma)

$$l : \sigma \quad \Rightarrow \quad l \to_\beta l' \quad \Rightarrow \quad \text{typeof}(l) = \text{typeof}(l')$$

We have also proven results about commutations between eliminations, and derived the following standard semantic results:

Proposition 8.4.9 (Commutation lemmas)

$$l \to_\beta l' \quad \Rightarrow \quad [\downarrow l_e]^\sigma_\mu \, l \to_\beta [\downarrow l_e]^\sigma_\mu \, l'$$

$$l_e \to_\beta l'_e \quad \Rightarrow \quad [\downarrow l_e]^\sigma_m \, l \to^*_\beta [\downarrow l'_e]^\sigma_m \, l$$

Note that these lemmas are not expressed with substitution but with elimination, and that the first lemma is only valid with the context μ.

With these results, confluence can be proven; at this stage, we have just shown the weak confluence:

Proposition 8.4.10 (Weak confluence)

$$l \to_\beta l_1 \quad \Rightarrow \quad l \to_\beta l_2 \quad \Rightarrow \quad \exists\, l',\, l_1 \to^*_\beta l' \wedge l_2 \to^*_\beta l'$$

The proof for strong confluence is expected to be immediate, and the proof of confluence is likely to be rather easy as well. Normalisation is yet to be done.

8.4.8 Grafting

The last operation described in this section is the grafting, defined as follows:

Definition 8.4.17 (Grafting)

$$\begin{array}{rcll} [(\sigma,i) \lhd l_g]_m : \Lambda \to \Lambda & \triangleq & \delta(\sigma', i') & \mapsto \quad \textbf{if } \mathcal{D}_{\sigma,m}(\sigma', i') \wedge i = i' \textbf{ then } l_s \textbf{ else } \delta(\sigma', i') \\ & | & \underline{\lambda}^{\sigma'} \, l' & \mapsto \quad \underline{\lambda}^{\sigma'} [\uparrow_{\sigma',m} (\sigma, i) \lhd l_g]_{m \oplus \sigma'} l' \\ & | & l_1 @ l_2 & \mapsto \quad [(\sigma, i) \lhd l_g]_m l_1 @ [(\sigma, i) \lhd l_g]_m l_2 \end{array}$$

It is very similar with substitution, the only difference being that substitution lifts its term parameter when crossing a binder, while grafting does not. That is, substitution prevents the capture of the free variables in its parameter, but grafting does not.

In λ_{TdB}, grafting obviously preserves typing:

Proposition 8.4.11 (Grafting and typing)

$$l_g : \sigma \quad \Rightarrow \quad \text{typeof}([(\sigma, i) \lhd l_g]_m l) \quad = \quad \text{typeof}(l)$$

This property is not valid in standard *de Bruijn* representations, and contrary to what was noted for lifting and typing, we are not aware of any reasonable way to derive a similar property in a standard notation – shifting the typing context would not be sufficient.

In essence, the satisfaction of this property is the main justification for introducing namespaces in BiCoq and typed *de Bruijn* indexes here. Indeed, grafting, by not lifting its parameter when crossing a binder, does not update variable representation to ensure their consistency with the context. That is, in practice, we lose the semantics of the indexes, and in a named representation this would correspond to an unmonitored and untraceable replacements of y by x, z by y, etc. – a situation made worse if the grafted

variable appear several times, as these replacements would not be the same at different locations in the term to which grafting is applied.

At this stage however, grafting is still an exotic operation, possibly meaningless as would be any other operation scrambling terms. We have not yet developped an associated formal theory, and we just provide here the intuition behind the definition of this function, and its potential uses – solely based on the observations and proofs done for BiCoq.

Grafting, in a way, is the form of substitution that is used for rewriting. Provided a rule $l \to r$, when identifying that in a term t the subterm matches l, we can apply the rule to obtain a new term where the subterm l has been replaced by r. Such a replacement is independent from the fact that l or r can have free variables captured by a binder in t – that is, it is inherently first-order.

A rewriting rule can be emulated using grafting: given a term t with a subterm l, we produce a form of pattern $t = [(\sigma, i) \lhd l]t'$, where (σ, i) is an arbitrary free variable (a dangling index), then apply a congruence rule such as:

$$C \Rightarrow l \rightsquigarrow r \Rightarrow [\sigma, i \lhd l]t' \rightsquigarrow [\sigma, i \lhd r]t'$$

Where C is a side condition. Of course the validity of such a congruence rule has to be proven, as it was done for BiCoq in the B logic. Grafting is therefore just a tool to express such rules, substitution being too limited.

An intuitive illustration of these concepts is provided by the β-reduction, which is defined as a base rule β-red and recursion rules that transform the base rule into a rewriting rule. β-reduction can indeed also be defined using grafting as follows:

Definition 8.4.18 (β-reduction by grafting)

$$\underline{\lambda}(\sigma, l)@l_a \quad \to_{\beta'} \quad [\downarrow l_a]^{\sigma}_{\mu}\, l \qquad \beta'\text{-}red$$

$$(l_g \to_{\beta'} l'_g) \quad \Rightarrow \quad [(\sigma, i) \lhd l_g]_{\mu}\, l \quad \to_{\beta'} \quad [(\sigma, i) \lhd l'_g]_{\mu}\, l \qquad \beta'\text{-}graft$$

8.4.9 Namespaces and meta-variables

Back to generic congruence rules, it may happen that the side condition C is not trivial. For example, in BiCoq, the congruence results that allow for the replacement of equal expressions or of equivalent predicates are associated to a condition about the variables appearing free in the proof context and in the equality or equivalence proposition – the so-called orthogonality condition (cf. Sections 8.3.2 and 8.3.3).

As indicated, this is justified by the fact that grafting permits captures but also results into a loss of context. This has led us to introduce namespaces in BiCoq, not as a solution to prevent losses of context, but rather as a solution to limit the scope of such losses – allowing for a weaker side condition, easier to satisfy, and with an intuitive justification (counter-examples are easier to elaborate). In fact, the use of namespaces is nothing more than a technical trick to benefit of stronger results by weakening side conditions.

In the λ_{TdB} notation we have extended the idea of namespaces to types. However, our feeling is that for advanced results, the problem of the loss of context caused by grafting can reappear. Of course, by construction, such losses of context will never results into losses of typing, but nevertheless we are not able to preserve the semantics of the dangling indexes. That is, we still need a way to restrict the scope of such losses by extending the notation, a way to "put apart" some variables to preserve them from unmonitored shifts.

A possible solution to this problem is to introduce a new syntactical class of named (meta) variables, with $\mathcal{N}$ a type of names with a decidable equality:

Definition 8.4.19 (Syntax extended with meta-variables)

$$\begin{array}{rcll} \Lambda & \triangleq & \delta(\Sigma,\mathbb{N}) & \text{de Bruijn } \textit{indexes} \\ & | & \chi(\Sigma,\mathcal{N}) & \textit{Variables} \\ & | & \underline{\lambda}(\Sigma,\Lambda) & \textit{Abstractions} \\ & | & \Lambda@\Lambda & \textit{Applications} \end{array}$$

Such variables are never bound, and never lifted or subject to computations – this is why we use names instead of indexes. It is then possible to extend all the operations previously defined, and to prove the same properties. For advanced results, side conditions can for example be the obligation to work only with terms without dangling indexes (at least for some given types), free variables being represented only by named variables.

The other solution is to reintroduce namespaces. In such a case, $\mathcal{N}$ is a type of namespaces with a decidable equality, parameterising the variables (represented by tuples with indexes) and the binders[11]:

Definition 8.4.20 (Syntax modified with namespaces)

$$\begin{array}{rcll} \Lambda & \triangleq & \delta(\Sigma,\mathcal{N},\mathbb{N}) & \text{de Bruijn } \textit{indexes} \\ & | & \underline{\lambda}(\Sigma,\mathcal{N},\Lambda) & \textit{Abstractions} \\ & | & \Lambda@\Lambda & \textit{Applications} \end{array}$$

We can consider for example a infinite set of namespaces, being able to define as many context scopes as required to derive interesting results. In other words, if a grafting leads to a possible confusion between variables x and y in the type σ, a possible solution is to move x and y in their own namespaces, that can be such that they are never bound. For example we can use $\mathbb{Z}$ as the set of namespaces, and parameterise binders by a namespace in $\mathbb{N}$; by doing so, we can ensure that negative namespaces are never bound, and that indexes in these namespaces can be considered as names, while indexes in positive namespaces can still be bound or dangling, lifted, etc. Remember that we do not expect namespaces to have semantics; they are merely a technical trick.

Beyond not introducing a new constructor – a rather trivial observation – we consider that this solution has some merits of its own when compared with the introduction of meta-variables.

The first justification is that, compared with the named variables approach, we can keep a set of operations that is generic and reduced: considering for example substitution, we do not need to assess whether we want an operation for the substitution of free variables represented by a dangling index and another one for the substitution of free variables represented by a named variable.

A much more important justification is provided by considering possible extensions, for example to polymorphic types. As previously illustrated, in λ_{TdB}, variables representations can be tricky to determine:

Example 8.4.5 (Context and variable representations)

$$\begin{array}{lll} \lambda_V \textit{ notation} & X_0{:}\sigma \ \vdash & \lambda x{:}\sigma\cdot(x{+}\lambda y{:}\sigma'\cdot(x{-}y)\ X_0) \\ \lambda_{TdB} \textit{ notation} & & \underline{\lambda}^{\sigma}((\sigma,0){+}\underline{\lambda}^{\sigma'}((\sigma,0){-}(\sigma',0))(\sigma,1)) \quad \textit{if} \quad \sigma{\neq}\sigma' \\ & & \underline{\lambda}^{\sigma}((\sigma,0){+}\underline{\lambda}^{\sigma}\ ((\sigma,1){-}(\sigma\ ,0))(\sigma,2)) \quad \textit{if} \quad \sigma{=}\sigma' \end{array}$$

[11] We have kept here both parameters, that is a type and a namespace; this ensures for example that lifting preserves typing, and does not require the management of typing contexts. Yet if such properties have no interest, it may be more efficient to drop types and to keep only namespaces in this representation.

We see that the question whether $\sigma = \sigma'$ or not is important to process the correct term representation. Now suppose we introduce type variables in the syntax of types to represent polymorphism, and let us reconsider our previous example with type variables α and β instead of σ and σ':

Example 8.4.6 (Polymorphic $\lambda_{\mathbf{TdB}}$)

$$\underline{\lambda}^{\alpha}((\alpha,0)+\underline{\lambda}^{\beta}((\alpha,0)-(\beta,0))(\alpha,1))$$

Instantiating these type variables is unfortunately not so easy. Indeed, if they are instantiated by different types, then it is just replacing the occurrences of the variables; but if these types are equal then we also need to renumber some of the indexes. We have in fact collapsed together two separate contexts and we need to adapt the representation accordingly.

On the contrary, using for example type variable names as namespaces (represented thereafter by boxed characters), we have a solution to maintain artifically the context separation. Even if we instantiate the type variables α and β by the same type σ, we can avoid renumbering the indexes; the procedure of instantiation is simpler and common to all cases. Applied to the previous example, the polymorphic and instantiated versions are as follows:

Example 8.4.7 (Polymorphic $\lambda_{\mathbf{TdB}}$ with namespaces)

$$\underline{\lambda}^{\alpha,\boxed{\alpha}}((\alpha,\boxed{\alpha},0)+\underline{\lambda}^{\beta,\boxed{\beta}}((\alpha,\boxed{\alpha},0)-(\beta,\boxed{\beta},0))(\alpha,\boxed{\emptyset},0))$$

$$\underline{\lambda}^{\sigma,\boxed{\alpha}}((\sigma,\boxed{\alpha},0)+\underline{\lambda}^{\sigma,\boxed{\beta}}((\sigma,\boxed{\alpha},0)-(\sigma,\boxed{\beta},0))(\sigma,\boxed{\emptyset},0))$$

Our feeling is that such representations, with parameterised variables or indexes, deserve additional studies. Whereas they are relatively complex to interpret for a human reader, their manipulation by programs is straightforward. These representations seem to mix some of the advantages of named representations and *de Bruijn* representations.

One could note however that a representation with namespaces is not α-quotiented as soon as there is more than one possibly bound namespace (per type). In such case we either need to define such an α-equivalence, or to swap between several representations according to the problems we have to tackle, introducing namespaces as special marks to deal with grafting results for example.

Chapter 9

Conclusion and Perspectives

9.1 Security traps and oversights

We have started this memoir with a question about the scope and the level of confidence resulting of the use of deductive formal methods for the development of secure systems.

This question, initially motivated by personal experiences emphasising the difficulties to correctly specify some elusive security requirements, has led us to a detailed review of specification-driven developments supported by deductive formal methods in Chapter 5. And, adopting a very skeptical vision, there is indeed a lot to say about the problems that can result of contradictory or insufficient specifications and of unexpected uses of proven systems – whatever the method that is used.

Yet, beyond illustrations of the potential consequences of inappropriate specifications, of the Refinement Paradox or of overoptimistic hypotheses, we have also tried in this memoir to explore the causes, to propose recommendations for developers or evaluators, or to discuss possible technical solutions.

We have emphasised why some common practices should be reconsidered when dealing with security applications. For example, the use of preconditions does not appear to be relevant, and guards should be preferred instead.

Inappropriate specifications also result of misunderstandings of the theory supporting the formal method – misunderstandings that can be tackled by more intuitive presentations of the underlying concepts. We have adopted this approach in this memoir with an explicit representation of the refinement process, illustrating how for example it is possible for a malicious developer to introduce hidden dependencies, in the B method but also in Coq or FoCaLize. We have also noted that the assessment of the quality of specifications can be supported by tools, such as model animators, test generators, and so on.

On the other hand, as far as the promotion of formal methods in industry is concerned, we are not convinced at this stage by alternative approaches that would complexify the theory to address all possible concerns related to security, such as for example finer refinement relations considering not only non determinism but also randomness and observability. Whereas these approaches definitely have an academic justification, such an increased complexity would have a cost – possibly unbearable for the industry – both in terms of training and use, without even speaking about the existence of tool support.

It is our feeling that, on the contrary, the interaction of deductive formal methods with other (more automated) approaches is preferrable. That is, for example, a form of dependency calculus can easilly deal with our concerns related to confidentiality and covert channels. This advocates the development of IDEs that would integrate such various approaches, as it is the objective for the FoCaLize environment.

9.2 Validation of theories and tools

Having noted the potential consequences of paradoxes in theories or tools, we have also explored in this memoir the validation of formal methods, with the example of the B method and the embedding of its logic in the Coq proof assistant in Chapters 7 and 8.

Such a validation is not a simple problem, and it has a cost which is not negligible. On the other hand, we have illustrated that it is both feasible and necessary, identifying for example multiple oversights in the B logic. The validation of the theory is in our view a typical academic activity, to be done once, for the benefit of the whole formal community, including industrial users. Such a validation has of course to be mechanically assisted, but several tools perfectly fitted for this task exist, and beyond confidence in the results their use brings various additional advantages.

In our case for example, we have benefited of the use of Coq not only to consider the validation of the B logic, but also to explore different representations, to derive mechanically checked tools, and to prove new congruence results. Having the support of a proof assistant, we have been able to tackle in trust numerous administrative details when dealing with advanced results such as substitution of subterms having free variables bound by their context. More generally, these tools appear relevant to deal efficiently and confidently with mathematical theories.

As far as the tools supporting formal methods are concerned, we have also noted that the completeness and the correction of a prover, for example, are not the only objectives to be considered when developing a formal environment. Indeed, one can also expect simplicity, ergonomics and automation – favouring again effective use in the industry. This can in our view justify the coexistence and collaboration of several tools and approaches.

9.3 A few perspectives

This memoir discusses numerous additional activities, that unfortunately have not been addressed by lack of time. We propose here a short summary of the perspectives that we have identified.

The seamless integration of mutiple approaches in a formal environment to support secure developments, for example, would be a conspicuous improvement. Such an integration is an observable trend in the B community – yet for better efficiency and ergonomics, rather than improved security – and is one of the primary objective of the FoCaLize project. Whereas we are not sure that the integration into a unified theory (of refinement for example) would be usable in an industrial environment, the ability to combine multiple orthogonal analyses, e.g. proofs for functional compliance, dependency graphs for confidentiality, and so on, would bring a much more complete vision at a minimal cost – at least as far as the development of secure systems is concerned.

Beyond existing techniques, it is our feeling that additional features should be addressed as well for integration into such formal environments. Dependency calculus, for example, operates at implementation level, and can identifies undesirable information flows for the benefit of an independent evaluator. Yet we have not identified how to specify dependencies in such a way that the constraints are propagated automatically during refinements. This would be a very interesting feature, for both safety (assessing fault propagations) and security (identifying covert channels), to be able to express dependency constraints at the abstract level while having guarantees about their preservation at the concrete level.

Similarly, additional specification engineering tools should also be considered. Such

tools would for example help to criticize specifications e.g. by generating additional proof obligations (to detect vacuous truths, empty types, and so on) or by building counter-examples (to detect inconsistencies or insufficient specifications). Another interesting application would be the ability to transform – or to assist with the transformation of – the specification, in whole or in part, into guards integrated within the code.

We have also discussed at length the problem of the validation of the theory and tools associated to a formal method. With respect to BiCoq for example, there is still a lot to do, such as assessing the consistency of the B logic and the exact role of the well-formedness checking. The integration of BiCoq with other B tools seems also desirable, e.g. as a robust base for shallow embeddings or as a proof checker for Ides. The development of new results, such as the congruence rules presented in this memoir, should also be further explored for example to enrich tactics – and potentially increase automation – of the existing provers.

Finally, we have described various forms of *de Bruijn* representations for terms, and the associated operations, considering better context management and variable identifiers with sorts (namespaces) or types. The interest of these approaches has to be studied, one of the immediate results being the definition of typed λ-calculi not requiring explicit typing contexts. One of the other subjects deserving further investigations is the development of a library for language mechanisation, not using a specific representation but combining them, e.g. describing notations with names, *de Bruijn* indexes and levels but also homomorphism properties between them. This would allow for choosing the most appropriate representation for the proof of a result, before being able to propagate this result toward the other representations using homomorphisms.

9.4 On the interest of formal methods

The presentation of deductive formal methods provided in this memoir, and in Chapter 5 in particular, is biased, considering worst-case scenarios and well-chosen examples to support our discussion. This leads to numerous remarks about the traps that any secure system development is facing.

As mentioned, however, we consider that formal methods in general, and deductive formal methods in particular, are very powerful tools that allow for effectively developping bug free implementations.

In our view, formal methods do provide a silver bullet for system engineering, and appropriately used they can lead to the detection and the eradication of flaws in protocols or policies, of non-compliancies, or of other forms of problems such as buffer overflows, out-of-bound accesses, and exceptions due to invalid computations. One of the other benefits of the use of deductive formal methods – and of other user-assisted formal methods – is that they bring a genuine knowledge about the system under development. If nothing else, proving a program is a remarquable way to reconsider and justify its design.

In our view, formal methods are not sufficiently used by industry today. They indeed have a poor reputation of being too complex and too expensive, and as a consequence they are mainly used for critical systems to support the certification process – that is as a mean to convince the independent evaluators and the certification authorities rather than to improve the reliability of the system.

But the fact is that the genuine complexity of a formal development is not related to the method – once appropriate training has been given, of course – but to the considered system. It is clear for example that the difficulty of software development is generally underestimated; formal methods just emphasise this difficulty by requesting justifications

and forbidding oversights. And as far as the cost is concerned, various studies and surveys indicate that the appropriate use of formal method in fact leads to significant reductions of the overall cost, considering not only development but also verification and maintenance.

We have decided, for the work described in this memoir, to always use formal methods to support our reasonings and developments. It has clearly illustrated both the interest of such an approach as well as its feasibility, and it has in fact helped us to consider problems that we would never have tried to tackle without such mechanical support. We expect this experience to give some weight to our recommendations and proposals.

Bibliography

[ABF+05] B.E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In Hurd and Melham [HM05], pages 50–65.

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[ABM01] Alessandro Avellone, Marco Benini, and Ugo Moscato. How to avoid the formal verification of a theorem prover. *Logic Journal of the IGPL*, 9(1), 2001.

[Abr96] J. R. Abrial. *The B-Book - Assigning Programs to Meanings.* Cambridge University Press, August 1996.

[ACCL91] M. Abadi, L. Cardelli, P-L. Curien, and J-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[ACM05] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Refinement and reachability in Event-B. In Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors, *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 222–241. Springer, 2005.

[ACPM05] June Andronick, Boutheina Chetali, and Christine Paulin-Mohring. Formal verification of security properties of smart card embedded source code. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 302–317. Springer, 2005.

[AHN08] Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. A compiled implementation of normalization by evaluation. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2008.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[AP02] A. Azurat and I.S.W.B. Prasetya. A survey on embedding programming logics in a theorem prover. Technical Report UU-CS-2002-007, Institute of Information and Computing Sciences, Utrecht University, 2002.

[Bac81] R-J. Back. On correct refinement of programs. *J. Comput. Syst. Sci.*, 23(1):49–68, 1981.

[Bac88] R-J. Back. A calculus of refinements for program derivations. *Acta Inf.*, 25(6):593–624, 1988.

[Bar99] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, November 1999.

[BAW98] Ralph-Johan J. Back, Abo Akademi, and J. Von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[BBM98] Patrick Behm, Lilian Burdy, and Jean-Marc Meynadier. Well defined B. In Bert [Ber98], pages 29–45.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BC06] Mike Bond and Jolyon Clulow. Integrity of intention (a theory of types for security APIs). *Information Security Technical Report*, 11(2):93 – 99, 2006.

[BCM07] Nazim Benaïssa, Dominique Cansell, and Dominique Méry. Integration of security policy into system modeling. In Julliand and Kouchnarenko [JK06], pages 232–247.

[BDD07] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In Dershowitz and Voronkov [DV07], pages 151–165.

[BDFF04] K. Berkani, C. Dubois, A. Faivre, and J. Falampin. Validation des règles de base de l'Atelier B. *Technique et Science Informatiques*, 23(7):855–878, 2004.

[BDL01] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *J. Cryptology*, 14(2):101–119, 2001.

[BDM98] P. Behm, P. Desforges, and J. M. Meynadier. MÉTÉOR : An industrial success in formal development. In Bert [Ber98], page 26.

[Ber98] D. Bert, editor. *B'98: Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, April 22-24, 1998, Proceedings*, volume 1393 of *Lecture Notes in Computer Science*. Springer, 1998.

[BF02] J.P. Bodeveix and M. Filali. Type synthesis in B and the translation of B to PVS. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 350–369, London, UK, 2002. Springer-Verlag.

[BFM99] J.-P. Bodeveix, M. Filali, and C. Muñoz. A formalization of the B-Method in Coq and PVS. In *Electronic Proceedings of the B-User Group Meeting at the World Congress on Formal Methods FM 99*, pages 33–49, 1999.

[BGG+92] R. J. Boulton, A. Gordon, M. J. C. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *TPCD*, volume A-10 of *IFIP Transactions*, pages 129–156. North-Holland, 1992.

[Bie96] P. Bieber. Formal techniques for an ITSEC-E4 secure gateway. In *ACSAC*, pages 236–246. IEEE Computer Society, 1996.

[Boe10] Mathieu Boespflug. Conversion by evaluation. In Manuel Carro and Ricardo Peña, editors, *PADL*, volume 5937 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2010.

[Bou07] Sylvain Boulmé. Intuitionistic refinement calculus. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *Lecture Notes in Computer Science*, pages 54–69. Springer, 2007.

[BP99] R. Bird and R. Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

[BP00] Richard Banach and Michael Poppleton. Retrenchment, refinement, and simulation. In Jonathan P. Bowen, Steve Dunne, Andy Galloway, and Steve King, editors, *ZB*, volume 1878 of *Lecture Notes in Computer Science*, pages 304–323. Springer, 2000.

[BP07] Sylvain Boulmé and Marie-Laure Potet. Interpreting invariant composition in the b method using the Spec# ownership relation: A way to explain and relax B restrictions. In Julliand and Kouchnarenko [JK06], pages 4–18.

[Bur00] Lilian Burdy. *Traitement des expressions dépourvues de sens de la théorie des ensembles – Application à la méthode B.* Thèse de doctorat, Conservatoire National des Arts et Métiers, may 2000.

[BvW00] Ralph-Johan Back and Joakim von Wright. Encoding, decoding and data refinement. *Formal Asp. Comput.*, 12(5):313–349, 2000.

[CC] ISO/IEC 15408: Common criteria for information technology security evaluation. http://www.commoncriteriaportal.org/.

[CD08] M. Carlier and C. Dubois. Functional testing in the FoCaL environment. In B. Berckert and R. Hahnle, editors, *Test And Proof (TAP'2008)*, volume 4966, pages 84–98. LNCS, 2008.

[CD09] Ana Cavalcanti and Dennis Dams, editors. *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*. Springer, 2009.

[CGL96] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking. In Manfred Broy, editor, *NATO ASI DPD*, pages 305–349, 1996.

[CH01] Bill Councill and George T. Heineman. Summary. pages 741–752, 2001.

[Cha98] P. Chartier. Formalisation of B in Isabelle/HOL. In Bert [Ber98], pages 66–82.

[CHL96] P-L. Curien, T. Hardin, and J-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, March 1996.

[CK98] H. Cirstea and C. Kirchner. Using rewriting and strategies for describing the B predicate prover. In Claude Kirchner and Hélène Kirchner, editors, *CADE-15 : Workshop on Strategies in automated deduction*, volume 1421 of *Lecture Notes in Computer Science*, pages 25–36, Lindau, Germany, 1998. Springer.

[Clu03] Jolyon Clulow. On the security of PKCS#11. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2779 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2003.

[CM06a] Dominique Cansell and Dominique Méry. Formal and incremental construction of distributed algorithms: On the distributed reference counting algorithm. *Theor. Comput. Sci.*, 364(3):318–337, 2006.

[CM06b] Judicaël Courant and Jean-François Monin. Defending the bank with a proof assistant. In *WITS 2006*, Vienna, March 2006. In WITS proceedings.

[CM09] Samuel Colin and Georges Mariano. Coq, l'alpha et l'omega de la preuve pour B ? 14 pages + annexe de deux pages, 2009.

[Coq] The Coq proof assistant. http://coq.inria.fr.

[CP88] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988.

[CP95] Jeffrey A. Clark and Dhiraj K. Pradhan. Fault injection. *Computer*, 28(6):47–56, 1995.

[CPR+05] S. Colin, D. Petit, J. Rocheteau, R. Marcano, G. Mariano, and V. Poirriez. BRILLANT : An open source and XML-based platform for rigourous software development. In *SEFM (Software Engineering and Formal Methods)*, Koblenz, Germany, september 2005. AGKI (Artificial Intelligence Research Koblenz), IEEE Computer Society Press. selectivity : 40/120.

[CPW06] A. Charguéraud, B. C. Pierce, and S. Weirich. Proof engineering: Practical techniques for mechanized metatheory, September 2006. Submitted for publication.

[CSC] John A. Clark, Susan Stepney, and Howard Chivers. Breaking the model: Finalisation and a taxonomy of security attacks.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, pages 381–392, 1972.

[Del00] D. Delahaye. A tactic language for the system Coq. In M. Parigot and A. Voronkov, editors, *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer-Verlag LNCS/LNAI, November 2000.

[DHK00] G. Dowek, T. Hardin, and C. Kirchner. Higher order unification via explicit substitutions. *Inf. Comput.*, 157(1-2):183–235, 2000.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DL07] Z. Dargaye and X. Leroy. Mechanized verification of CPS transformations. In Dershowitz and Voronkov [DV07], pages 211–225.

[DV07] Nachum Dershowitz and Andrei Voronkov, editors. *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, volume 4790 of *Lecture Notes in Computer Science*. Springer, 2007.

[DY81] Danny Dolev and Andrew C. Yao. On the security of public key protocols. Technical report, Stanford, CA, USA, 1981.

[ED07] Didier Essamé and Daniel Dollé. B in large-scale projects: The canarsie line cbtc experience. In Julliand and Kouchnarenko [JK06], pages 252–254.

[Flo67] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.

[Foc] The FoCaLize project. http://focalize.inria.fr/.

[GFL05] Frédéric Gervais, Marc Frappier, and Régine Laleau. Vous avez dit raffinement? Technical Report CEDRIC-829, CNAM, march 2005.

[GHS] Integrity real-time operating system. http://www.ghs.com/products/rtos/integrity.html.

[GM92] J.A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Press, 1992.

[GMW09] Herman Geuvers, James McKinna, and Freek Wiedijk. Pure type systems without explicit contexts. Submitted at TLCA'09, 2009.

[Gor93] A. D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In J. J. Joyce and C-J. H. Seger, editors, *HUG '93: Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 413–425, London, UK, 1993. Springer-Verlag.

[Had07] Amal Haddad. Meca: A tool for access control models. In Julliand and Kouchnarenko [JK06], pages 281–284.

[HAF01] M. Randall Holmes and J. Alves-Foss. The Watson theorem prover. *J. Autom. Reasoning*, 26(4):357–408, 2001.

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 07(5):11–19, 1990.

[HHGB07] Sarah Hoffmann, Germain Haugou, Sophie Gabriele, and Lilian Burdy. The B-Method for the construction of microkernel-based systems. In Julliand and Kouchnarenko [JK06], pages 257–259.

[HM05] J. Hurd and T. F. Melham, editors. *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*. Springer, 2005.

[Hoa92] C. A. R. Hoare. Programs are predicates. In *FGCS*, pages 211–218, 1992.

[HR84] C. A. R. Hoare and A. W. Roscoe. Programs as executable predicates. In *FGCS*, pages 220–228, 1984.

[IEC] IEC 61508: Functional safety of electrical, electronic, programmable electronic safety-related systems. http://www.iec.ch/zone/fsafety/.

[Jae05] Éric Jaeger. De C à B, l'analyse de code par les méthodes formelles. Master's thesis, Université Paris 7, September 2005.

[Jaf07] Eddie Jaffuel. Using B machines for model-based testing of smartcard software. In Julliand and Kouchnarenko [JK06], page 2.

[JK06] Jacques Julliand and Olga Kouchnarenko, editors. *B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17-19, 2007, Proceedings*, volume 4355 of *Lecture Notes in Computer Science*. Springer, 2006.

[JL07] Eddie Jaffuel and Bruno Legeard. LEIRIOS test generator: Automated test generation from B models. In Julliand and Kouchnarenko [JK06], pages 277–280.

[JLH+09] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. "carbon credits" for resource-bounded computations using amortised analysis. In Cavalcanti and Dams [CD09], pages 354–369.

[Jos88] Mark B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3(1):9–18, 1988.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *SOSP*, pages 207–220. ACM, 2009.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[Lam02] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Ler09] Xavier Leroy. A formally verified compiler back-end. *CoRR*, abs/0902.2137, 2009.

[Lin05] Sam Lindley. *Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages*. PhD thesis, University of Edinburgh, College of Science and Engineering, School of Informatics, 2005.

[Mag03a] Nicolas Magaud. *Changements de Représentation des Données dans le Calcul des Constructions*. PhD thesis, Université de Nice Sophia-Antipolis, October 2003.

[Mag03b] Nicolas Magaud. Changing Data Representation within the Coq System. In *TPHOLs'2003*, volume 2758. LNCS, Springer-Verlag, 2003.

[M.J88] M.J.C. Gordon. Mechanizing programming logics in higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, pages 387–439, Banff, Canada, 1988. Springer-Verlag, Berlin.

[MM04] Annabelle McIver and Carrol Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer Verlag, 2004.

[MMM09] Annabelle McIver, Larissa Meinicke, and Carroll Morgan. Security, probability and nearly fair coins in the cryptographers' café. In Cavalcanti and Dams [CD09], pages 41–71.

[Mor90] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[Muñ99] C. Muñoz. PBS: Support for the B-method in PVS, 1999.

[Mus05] L. Mussat, 2005. Private Communication.

[NV07] M. Norrish and R. Vestergaard. Proof pearl: de Bruijn terms really do work. In K. Schneider and J. Brandt, editors, *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2007.

[PL07] Daniel Plagge and Michael Leuschel. Validating Z specifications using the ProBAnimator and model checker. In Jim Davies and Jeremy Gibbons, editors, *IFM*, volume 4591 of *Lecture Notes in Computer Science*, pages 480–500. Springer, 2007.

[RCMP04] Jérôme Rocheteau, Samuel Colin, Georges Mariano, and Vincent Poirriez. Évaluation de l'extensibilité de PhoX: B/PhoX un assistant de preuves pour B. In Valérie M. Morain, editor, *Journées Francophones des Langages Applicatifs (JFLA 2004)*, pages 37–54. INRIA, 2004.

[Req08] Antoine Requet. Bart: A tool for automatic refinement. In Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, page 345. Springer, 2008.

[RM05] Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In Hurd and Melham [HM05], pages 294–309.

[Sch] Arno Schönegge. Proof obligations for monomorphicity.

[Sch95] Arno Schönegge. Would you ever risk a non-monomorphic specification?, 1995.

[SL00] D. Sabatier and P. Lartigue. The use of the B formal method for the design and the validation of the transaction mechanism for smart card applications. *Formal Methods in System Design*, 17(3):245–272, 2000.

[SV07] Marko Samer and Helmut Veith. On the notion of vacuous truth. In Dershowitz and Voronkov [DV07], pages 2–14.

[TCS] DoD 5200.28-STD: Trusted computer system evaluation criteria. http://csrc.nist.gov/publications/history/dod85.pdf.

[Wer94] Benjamin Werner. *Une théorie des constructions inductives*. Thèse de doctorat, Université Paris 7, 1994.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4), 2009.

[WN04] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *LNCS*, pages 305–320. Springer, 2004.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In Laurence Pierre and Thomas Kropf, editors, *CHARME*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 1999.

Appendix A

Cross References with Coq Developments

The developments discussed in this memoir can be downloaded from the scientific publications pages of the Anssi web site (`http://www.ssi.gouv.fr`); they are made available under CeCILL-B licence.

A.1 Refinement

The various representations of refinement have been developed with Coq version 8.2.

The edition of the sources requires the use of Utf-8 encoding and the *Everson Mono Unicode* font (*evermono.ttf*). The content of the Coq files is as follows:

- *subset_gen.v*, *relation_gen.v*

  Definitions, operations and proofs for subsets and relations.

- *data_ref.v*

  Definitions and proofs for data-refinement.

- *choice_ref.v*

  Definitions and proofs for choice-refinement (with preconditions and guards).

- *gen_ref.v*

  Definitions and proofs for gen-refinement.

- *choice_ref_pre.v*, *choice_ref_guard.v*, *choice_ref_try1.v*, *choice_ref_try2.v*

  Definitions and proofs exploring different (limited) versions of choice-refinement.

- *gen_ref_pre.v*, *gen_ref_guard.v*

  Definitions and proofs exploring different (limited) versions of gen-refinement.

- *bool_choice.v*

  Illustration of the refinement paradox using Coq modules.

A.2 BiCoq

A.2.1 BiCoq 2

BiCoq 2 is the first complete version of the embedding of B in Coq; it was developed with Coq version 8.1. It is considered to be deprecated, and should not be used anymore; it is provided for reference.

The edition of BiCoq 2 sources requires the use of Utf-8 encoding and the BiCoq font (*BiCoq.ttf*). The dependencies are explicited in the file *make.txt*. The content of the Coq files is as follows:

- *Mscratch.v*, *Mbool.v*, *Mindex.v*, *Mlist.v*

  Bootstrap not including any element specific to the B method.

- *Bterm.v*

  Definition of B terms, induction principle, equality, non-freeness, fresh variables.

- *Blift.v*, *Binst.v*, *Bbind.v*, *Baffec.v*

  Lifting, instantiation (that is functional application), binding (that is functional abstraction), affectation (that is meta-language substitution) and associated results, including commutation lemmas.

- *Bgamma.v*, *Binfer.v*

  Definition of proof environments (as lists of predicates) and B provability predicate.

- *Bprop.v*, *Biffres.v*, *Bpred.v*, *Bset.v*

  Results for propositional calculus, predicate calculus and set theory.

- *Bdbinfer.v*

  Raw B inference rules and theorems.

- *Bcoqinfer.v*, *Binfer2.v*

  Correspondance between Coq and B logical operators, and derived lemmas for B proofs in Coq.

- *Btrmind.v*, *Binfind.v*

  Semantic induction on terms and induction on proofs.

- *Baffprd.v*

  Results for the substitution of (unbound) equivalent predicates.

- *Bmaffec.v*

  Definition of multiple affectations (that is parallel substitutions) as lists of pairs, and associated theorems.

- *Brawprd.v*

  Weak version of the congruence results for the substitution of bound equivalent predicates.

- *Pinfer.v*, *Pprop.v*, *Pinfer2.v*, *Paffprd.v*, *Prawprd.v*, *Ppred.v*, *Pset.v*

  B tactics for the proven prover.

A.2.2 BiCoq 3

BiCoq 3 is the second complete version of the embedding of B in Coq; it was developed with Coq version 8.2. We describe thereafter the content of BiCoq 3I (based on *de Bruijn* indexes), which has been developed further than BiCoq 3L (based on *de Bruijn* levels).

The edition of BiCoq 3 sources requires the use of Utf-8 encoding and the *Everson Mono Unicode* font (*evermono.ttf*). The dependencies are explicited in the file *make.txt*, the notations in the file *notations.txt*, the tactics in the file *tactics.txt*. The content of the Coq files is as follows:

- *basic_logic.v*

  Results for booleans and correspondance between predicates and boolean functions.

- *basic_nat.v*

  Results for natural values and generic induction principle based on a measure.

- *index.v*

  Definition of *de Bruijn* indexes with namespace, lifting of indexes and identification of dangling indexes (at a given λ-height).

- *term.v*

  Definition of B terms, freeness, fresh variables.

- *lift.v*

  Lifting of B terms and associated theorems.

- *abstr.v*

  Functional abstraction for B terms, definition of notations mimicking natural representations for B binders, and associated theorems.

- *apply.v*

  Functional application for B terms and associated theorems.

- *esubst.v*, *psubst.v*

  Meta-language substitution of expression and predicates for B terms and associated theorems.

- *egraft.v*

  Grafting of expression for B terms and associated theorems.

- *cmp.v*

  Results about the composition of functional abstraction, functional application and meta-language substitution: substitution is not primitive, representation is α-quotiented, and it is always possible to build a functional representation for any term.

- *proof_env.v*, *proof.v*

  Definition of proof environments as an abstract datatype, and definition of the B provability predicate.

- *lemma.v*

  Trivial lemmas completing the B inference rules.

- *bbprop.v*

  B-Book results for propositional calculus and associated Ltac tactic.

- *congr.v*

  Standard B congruence results used as lemmas for advanced results.

- *bbpred.v*

  B-Book results for predicate calculus.

- *bbfix.v*

  Stub (incomplete file) for the proof of B fixpoint results.

- *bbset.v*

  A few results about B set theory.

- *swap.v*

  Relationships between B and Coq logical operators.

- *raw_proof.v*

  Raw B inference rules, expressed directly in the *de Bruijn* representation instead of the functional representation.

- *sem_ind.v*

  Definition of the semantic accessibility relation, associated with a proof that all B terms are accessible by this relation.

- *wf_proof.v*

  Definition of a measure for proofs, and associated induction principle.

- *mesubst.v*

  Definition of parallel substitutions as maps, and associated operations (such as lifting and application to a term). Representation of standard operations on B terms as parallel substitutions. Composition of parallel substitutions.

- *mesubst_framed.v*

  Definition of framed parallel substitutions (maps whose scope is limited by a list), and proof of a form of induction principle for parallel substitutions applied to B terms. Proof of a semantic result about pseudo-ground sequents.

- *egraft_congr.v*

  Full version of congruence results for the substitutions of bound expressions.

- *bmeta.v*

  Considerations about the consistency of the B logic.

- *gsl.v*

  Definition of the B Generalized Substitution Language (Gsl), and representation of the semantics of predicate transformers through a functional application. Considerations about the refinement.

A.3 $\lambda_{\mathbf{Tdb}}$ representation

The formalization of the λ_{Tdb} representation has been developed with Coq version 8.2.

The edition of the sources requires the use of Utf-8 encoding and the *Everson Mono Unicode* font (*evermono.ttf*). The dependencies are explicited in the file *make.txt*, the notations in the file *notations.txt*. The content of the Coq files is as follows:

- *term_weak.v*, *term.v*

  Definition of types, indexes, contexts (as maps) and terms. Typing of a term. Lifting of a term. Binder elimination (that is application followed by reduction), functional abstraction, substitution, grafting, freeness.

  term_weak.v is the standard version, the *term.v* the version with improved context management.

- *beta.v*

  Definition of normal forms and β-reduction, asssociated properties.

- *tech.v*

  Technical lemmas.

- *results.v*

  Progress lemma, weak confluence.