Yet Another Deep Embedding of B:Extending de Bruijn Notations

Eric Jaeger, Thérèse Hardin

Yet Another Deep Embedding of *B*: Extending *de Bruijn* Notations

Éric Jaeger[1,2] and Thérèse Hardin[1]

[1] LIP6, UPMC, 4 place Jussieu, 75252 Paris Cedex 05, France
[2] DCSSI, 51 boulevard de La Tour-Maubourg, 75700 Paris 07 SP, France

Abstract. We present BiCoq$_3$, a deep embedding of the B system in Coq, focusing on the technical aspects of the development. The main subjects discussed are related to the representation of sets and maps, the use of induction principles, and the introduction of a new *de Bruijn* notation providing solutions to various problems related to the mechanisation of languages and logics.

Key words: formal methods, deep embedding, *de Bruijn* notation

Embedding a language or a logic is now a well-established practice in the academic community, to answer various types of concerns, *e.g.* normalisation of terms and influence of reduction strategies for a programming language or consistency for a logic. It indeed supports such meta-theoretical analyses as well as comparing and promoting interesting concepts and features of other languages, or developing mechanically checked tools to deal with a language.

But a lot of difficulties arise that have to be addressed. First of all, an important design choice has to be made between *shallow* and *deep* approaches, consistently with the objectives of the embedding. Justifying the validity of an embedding – its correctness and completeness – can also be difficult. Finally, a lot of technical details have to be considered *e.g.* to manage variables.

We address these questions through the presentation of BiCoq and BiCoq$_3$, two versions of a deep embedding of the B logic in the Coq system. The main objective for these embeddings is to evaluate the correctness of the B method itself, in the context of security developments; other objectives include the development of proven tools for the B and the derivation of new results about the B logic. Yet we focus in this paper on the *technical* aspects of these embeddings, and explain the need for a full redevelopment between the two versions by describing painfully learned lessons. The presentation includes the definition of an extended *de Bruijn* notation with interesting potentialities to solve some frequently encountered problems related to the mechanisation of languages.

This paper is divided into 6 sections. Sections 1-3 briefly introduce Coq, the notion of embedding and B. Section 4 presents *de Bruijn* notations. The technical aspect of the development of BiCoq and BiCoq$_3$ are described in Sec. 5, considering in particular *de Bruijn* context management, induction principles, techniques to implement maps and new results obtained through an extension

of the *de Bruijn* notation using namespaces. Section 6 concludes and identifies further activities.

1 About *Coq*

Coq [1] 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 language with pattern-matching. It is possible in Coq to define values and types, including dependent types (*i.e.* types that explicitly depend on values); types of sort **Set** represent sets of computational values, while types of sort **Prop** represent logical propositions. When defining an inductive type – which is a least fixpoint – associated structural induction principles are automatically generated.

For the intent of this paper, it is sufficient to see Coq as allowing for the manipulation of inductive sets of terms and inductive logical properties. Let's consider the following standard example:

$$\textbf{Inductive}\ \mathbb{N}:\textbf{Set}:=0:\mathbb{N}\mid S:\mathbb{N}\to\mathbb{N}$$
$$\textbf{Inductive}\ \mathrm{even}:\mathbb{N}\to\textbf{Prop}:=ev_0:\mathrm{even}\ 0\mid ev_2:\forall(n:\mathbb{N}),\ \mathrm{even}\ n\to\mathrm{even}\ S(S\ n)$$

The first line defines a type $\mathbb{N}$ which is 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, structural induction on $\mathbb{N}$ is possible. The second line defines a family of *logical types*: $even\ 0$ is a type inhabited by the term ev_0, $even\ 2$ is an other type inhabited by $(ev_2\ 0\ ev_0)$, and $even\ 1$ is an empty type. The standard interpretation is that ev_0 is a proof of the proposition $even\ 0$ and that there is no proof of $even\ 1$, that is we have $\neg(even\ 1)$. The intuitive view of our example is that $\mathbb{N}$ is a set of terms, and *even* a predicate marking some of them, defining a subset of $\mathbb{N}$.

2 Deep and Shallow Embeddings

Embedding in a proof assistant consists in mechanizing a *guest* logic by encoding its syntax and semantic into a *host* logic ([2,3,4]). In a *shallow* embedding, the encoding is partially based on a direct translation of the guest logic into constructs of the host logic; in terms of programming languages, a shallow embedding can intuitively be seen as the development of a translation function between two langages, that is a compiler. On the contrary, a *deep* embedding is better intuitively described as the development of a virtual machine: the syntax and the semantic of the guest logic are formalised as datatypes of the host logic. Taking the view presented in Sec. 1, the deep embedding of a logic defines the set of all sequents – the terms – and the subset of provable sequents (the inference rules of the guest logic being encoded as constructors of the provability predicate).

Both approaches have pros and cons. The one we are concerned with, and that has led us to choose the deep embedding approach, is *accuracy*: a deep embedding

allows for an exact representation of the syntax and semantic of the guest logic, whereas a shallow embedding appears to enforce a form of interpretation whose validity can be difficult to justify.

3 About *B*

3.1 A Short Description of *B*

B [5] is a popular formal method that allows for the derivation of correct programs from specifications. Several industrial implementations are available (*e.g.* AtelierB, B Toolkit), and it is widely used by both the academic world and the industry for projects where safety or security is mandatory.

The B method defines a first-order predicate logic completed with elements of set theory, a *Generalised Substitution Language* (GSL) and a methodology of development based on the explicit concept of *refinement*.

The logic is used to express preconditions, invariants, etc. and to conduct proofs. This logic is not typed; a kind of well-formedness checking is described but is not integrated within the logic.

The GSL allows for the definitions of a form of *Hoare* substitutions [6,7,8] that can be abstract, declarative and non-deterministic (*i.e.* specifications) as well as concrete, imperative and deterministic (*i.e.* programs): the substitution **ANY** x **WHERE** $x^2 \leq n < (x+1)^2$ for example specifies $x \leftarrow \sqrt{n}$.

Regarding the methodology, B developments are made of *machines* (modules combining a *state* in the form of variables, *invariants* and *operations* described as generalised substitutions to read or alter the state). Intuitively a machine M_C *refines* a machine M_A if any observable behaviour of M_C is a possible behaviour of M_A – this encompasses the notion of correctness. Refinement being transitive, it is possible to go progressively from the specification to the implementation; by discharging at each step the *proof obligations* of the B method, a program can be proven to be a correct and complete implementation of a specification.

Note that the language represented by the GSL is imperative; at the last stage of refinement the machines are written using only the B0 sublanguage of the GSL and are easily translated *e.g.* into C programs.

3.2 Embedding *B*: Related Works and Motivations

Shallow embeddings of B in higher-order logics have been proposed in several papers (cf. [9,10]) formalising the GSL in PVS, Coq or Isabelle/HOL. Such embeddings are not dealing with the B logic, and by using directly the host logic to express B notions, they introduce a form of interpretation – which is fully acceptable for example to promote the B methodology in other formal methods.

The objectives of BiCoq and BiCoq$_3$ are very different, the main concern being related to validation. Indeed, the B method is used for the development of safe or secure systems (*e.g.* [11,12]), and it is therefore important to know 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. The other objectives are the development of formally checked tools for B developments, illustrated by a proven prover (not discussed further in this paper but detailed in [13]) and the derivation of new results about the B logic. Regarding the latter, it is again important to be able to justify that such results are not a consequence of the embedding itself, *e.g.* using an 'alien' trick provided by Coq, and are indeed valid for use in a standard B development.

With the objectives of accuracy and independancy, the translation for a shallow embedding would be difficult to define but also to defend against a skeptical independent evaluator. Consider B functions that are relations, possibly partial and undecidable: translating accurately this concept in Coq is a tricky exercise. A deep embedding makes the justification easier, and has also the advantage to clearly separate the host and the guest logics: excluded middle, provable in the B logic as well as in BiCoq or BiCoq$_3$, is not promoted to the Coq logic. Such a deep embedding of the B logic in Coq is described in [14], to validate the *base rules* used by the prover of AtelierB – yet not checking standard B results, and without implementation goal.

4 *De Bruijn* Notations

There are numerous problems to deal with when mechanising a language (cf. [15,16]), one of them being related to the representation of bound variables. Indeed, two terms differing only by the names of their bound variables (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 (denoted λ_{V} in this paper); one may also wonder how to compute the reduction of the substitution $[x := E]\lambda x \cdot T$.

De Bruijn notations (cf. [17,18] or more recently [19]) address these problems by encoding bound variables as natural values pointing to a binder; they define an α-quotiented representation, *i.e.* terms equivalent modulo α-renaming are indeed equal. They also provide a clear semantic to deal with capture phenomena applicable between others when considering substitutions.

4.1 *De Bruijn* Indexes: The $\lambda_{\mathbf{dBi}}$ Notation

The most known *de Bruijn* notation uses indexes, that are relative pointers counting binders from the variables (the leaves in the tree representing the term). The value 0 represents the variable bound by the closest parent binder, as illustrated hereafter (*de Bruijn* binders are underlined for the sake of clarity):

λ_{V} notation	$\lambda x \cdot \lambda y \cdot (X_0 + x - y)$
λ_{dBi} notation	$\underline{\lambda\lambda}(2+1-0)$

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. Another possible alternative is to use the *locally nameless notation*; in this case, free variables are represented by names (and are syntactically different of bound variables). We will not consider further this approach that requires to give a specific semantic to dangling pointers or to manage side conditions enforcing terms to be ground (without dangling pointers).

4.2 *De Bruijn* Levels: The λ_{dBl} Notation

Another option when defining a *de Bruijn* representation is to use levels, discussed *e.g.* in [20]. Levels are absolute pointers counting binders from the root of the term; the value 0 then represents the variable bound by the farest parent binder, as illustrated here:

λ_{V} notation	$\lambda x{\cdot}\lambda y{\cdot}(X_0{+}x{-}y)$
λ_{dBl} notation	$\underline{\lambda\lambda}(2{+}0{-}1)$

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; on the other hand, bound levels need frequent renumbering during abstraction or substitution whereas bound indexes are never modified. Other pros and cons of these approaches will be considered later in the paper to explain BiCoq$_3$ design choices.

4.3 Managing *de Bruijn* Indexes in λ-Calculus

As mentioned, the index representing a given bound variable change with its λ-*height*, *i.e.* the number of parent binders, as illustrated by this example:

λ_{V} notation	$\lambda x{\cdot}(x{+}\lambda y{\cdot}(x{-}y)X_0)$
λ_{dBi} notation	$\underline{\lambda}(0{+}\underline{\lambda}(1{-}0)2)$

This makes 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* or user-relevant such as *substitution*. The former is used by the latter to adapt terms when crossing a binder, as illustrated here (where $\mathbb{T}$ denotes the set of λ_{dBi} terms, i an index in $\mathbb{I}{=}\mathbb{N}$, $\uparrow$ the lifting and $[i{:=}E]T$ the replacement of all occurences of the *free* variable i in T by E):

$$
\begin{array}{l}
\uparrow_d : \mathbb{T} \to \mathbb{T} :=\\
\quad \mid \underline{\lambda} T' \Rightarrow \underline{\lambda}(\uparrow_{d+1} T')\\
\quad \mid i' \Rightarrow \mathbf{if}\ d \leq i'\ \mathbf{then}\ i'+1\ \mathbf{else}\ i'\\
\quad \mid \ldots
\end{array}
\qquad
\begin{array}{l}
[i := E] : \mathbb{T} \to \mathbb{T} :=\\
\quad \mid \underline{\lambda} T' \Rightarrow \underline{\lambda}([x+1 := \uparrow E] T')\\
\quad \mid i' \Rightarrow \mathbf{if}\ i = i'\ \mathbf{then}\ E\ \mathbf{else}\ i'\\
\quad \mid \ldots
\end{array}
$$

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 semantic as well as to avoid their capture – this is the role of lifting. To identify dangling indexes, lifting is parameterised by the *contextual information* d recording the current λ-height, left implicit when $d{=}0$ (other values of d resulting only from recursive calls for bound subterms).

This toolbox for λ-calculus is completed with operators defining a user-friendly representation, as in [18]. The idea is to emulate the λ_{V} abstraction, a not so simple transformation in λ_{dBi} as illustrated here (capturing X_1):

$$\begin{array}{lllll} \lambda_{\mathrm{V}} \text{ notation} & X_0{+}X_1{+}X_2 & \rightarrow & \lambda x{\cdot}(X_0{+}x{+}X_2) \\ \lambda_{\mathrm{dBi}} \text{ notation} & 0{+}1{+}2 & \rightarrow & \underline{\lambda}(1{+}0{+}3) \end{array}$$

To this end, we define the abstraction function $\lambda(i{\cdot}T){:=}\underline{\lambda}(\mathrm{Abstr}_0\ i\ T)$ with:

$$\begin{array}{l} \mathrm{Abstr}_d(i{:}\mathbb{I}){:}\mathbb{T}\rightarrow\mathbb{T}{:=} \\ \quad |\ \underline{\lambda}T' \Rightarrow \underline{\lambda}(\mathrm{Abstr}_{d+1}\ (i{+}1)\ T') \\ \quad |\ i' \Rightarrow \begin{cases} i' \text{ if } i' < d \\ d \text{ if } i' \geq d \text{ and } i' = i \\ i'{+}1 \text{ if } i' \geq d \text{ and } i' \neq i \end{cases} \\ \quad |\ \ldots \end{array}$$

Here $\lambda(i{\cdot}T)$ is not the λ_{V} abstraction but a *function* computing the correct λ_{dBi} term, defining a *form* of λ_{V} representation (i being an index and T a λ_{dBi} term).

5 A Detailed presentation of *BiCoq3*

We now discuss the design choices made for developing BiCoq$_3$, also addressing the technical alternatives and their consequences. From this point, illustrations and codes will describe the B logic as encoded in Coq, instead of the λ-calculus considered up to now; dotted notations will represent B logical operators in Coq (*e.g.* $\neg$ is the Coq negation and $\dot{\neg}$ the embedded B negation).

5.1 Embedding the Syntax

Using *de Bruijn* indexes. We have chosen for BiCoq and BiCoq$_3$ to use a *de Bruijn* notation, and have investigated both indexes and levels: two *full* versions of BiCoq$_3$ 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):

$$\begin{array}{l} \uparrow_d{:}\mathbb{T}\rightarrow\mathbb{T}{:=} \\ \quad |\ \underline{\lambda}T' \Rightarrow \underline{\lambda}(\uparrow_{d+1} T') \\ \quad |\ i' \Rightarrow \textbf{if } d{\leq}i' \textbf{ then } i'{+}1 \textbf{ else } i' \\ \quad |\ \ldots \end{array} \qquad \begin{array}{l} \uparrow^{\mathrm{L}}{:}\mathbb{T}\rightarrow\mathbb{T}{:=} \\ \quad |\ \underline{\lambda}T' \Rightarrow \underline{\lambda}(\uparrow^{\mathrm{L}} T') \\ \quad |\ i' \Rightarrow i'{+}1 \\ \quad |\ \ldots \end{array}$$

As mentioned in Sub. 4.3, $\uparrow_d$ requires a contextual parameter to identify dangling indexes, bound indexes being never modified. On the contrary its λ_{dBl} equivalent $\uparrow^{\mathrm{L}}$ increments all levels, so this parameter is not required and theorems about lifting are not specialised according to its value.

Our final (and late) choice is however to use *de Bruijn* indexes. Indeed complex results in our developement require as a proof tool the definition of *parallel λ-substitutions* providing an alternative encoding of standard operations on

terms (such as lifting). This is feasible with λ_{dBi}, those operations being similar to substitutions in never modifying bound indexes, but not in λ_{dBl}. We therefore consider that whereas *de Bruijn* levels are simpler to use, there is a clear advantage for *de Bruijn* indexes when dealing with *advanced* techniques related *e.g.* to term transformations under binders detailed later in this paper.

Representing *B* terms. Given a set of identifiers I, the B logic syntax defines predicates P, expressions E, sets S and variables V as follows:

$$\begin{array}{rcllllllll} P & := & P\wedge P & \mid P\Rightarrow P & \mid \neg P & \mid \forall V\cdot P & \mid E=E & \mid E\in S & \mid [V:=E]P \\ E & := & V & \mid S & \mid E\mapsto E & \mid \Downarrow S & \mid [V:=E]E & & \\ S & := & \mathbf{BIG} & \mid \Uparrow S & \mid S\times S & \mid \{V\mid P\} & & & \\ V & := & I & \mid V,V & & & & & \end{array}$$

In this syntax, $[V:=E]T$ represents the (elementary) substitution, V_1, V_2 a list of variables, $E_1 \mapsto E_2$ a pair of expressions, $\Downarrow$ and $\Uparrow$ the *choice* and *powerset* operators, and **BIG** a constant set. Other connectors are standard, and new connectors are defined from the previous ones, $P\Leftrightarrow Q$ as $P\Rightarrow Q \wedge Q\Rightarrow P$, $P\vee Q$ as $\neg P\Rightarrow Q$, $\exists V\cdot P$ as $\neg\forall V\cdot\neg P$, $S\subseteq T$ as $S\in\Uparrow T$, etc.

The B syntax is formalised in Coq by two mutually inductive types with the following constructors[3], $\mathbb{I}$ being the set of indexes (*i.e.* $\mathbb{N}$):

$$\begin{array}{rcllllllll} \mathbb{P} & := & \mathbb{P}\dot\wedge\mathbb{P} & \mid \mathbb{P}\dot\Rightarrow\mathbb{P} & \mid \dot\neg\mathbb{P} & \mid \underline{\forall}\mathbb{P} & \mid \mathbb{E}\doteq\mathbb{E} & \mid \mathbb{E}\dot\in\mathbb{E} & \\ \mathbb{E} & := & \dot\chi\mathbb{I} & \mid \mathbb{E}\dot\mapsto\mathbb{E} & \mid \dot\Downarrow\mathbb{E} & \mid \dot\Omega & \mid \dot\Uparrow\mathbb{E} & \mid \mathbb{E}\dot\times\mathbb{E} & \mid \{\!\underline{|}\,\mathbb{E}\,\underline{|}\,\mathbb{P}\,\underline{\}}\!\} \end{array}$$

$\mathbb{P}$ represents B predicates and $\mathbb{E}$ merges B expressions E, sets S and variables V to enrich the B syntax that is too strict (*e.g.* $E\in\Downarrow(\Uparrow S)$ is syntactically invalid in standard B). In the rest of this paper $\mathbb{T}=\mathbb{P}\cup\mathbb{E}$ denotes the type of B terms.

$\dot\Omega$ represents the constant set **BIG**, $\dot\chi$ unary *de Bruijn* variables (using $\dot\chi_i$ to denote the application of constructor $\dot\chi$ to $i:\mathbb{I}$). The B binders $\forall V\cdot P$ and $\{V\mid P\}$ are respectively represented by the constructors $\underline{\forall}$ and $\{\!\underline{|}\,\underline{\}}\!\}$, that are raw *de Bruijn* binders (we therefore use the underlined notation, the dotted notation $\dot\forall$ and $\{\!\dot|\dot\}\!\}$ being reserved for a user-friendly notation, cf. Sub. 4.3). Using *de Bruijn* indexes, they have no attached names and only bind a single variable – binding over list of variables being eliminated without loss of expressivity[4]. The constructor $\{\!\underline{|}\,\underline{\}}\!\}$ is further modified to keep in the syntax definition only well-formed terms (cf. Sub. 3.1). Indeed, the well-formedness checking in B requires comprehension sets to be of the form $\{x \mid x\in S\wedge P\}$ with x not free in S. *Both* constraints are embedded in our syntax. The comprehension set constructor has two parameters, the left one being an expression representing S and the right one a predicate representing P; the non-freeness condition is ensured by considering

[3] This is a slightly simplified presentation of BiCoq$_3$ focusing on relevant aspects.

[4] Remark by the way that the notation $\{V_1, V_2 \mid V_1, V_2 \in S_1\times S_2\wedge P\}$ used in [5] is an example of syntactically invalid term confusing the expression $x\mapsto y$ with the variable x,y, whose 'correct' version $\{V_1,V_2\,|\,V_1\mapsto V_2\in S_1\times S_2\wedge P\}$ is not well-formed.

this constructor as a binder only for its right parameter[5]. This bridges the gap between syntactically correct terms and well-formed ones.

Note finally that we do not represent B syntactical constructs $[V := E]T$ (elementary substitutions); this will be justified later in this paper.

5.2 *De Bruijn* Management: Improving Context Awareness

We ease the use of the *de Bruijn* notations by providing functions, as in Sub. 4.3. First of all, lifting is adapted to our constructors – noting that as $\{\!|\,|\!\}$ does not bind its left parameter, the left λ-height is not incremented:

$$\begin{array}{lll}
\uparrow_d : \mathbb{T} \to \mathbb{T} := \underline{\forall} P' & \Rightarrow \underline{\forall}(\uparrow_{d+1} P') \\
\quad \mid \{\!| E' \underline{\downarrow} P' |\!\} & \Rightarrow \{\!| \uparrow_d E' \underline{\downarrow} \uparrow_{d+1} P' |\!\} \\
\quad \mid \dot{\chi}_{i'} & \Rightarrow \dot{\chi}(\textbf{if } d \leq i' \textbf{ then } i'+1 \textbf{ else } i') \\
\quad \mid \ldots & \Rightarrow \ldots \text{(straightforward recursion)}
\end{array}$$

We also define abstraction functions, but with *additional* subtle changes:

$$\begin{array}{lll}
\mathrm{Abstr}_d(i : \mathbb{I}) : \mathbb{T} \to \mathbb{T} := \underline{\forall} P' & \Rightarrow \underline{\forall}(\mathrm{Abstr}_{d+1}\ (\uparrow_d i)\ P') \\
\quad \mid \{\!| E' \underline{\downarrow} P' |\!\} & \Rightarrow \{\!| \mathrm{Abstr}_d\ i\ E' \underline{\downarrow} \mathrm{Abstr}_{d+1}\ (\uparrow_d i)\ P' |\!\} \\
\quad \mid \dot{\chi}_{i'} & \Rightarrow \dot{\chi}(\textbf{if } i = i' \textbf{ then } d \textbf{ else } \uparrow_d i) \\
\quad \mid \ldots & \Rightarrow \ldots \text{(straightforward recursion)}
\end{array}$$

$$\dot{\forall}\, i \cdot P := \underline{\forall}(\mathrm{Abstr}_0\ i\ P) \qquad \dot{\exists}\, i \cdot P := \dot{\neg}(\dot{\forall} i \cdot \dot{\neg} P) \qquad \dot{\{} i : E \mid P \dot{\}} := \{\!| E \underline{\downarrow} \mathrm{Abstr}_0\ i\ P |\!\}$$

Compared with the abstraction function defined in Sub. 4.3, it is important to note the difference w.r.t. the λ-height parameter d. We do not *increment* indexes anymore but we *lift* them; furthermore when we lift an expression, we ensure that we use d instead of the default value 0. This does not change the result: applying n times the function $\uparrow_0$ yields exactly the same result as applying successively $\uparrow_0, \uparrow_1, \ldots, \uparrow_{n-1}$ – benefits are not *computational* but *logical*. Indeed we have (painfully) discovered that a stricter discipline in managing contexts is a very good practice, easing the expression of theorems as well as their proofs. In fact, this discipline leads to generalise the λ-height parameter to functions *that don't need it*. For example, deciding if a variable appears free in a term does not require this parameter (left code), but proofs are easier by adding it *and* using it to lift the variable parameter (right code):

$$\begin{array}{ll}
\mathrm{Free}(i : \mathbb{I}) : \mathbb{T} \to \mathbb{B} := & \mathrm{Free}_d(i : \mathbb{I}) : \mathbb{T} \to \mathbb{B} := \\
\quad \mid \underline{\forall} P' \Rightarrow \mathrm{Free}\ (i+1)\ P' & \quad \mid \underline{\forall} P' \Rightarrow \mathrm{Free}_{d+1}\ (\uparrow_d i)\ P' \\
\quad \mid \{\!| E' \underline{\downarrow} P' |\!\} \Rightarrow \mathrm{Free}\ i\ E' \vee \mathrm{Free}\ (i+1)\ P' & \quad \mid \{\!| E' \underline{\downarrow} P' |\!\} \Rightarrow \mathrm{Free}_d\ i\ E' \vee \mathrm{Free}_{d+1}\ (\uparrow_d i)\ P' \\
\quad \mid \dot{\chi}_{i'} \Rightarrow i' = i & \quad \mid \dot{\chi}_{i'} \Rightarrow i' = i \\
\quad \mid \ldots \Rightarrow \ldots \text{(straightforward recursion)} & \quad \mid \ldots \Rightarrow \ldots \text{(straightforward recursion)}
\end{array}$$

Generalising the λ-height parameter and using it ensures an explicit management of the context, a form of weak typing useful for complex proofs.

[5] Similarly consider the $\lambda x : T \cdot E$ notation in simply-typed λ-calculus; the λ captures x in E but not in T, binding only one of its parameters.

We also define additional functions (not described in Sub. 4.3) to deal with the B syntactical constructs $[V:=E]T$ not represented in our syntax. It is our view that these constructs are introduced early in B *only* for expressing inference rules such as the $\forall$-elimination ($\Gamma \vdash \forall V \cdot P \rightarrow \Gamma \vdash [V := E]P$), that is a form of application followed by β-reduction as in standard λ-calculus; there is no reason to enforce this operation to be the B elementary substitution defined by the GSL... Neither do we represent the application in our syntax, as in standard formalisations of λ-calculus: representing application (and β-reduction either as an external or internal operation *e.g.* using the *explicit substitution* approach [21,22]) is interesting for example to study normalisation strategies, but this is not relevant in our case. We encode *directly* such elimination rules, *i.e.* application followed by β-reduction, as an external operation, through *application functions* in Coq denoted $T@_\forall E$ and $T@_{\{\}} E$, one per binder[6]:

$$\begin{array}{lll}
\mathrm{App}_d(E:\mathbb{E}):\mathbb{T}\rightarrow\mathbb{T} := \underline{\forall} P' & \Rightarrow \underline{\forall}(\mathrm{App}_{d+1}\ (\uparrow_d E)\ P') \\
\quad \mid \ \{\!\mid E' \mid\!\} P' \ \{\!\mid & \Rightarrow \{\!\mid \mathrm{App}_d\ E\ E' \mid \mathrm{App}_{d+1}\ (\uparrow_d E)\ P' \mid\!\} \\
\quad \mid \ \dot{\chi}_{i'} & \Rightarrow \begin{cases} \dot{\chi}_{i'-1} \text{ if } d < i' \\ E \text{ if } d = i' \\ \dot{\chi}_{i'} \text{ if } d > i' \end{cases} \\
\quad \mid \ \ldots & \Rightarrow \ldots \text{(straightforward recursion)}
\end{array}$$

$$T@_\forall E := \textbf{match}\ T\ \textbf{with}\ \underline{\forall} T' \Rightarrow \mathrm{App}_0\ E\ T'$$

$$T@_{\{\}} E := \textbf{match}\ T\ \textbf{with}\ \{\!\mid E' \mid T' \mid\!\} \Rightarrow E \doteq E' \dot{\wedge} \mathrm{App}_0\ E\ T'$$

The $\forall$-elimination can then be written $\Gamma \vdash \forall V \cdot P \rightarrow \Gamma \vdash (\forall V \cdot P)@_\forall E$. As abstraction, application and substitution functions are such that the following properties hold (the left one being valid only after generalising the λ-height parameter to the substitution function), our rule is equivalent to the standard one:

$$[i := E]_d T = \mathrm{App}_d\ E\ (\mathrm{Abstr}_d\ i\ T) \quad \text{or more simply} \quad [i := E]T = (\dot{\forall} i \cdot P)@_\forall E$$

The point is that we do not consider substitution as primitive. The standard definition of β-reduction $\lambda x \cdot T@E \rightarrow_\beta [x := E]T$ describes the semantic of application using substitution; in BiCoq$_3$ on the contrary application is directly defined and the substitution is a composite operation. Note also that we can write $\mathrm{App}_d\ \dot{\chi}_i\ (\mathrm{Abstr}_d\ i\ T) = T$, or more simply $(\dot{\forall} i \cdot P)@_\forall \dot{\chi}_i = T$, to emphasise that application is the reverse of abstraction[7].

5.3 Embedding the Inference Rules

Having formalised the B syntax as a datatype, the next step is to encode the B inference rules as the constructors of an inductive *provability* predicate defining a dependent type. We denote $\Gamma \dot{\vdash} P$ the Coq type of all B proofs of P under the assumptions Γ; if it is inhabited then P is provable assuming Γ. Note that $\neg(\Gamma \dot{\vdash} P)$, *i.e.* '$\Gamma \dot{\vdash} P$ is an empty type', is different from $\Gamma \dot{\vdash} \dot{\neg} P$.

[6] These functions only apply to terms starting with the appropriate binder; the partiality is encoded in Coq by an additonal proof parameter left implicit here.

[7] This result commutes, $\mathrm{Abstr}_d\ i\ (\mathrm{App}_d\ \dot{\chi}_i\ T) = T$ provided that $\mathrm{Free}_d\ i\ \underline{\forall} T = \bot$.

Thanks to the use of the user-friendly functions described in Subs. 4.3 and 5.2, the constructors look very much like the standard B rules[8]. The translation is straightforward, merely a syntactical one, limiting the risk of error, as illustrated here (where $V \backslash \Gamma$ means that V does not appear free in Γ):

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \quad \text{is encoded by} \quad \Gamma \mathrel{\dot\vdash} P \to \Gamma \mathrel{\dot\vdash} Q \to \Gamma \mathrel{\dot\vdash} P \mathbin{\dot\wedge} Q$$

$$\frac{\Gamma \vdash P \quad V \backslash \Gamma}{\Gamma \vdash \forall V \cdot P} \qquad\qquad i \mathbin{\dot\setminus} \Gamma \to \Gamma \mathrel{\dot\vdash} P \to \Gamma \mathrel{\dot\vdash} \dot\forall\, i \cdot P$$

The main divergence is a correction of the definition of the cartesian product. Indeed, beyond minor syntactical problems, BiCoq has also pointed out B logical oversights; analyses have shown that the following results, presented in [5] as theorems, are in fact not provable with the standard B inference rules[9]:

$$\vdash E_1 \mapsto F_1 = E_2 \mapsto F_2 \Rightarrow E_1 = E_2 \wedge F_1 = F_2$$
$$\vdash S_1 \subseteq S_2 \wedge T_1 \subseteq T_2 \Rightarrow S_1 \times T_1 \subseteq S_2 \times T_2$$

To our knowledge, this was not known by the B community – whereas implementations of the B method correct this flaw, consciously or not. The flawed rule $\vdash (E \mapsto F) \in (S \times T) \Leftrightarrow (E \in S) \wedge (F \in T)$ has therefore been replaced in BiCoq by:

$$\Gamma \mathrel{\dot\vdash} E_1 \mathrel{\dot\mapsto} E_2 \doteq E_3 \mathrel{\dot\mapsto} E_4 \to \Gamma \mathrel{\dot\vdash} E_1 \doteq E_3 \mathbin{\dot\wedge} \dot\vdash E_2 \doteq E_4$$
$$i_1, i_2 \mathbin{\dot\setminus} E \mathrel{\dot\in} (E_1 \mathbin{\dot\times} E_2) \to i_1 \neq i_2 \to \Gamma \mathrel{\dot\vdash} \dot\exists\, i_1 \cdot i_1 \mathrel{\dot\in} E_1 \mathbin{\dot\wedge} \dot\exists\, i_2 \cdot i_2 \mathrel{\dot\in} E_2 \mathbin{\dot\wedge} E \doteq i_1 \mathrel{\dot\mapsto} i_2 \mathrel{\dot\Leftrightarrow} E \mathrel{\dot\in} (E_1 \mathbin{\dot\times} E_2)$$

5.4 A Generic Induction Principle

The definition of an inductive datatype in Coq yields automatically the associated structural induction principle. This principle is relevant to prove structural properties such as those about freeness, but not to prove *semantical* results.

Indeed, it identifies T as the predecessor of $\underline{\forall} T$, *i.e.* that proving $P(T)$ by structural induction requires proving a subgoal of the form $P(T') \Rightarrow P(\underline{\forall} T')$. But using *de Bruijn* indexes this approach is not appropriate:

de Bruijn indexes	$\underline{\exists}(1 * 0 > 2)$	$\underline{\forall}(\underline{\exists}(1 * 0 > 2))$
Natural notation	$\exists\, z \cdot X_0 * z > X_1$	$\forall\, y \cdot \exists\, z \cdot y * z > X_0$

The two *de Bruijn* terms are related structurally, but not semantically because of the unmonitored shift of the context modifying free variables representation.

To address this problem and some others, numerous induction principles were derived in BiCoq : (weak) structural induction, semantical induction, strong induction based on a measure for a given type or for mutually recursive types. And this was not yet sufficient for proof induction because the predecessors (sub-proofs) of a step in a proof have different (dependent) types. This was not considered as a proper approach, because of the number of principles to be expressed and proved as well as the absence of genericity of the proof method.

[8] We also benefit from the **Notation** command provided by Coq to use UTF-8 symbols instead of constructors or functions names.

[9] Further details are discussed in [13].

For BiCoq$_3$ a general approach has been designed. It combines a single induction principle based on a measure in $\mathbb{N}$ (something rather intuitive) with a strategy for conducting the proof defined through a inductive relation (so-called *accessibility relation*). The induction principle is generic, as D is any family of types (indexed by T), M any measure and P any predicate:

$$\forall\,(T\!:\!\mathbf{Type})(D\!:\!T\to\mathbf{Type})(M\!:\!\forall\,(t\!:\!T),\ D\,t\to\mathbb{N})(P\!:\!\forall\,(t\!:\!T),\ D\,t\to\mathbf{Prop}),\\ (\forall\,(t\!:\!T)(d\!:\!D\,t),(\forall\,(t'\!:\!T)(d'\!:\!D\,t'),\ M\,t'<M\,t\to\ P\,t')\to P\,t)\to\forall\,(t\!:\!T),\ P\,t$$

It does not describe what are the 'smaller' terms to consider – this results of the selected accessibility relation. Choosing this relation is choosing the strategy, the cases in a proof by cases, the predecessors for the entity you are considering. Intuitively, this defines paths to reach terms in D, and provided the measure is compatible with the relation (*i.e.* predecessors are smaller) 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. For example a semantically relevant strategy can be defined as follows:

$$\begin{array}{l}\mathbf{Inductive}\ \Sigma_{\mathrm{Sem}}\!:\!\mathbb{T}\to\mathbf{Type}\!:=\\ \mid\Sigma_{\chi}\!:\!\forall\,(i\!:\!\mathbb{I}),\ \Sigma_{\mathrm{Sem}}\,\dot{\chi}i\\ \mid\Sigma_{\forall}\!:\!\forall\,(P\!:\!\mathbb{P})(i\!:\!\mathbb{I}),\ \Sigma_{\mathrm{Sem}}\,P\to\Sigma_{\mathrm{Sem}}\,\dot{\forall}\,i\cdot P\\ \mid\Sigma_{\{\}}\!:\!\forall\,(P\!:\!\mathbb{P})(E\!:\!\mathbb{E})(i\!:\!\mathbb{I}),\ \Sigma_{\mathrm{Sem}}\,P\to\Sigma_{\mathrm{Sem}}\,E\to\Sigma_{\mathrm{Sem}}\,\dot{\{}i\!:\!E\mid P\dot{\}}\\ \mid\ldots(\text{straightforward induction})\end{array}$$

This relation is surjective, *i.e.* $\forall\,(T:\mathbb{T}),\ \Sigma_{\mathrm{Sem}}(T)$. To prove a property 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 using inversion of the Coq term $\Sigma_{\mathrm{Sem}}(T)$. The generated subgoals are then semantically relevant, *e.g.* $Q\,E'\to Q\,P'\to Q\,\dot{\{}i'\!:\!E'\mid P'\dot{\}}$.

5.5 About Lists, Maps and Abstract Data Types

Various syntactical entities are represented in our embedding, including sequents and parallel substitutions (used as a technical tool to prove complex results presented thereafter). In BiCoq these constructs are implemented through lists, but we have explored other alternatives in BiCoq$_3$.

Proof environments in sequents are finite sets of predicates. In BiCoq$_3$ they are represented by a specification: signature of functions for membership, freeness, etc. with the appropriate properties as axioms. The specification has the advantage to describe only what we *need* to know, and permits to use efficient concrete functions of the target language when there is an implementation objective[10]. Yet we do *not* recommend this approach for a deep embedding, as the workload is not significantly reduced, whereas there is a risk to introduce inconsistent axioms.

[10] E.g. BiCoq$_3$ specifies the terms equality to use OCaml's = in the implementation.

Another possibility adopted in BiCoq$_3$ is the use of maps to represent parallel substitutions. They can be described as lists of pairs in $\mathbb{I}\times\mathbb{E}$ provided that there are never two pairs (i, E) and (i, E') s.t. $E \neq E'$, but it is more efficient to consider them as functions in $\mathbb{I}\to\mathbb{E}$. In our experience, this approach simplifies the development and the proofs – consider the use of parallel substitutions to represent lifting: it is not possible to build a generic lift substitution using finite lists, because any index i that can appear dangling in a term T has to be modified, whereas a unique (infinite) map can represent lifting for any term. 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. Yet the main results consider parallel substitutions applied to a term, for which well-foundedness holds. A more straightforward approach, yet to be explored, would be to reintroduce well-foundedness through scoped maps, that is parallel substitutions represented by elements of $(\mathbf{List}\,\mathbb{I})\times(\mathbb{I}\to\mathbb{E})$, the list enumerating the relevant indexes.

Maps are therefore efficient tools for deep embeddings, but our recommandation would be to carefully analyse *all* consequences of using such a design. For example, they cannot be analysed extensionally – just another way to say that they are not well-founded. That means in practice *e.g.* that as we need to be able to decide whether or not a variable appears free in (one of the predicates of) a proof environment Γ, we cannot encode Γ as a function in $\mathbb{P}\to\mathbb{B}$. Indeed, being unable to identify *a priori* predicates of Γ, testing freeness would require examining *all* predicates in the (infinite) type $\mathbb{P}$.

5.6 Relationships between *B* and *Coq* logics

Deep embeddings such as BiCoq and BiCoq$_3$ ensure a clear separation of the host and the guest logics, allowing *e.g.* 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:

$$\begin{array}{rcl}
\Gamma \vdash P_1 \mathbin{\dot\wedge} P_2 & \Leftrightarrow & (\Gamma \vdash P_1) \wedge (\Gamma \vdash P_2) \\
\Gamma \vdash \dot\forall\, i \cdot P & \Leftrightarrow & \forall\,(E{:}\mathbb{E}),\ \Gamma \vdash [i{:=}E]P \\
\Gamma \vdash P_1 \mathbin{\dot\Rightarrow} P_2 & \Rightarrow & \Gamma \vdash P_1 \Rightarrow \Gamma \vdash P_2 \\
\Gamma \vdash P_1 \mathbin{\dot\vee} P_2 & \Leftarrow & (\Gamma \vdash P_1) \vee (\Gamma \vdash P_2) \\
\Gamma \vdash \dot\exists\, i \cdot P & \Leftarrow & \exists\,(E{:}\mathbb{E}),\ \Gamma \vdash [i{:=}E]P \\
\Gamma \vdash E_1 \doteq E_2 & \Leftarrow & E_1 = E_2
\end{array}$$

The interesting results are those that are not equivalences. For example disjunction ($\vee$ vs $\dot\vee$) is very significant w.r.t. the difference between the classical logic of B and the constructive logic of Coq. The *excluded middle* being provable in B, it is always possible to provide a proof of $\vdash P \mathbin{\dot\vee} \dot\neg P$; should the disjunction being directly translated in Coq we would obtain $(\vdash P) \vee (\vdash \dot\neg P)$ for any P, that is a proof that the B logic is complete, which of course is not the case.

Note that these results provide a formal justification for the translation in a shallow embedding; one may wonder whether it would be possible to automatically derive (or extract) a shallow embedding from a deep embedding, provided such results.

5.7 New Results and Enriched *de Bruijn* Indexes

Using Standard Indexes. The B inference rules defined in [5] include a congruence rule: if $\Gamma \vdash E = F$ and $\Gamma \vdash [x := E]P$, then $\Gamma \vdash [x := F]P$. BiCoq generalises this congruence rule to equivalent predicates (extending the syntax with propositional variables). These results, however, are limited to the replacement of *unbound* subterms; that is, they are for example not applicable to systematically simplify $\Gamma \dot{\vdash} \dot{\forall}\, i \cdot (\dot{\neg}\dot{\neg} P)$ into $\Gamma \dot{\vdash} \dot{\forall}\, i \cdot P$ as i may appear free in P.

The substitution operator (left code) indeed mechanically avoid capture of variables by enforcing lifting when crossing a binder. So BiCoq$_3$ also addresses a more generic class of congruence rules by defining *grafting* (right code), which compared to the standard substitution allows for the capture of variables in the parameter E by *never* lifting it:

$$
\begin{array}{l}
[i := E]_d : \mathbb{T} \to \mathbb{T} := \\
\quad \mid \underline{\forall} T' \Rightarrow \underline{\forall}([\uparrow_d i := \uparrow_d E]_{d+1} T') \\
\quad \mid i' \Rightarrow \textbf{if } i' = i \textbf{ then } E \textbf{ else } i' \\
\quad \mid \ldots
\end{array}
\qquad
\begin{array}{l}
[i \lhd E]_d : \mathbb{T} \to \mathbb{T} := \\
\quad \mid \underline{\forall} T' \Rightarrow \underline{\forall}([\uparrow_d i \lhd E]_{d+1} T') \\
\quad \mid i' \Rightarrow \textbf{if } i' = i \textbf{ then } E \textbf{ else } i' \\
\quad \mid \ldots
\end{array}
$$

Grafting being defined, we have proven (using parallel substitutions) in BiCoq$_3$ the following congruence results for the replacement of sub-terms:

$$
\frac{\dot{\vdash} E_1 \doteq E_2}{\Gamma \dot{\vdash} [i \lhd E_1]P \dot{\Leftrightarrow} [i \lhd E_2]P}
\qquad
\frac{\dot{\vdash} E_1 \doteq E_2}{\Gamma \dot{\vdash} [i \lhd E_1]E \doteq [i \lhd E_2]E}
$$

These results extend the classical congruence rules to bound subterms – *e.g.* they justify why it is always valid to simplify a subterm $\neg\neg P$ into P, anywhere in a term. But they are not generic enough, as the equality $E_1 = E_2$ has to be proven in the empty context. So they cannot for example be used to unfold a *conditional* definition such as $y \neq 0 \vdash x/y = \mathbf{max}\{z \in \mathbb{N} \mid y \times z \leq x\}$. This limitation is not logical but technical. Preventing lifting when crossing a binder is necessary to permit captures of variables, but causes a loss of context: free variables representation is modified without control.

Introducing Namespaces. Several approaches were considered to avoid this limitation of the congruence results: using names, marking *De Bruijn* indexes during grafting, defining grafting as the composition of primitive operations... to finally develop for BiCoq$_3$ a simpler solution, *enriched de Bruijn indexes*.

In its most general form, this notation represents free and bound variables by pairs (n, x), the first parameter n being the *namespace* and the second one the index. Binders of the language are themselves parameterised by a namespace in which they capture variables. Namespaces can be seen as sorts, used to mark binders and indexes[11]. 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. This representation

[11] Sorts for *de Bruijn* indexes are considered in [25] but for different reasons, each of the two binders of the defined language using its own space of *de Bruijn* indexes.

defines 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 but dealt with using only decidable equality.

BiCoq$_3$ applies these principles in a simplified manner: the namespace set $\mathcal{N}$ contains (at least) two values, all the binders acting implicitly in the dedicated namespace n_0, the other namespaces being used for eternally free variables. Consistently, lifting only modifies pairs of the form (n_0, x) in a term, etc. It is then possible to prove improved congruence results:

$$\frac{\Gamma \dot{\vdash} E_1 \doteq E_2 \quad \Gamma \perp E_1 \doteq E_2}{\Gamma \dot{\vdash} [i \lhd E_1]P \dot{\Leftrightarrow} [i \lhd E_2]P} \qquad \frac{\Gamma \dot{\vdash} E_1 \doteq E_2 \quad \Gamma \perp E_1 \doteq E_2}{\Gamma \dot{\vdash} [i \lhd E_1]E \doteq [i \lhd E_2]E}$$

The side condition $\perp$ requires Γ and $E_1 = E_2$ to have no common free variable *in the namespace* n_0 – the technical difficulty is still there, but is now limited to a dedicated namespace. Provided we avoid using the namespace n_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 result, *e.g.* allowing for the replacement of conditional definitions. In their most general form, these results allow for β-reduction, unfolding of (conditional) definitions, as well as the replacement (rewriting) of equivalent subterms under a binder.

Applicability of the New Results. As noted in Sub. 3.2, it is important to justify that such new results are truly applicable to B and are not artefacts provable only using features of the host logic. We provide the intuitive justification by the *Curry-Howard* isomorphim. The interpretation of the congruence results is that provided a B proof of $\Gamma \vdash E_1 = E_2$, if $\Gamma \perp E_1 \doteq E_2$ then there *always* exists a B proof of $\Gamma \vdash [i \lhd E_1]P \Leftrightarrow [i \lhd E_2]P$. In fact, the Coq proof *is* a program building such a B proof, the Σ_{Sem} accessibility relation used in the Coq proof (cf. Sub. 5.4) being the recursion strategy of this program.

6 Conclusion

Through the presentation of two deep embeddings of the B logic in Coq, namely BiCoq and BiCoq$_3$, we have discussed techniques to deal with deep embeddings, or more generally with complex developments in higher-order logic (HOL) frameworks – *e.g.* combining a generic induction scheme with *ad hoc* accessibility relations or implementing sets with maps rather than lists. One of these techniques applicable to language mechanisations is to enrich *de Bruijn* representation.

The first proposed adaptation enforces an explicit and precise management of the λ-height parameter – to the extent that it is added to operations that do not strictly require it. This is in fact a form of encoding ensuring a consistent management of the context: not only are proofs easier to conduct, but in some cases it also allows for finer definitions and proofs of properties that would not be valid in a cruder version. Context management is intuitive and don't require to use the full arithmetics: the only required operators on indexes are successor, predecessor and comparison.

The second adaptation introduces *namespaces* to parameterise binders and indexes. It is a way to partition variables and to easily restrict scopes. Again, the required adaptations of the operations are simple and intuitive, but the benefits are in our case important: beyond obtaining the full power of complex congruence results, it is a frequent cause for proof simplifications. Namespaces also define an approach to consider substitution and grafting as a single operation: substitution is emulated by grafting provided free variables are in never bound namespaces.

We may also note that our design choice is to directly encode application as an external operation – *i.e.* a shallow representation of application in our deep embedding. Together, these adaptations of the *de Bruijn* representation seem to define a new form of calculus for languages, of which detailed properties are still to be carefully studied and compared to other calculi (*e.g.* [21,22,23,24]). Clearly, a full version of this calculus easily represents the concept of sorts, provided with an efficient management of contexts.

Taking the user view, these embeddings also demonstrate that it is possible to embed a non trivial logic while ensuring accuracy and readability. Their usefulness to check the validity of known results is illustrated by the identification of various oversights – in our view a sufficient justification for this activity, at least from a security perspective (cf. [26]). The development of proven tools and the derivation of non trivial theorems that were, in our knowledge, not proven in B (without even speaking of formally checked) are additional benefits.

Acknowledgements We thank Pr. C. Dubois for its advices.

References

1. The Coq development team: The Coq proof assistant reference manual. LogiCal Project. (2004)
2. M.J.C. Gordon: Mechanizing programming logics in higher-order logic. In G.M. Birtwistle, P.A. Subrahmanyam, eds.: Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification), Banff, Canada, Springer-Verlag, Berlin (1988) 387–439
3. Boulton, R.J., Gordon, A., Gordon, M.J.C., Harrison, J., Herbert, J., Tassel, J.V.: Experience with embedding hardware description languages in HOL. In Stavridou, V., Melham, T.F., Boute, R.T., eds.: TPCD. Volume A-10 of IFIP Transactions., North-Holland (1992) 129–156
4. Azurat, A., Prasetya, I.: 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)
5. Abrial, J.R.: The B-Book - Assigning Programs to Meanings. Cambridge University Press (August 1996)
6. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10) (1969) 576–580
7. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
8. Hoare, C.A.R.: Programs are predicates. In: FGCS. (1992) 211–218

9. Bodeveix, J.P., Filali, M., Muñoz, C.: 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. (1999) 33–49
10. Chartier, P.: Formalisation of B in Isabelle/HOL. [27] 66–82
11. Behm, P., Desforges, P., Meynadier, J.M.: MÉTÉOR : An industrial success in formal development. [27] 26
12. Bieber, P.: Formal techniques for an ITSEC-E4 secure gateway. In: ACSAC, IEEE Computer Society (1996) 236–246
13. Jaeger, É., Dubois, C.: Why would you trust B? [28] 288–302
14. Berkani, K., Dubois, C., Faivre, A., Falampin, J.: Validation des règles de base de l'Atelier B. Technique et Science Informatiques **23**(7) (2004) 855–878
15. Aydemir, B., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The POPLmark challenge. In Hurd, J., Melham, T.F., eds.: International Conference on Theorem Proving in Higher Order Logics (TPHOLs). Volume 3603 of Lecture Notes in Computer Science., Springer (August 2005) 50–65
16. Aydemir, B., Charguéraud, A., Pierce, B.C., Weirich, S.: Engineering aspects of formal metatheory (April 2007) Manuscript.
17. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae (Proceedings) (1972) 381–392
18. Gordon, A.D.: A mechanisation of name-carrying syntax up to alpha-conversion. In Joyce, J.J., Seger, C.J.H., eds.: 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., London, UK, Springer-Verlag (1993) 413–425
19. Norrish, M., Vestergaard, R.: Proof pearl: de Bruijn terms really do work. In Schneider, K., Brandt, J., eds.: TPHOLs. Volume 4732 of Lecture Notes in Computer Science., Springer (2007) 207–222
20. Holmes, M.R., Alves-Foss, J.: The Watson theorem prover. J. Autom. Reasoning **26**(4) (2001) 357–408
21. Abadi, M., Cardelli, L., Curien, P.L., Lévy, J.J.: Explicit substitutions. Journal of Functional Programming **1**(4) (1991) 375–416
22. Curien, P.L., Hardin, T., Lévy, J.J.: Confluence properties of weak and strong calculi of explicit substitutions. Journal of the ACM **43**(2) (March 1996) 362–397
23. Nadathur, G., Wilson, D.S.: A notation for lambda terms: A generalization of environments. Theor. Comput. Sci. **198**(1-2) (1998) 49–98
24. Nadathur, G.: A fine-grained notation for lambda terms and its use in intensional operations. Journal of Functional and Logic Programming **1999**(2) (1999)
25. Dargaye, Z., Leroy, X.: Mechanized verification of CPS transformations. [28] 211–225
26. Jaeger, É., Hardin, T.: A few remarks about formal development of secure systems. In: HASE, IEEE Computer Society (2008) 165–174
27. Bert, D., ed.: B'98: Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, April 22-24, 1998, Proceedings. In Bert, D., ed.: B. Volume 1393 of Lecture Notes in Computer Science., Springer (1998)
28. Dershowitz, N., Voronkov, A., eds.: Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings. In Dershowitz, N., Voronkov, A., eds.: LPAR. Volume 4790 of Lecture Notes in Computer Science., Springer (2007)