Why Would You Trust B?

Eric Jaeger, Catherine Dubois

HAL Id: hal-00363345
https://hal.archives-ouvertes.fr/hal-00363345

Submitted on 23 Feb 2009

Why Would You Trust B?

Éric Jaeger[12] and Catherine Dubois[3]

[1] LIP6, Université Paris 6, 4 place Jussieu, 75252 Paris Cedex 05, France
[2] LTI, Direction centrale de la sécurité des systèmes d'information, 51 boulevard de La Tour-Maubourg, 75700 Paris 07 SP, France
[3] CEDRIC, École nationale supérieure d'informatique pour l'industrie et l'entreprise, 18 allée Jean Rostand, 91025 Evry Cedex, France

Abstract. The use of formal methods provides confidence in the correctness of developments. Yet one may argue about the actual level of confidence obtained when the method itself – or its implementation – is not formally checked. We address this question for the B, a widely used formal method that allows for the derivation of correct programs from specifications. Through a deep embedding of the B logic in *Coq*, we check the B theory but also implement B tools. Both aspects are illustrated by the description of a proved prover for the B logic.

Key words: Confidence, Formal Methods, Prover, Deep embedding

A clear benefit of formal methods is to increase the confidence in the correctness of developments. However, one may argue about the actual level of confidence obtained, when the method or its implementation are not themselves formally checked. This question is legitimate for safety, as one may accidentally derive invalid results. It is even more relevant when security is a concern, as any flaw can be deliberately exploited by a malicious developer to obfuscate undesirable behaviours of a system while still getting a certification.

B [1] 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 in the industry for projects where safety or security is mandatory. So the B is 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). Those are the objectives of *BiCoq*, a deep embedding of the B logic in *Coq* [2].

BiCoq benefits from the support of *Coq* to study the theory of B, and to check the validity of standard definitions and results. *BiCoq* also allows us, through an implementation strategy, to develop formally checked B tools. This strategy is illustrated in this paper by the development of a prover engine for the B logic, that can be extracted and used independently of *Coq*. *Coq* is therefore our notary public, witnessing the validity of the results associated to the B theory, as well as the correctness of tools implementing those results – ultimately increasing confidence in B developments. The approach, combining a deep embedding and

an implementation technique, can be extended to address further elements of the B, beyond its logic, or to safely enrich it, as illustrated in this paper.

This paper is divided into 9 sections. Sections 1, 2 and 3 briefly introduce B, *Coq* and the notion of embedding. The B logic and its formalisation in *Coq* are presented in Sec. 4. Section 5 describes various results proved using *BiCoq*. Section 6 focuses on the implementation strategy, and presents its application to the development of a set of extractible proof tactics for a B prover. Section 7 discusses further uses of *BiCoq*, and mentions some existing extensions. Finally, Sect. 8 concludes and identifies further activities.

1   A Short Introduction to B

In a nutshell, 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. An abstract B *machine* is a module combining a state, properties and operations (described as substitutions) to read or alter the state.

The logic is used to express preconditions, invariants, etc. and to conduct proofs. The *GSL* allows for definitions of substitutions that can be abstract, declarative and non-deterministic (that is, specifications) as well as concrete, imperative and deterministic (that is, programs). The following example uses the non-deterministic substitution **ANY** (a "magic" operator finding a value which satisfies a property) to specify the square root of a natural number n:

Example 1. **ANY** x **WHERE** $x*x \leq n < (x+1)*(x+1)$ **THEN** $\sqrt{}(n) := x$ **END**

Regarding the methodology, a machine M_C *refines* an abstract machine M_A if one cannot distinguish M_C from M_A by valid operation calls – this notion being independent of the internal representations, as illustrated by the following example of a system returning the maximum of a set of stored values:

Example 2. The state of M_A is a (non implementable) set of natural numbers; the state of M_C is a natural number. Yet M_C, having the expected behaviour, refines M_A.

MACHINE M_A	**REFINEMENT** M_C
VARIABLES S	**VARIABLES** s
INVARIANT $S \subseteq \mathbb{N}$	**INVARIANT** $s = max(S \cup \{0\})$
INITIALISATION $S := \emptyset$	**INITIALISATION** $s := 0$
OPERATIONS	**OPERATIONS**
$store(n) \triangleq$	$store(n) \triangleq$
PRE $n \in \mathbb{N}$ **THEN** $S := S \cup \{n\}$ **END**	**IF** $s < n$ **THEN** $s := n$ **END**
$m \leftarrow get \triangleq$	$m \leftarrow get \triangleq$
PRE $S \neq \emptyset$ **THEN** $m := max(S)$ **END**	**BEGIN** $m := s$ **END**
END	**END**

Refinement being transitive, it is possible to go progressively from the specification to the implementation. By discharging at each step the *proof obligations* defined by the B methodology, a program can be proved to be a correct and complete implementation of a specification. This methodology, combined with

the numerous native notions provided by the set theory and the existence of toolkits, make the B a popular formal method, widely used in the industry.

Note that the B logic is not genuinely typed and allows for manipulation of free variables. A special mechanism, called *type-checking* (but thereafter referred to as *wf-checking*), filters ill-formed (potentially paradoxal) terms; it is only mentioned in this paper, deserving a dedicated analysis.

The rest of the paper only deals with the B logic (its inference rules).

2 A Short Introduction to *Coq*

Coq 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 an *ML*-like language with pattern-matching. It is possible in *Coq* to define values and types, including dependent types (that is, 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 (that 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. For example, let's consider the standard representation of natural numbers:

Example 3. **Inductive** $\mathbb{N}:\mathbf{Set}:=0:\mathbb{N}\,|\,S:\mathbb{N}\rightarrow\mathbb{N}$

It 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 $S^n(0)$ for any finite n; being well-founded, structural induction on $\mathbb{N}$ is possible.

Coq also allows for the declaration of inductive logical properties, e.g.:

Example 4. **Inductive** $ev:\mathbb{N}\rightarrow\mathbf{Prop}:=ev_0:ev\,0\,|\,ev_2:\forall(n:\mathbb{N}),\,ev\,n\rightarrow ev\,(S(S\,n))$

It defines a family of *logical types*: $ev\,0$ is a type inhabited by the term (ev_0), $ev\,2$ is another type inhabited by $(ev_2\,0\,ev_0)$, and $ev\,1$ is an empty type. The standard interpretation is that ev_0 is a proof of the proposition $ev\,0$ and that there is no proof of $ev\,1$, that is we have $\neg(ev\,1)$.

An intuitive interpretation of our two examples is that $\mathbb{N}$ is a set of terms, and ev a predicate marking some of them, defining a subset of $\mathbb{N}$.

3 Deep Embedding and Related Works

Embedding in a proof assistant consists in mechanizing a *guest* logic by encoding its syntax and semantic into a *host* logic ([3,4,5]). In a *shallow* embedding, the encoding is partially based on a direct translation of the guest logic into constructs of the host logic. In a *deep* embedding the syntax and the semantic are formalised as datatypes. At a fundamental level, taking the view presented in Sec. 2, the deep embedding of a logic is simply a definition of the set of

all sequents (the terms) and a predicate marking those that are *provable* (the inference rules of the guest logic being encoded as constructors of this predicate).

Shallow embeddings of B in higher-order logics have been proposed in several papers (cf. [6,7]) 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*. If the objective is to have an accurate formalisation of the guest system, the definition of a valid interpretation is difficult – e.g. B functions are relations, possibly partial or undecidable, and translating accurately this concept in *Coq* is a tricky exercise.

BiCoq aims at such an accurate formalisation, to pinpoint any problem of the theory with the objective to increase confidence in the developments when safety or security is a concern; in addition, we also have an implementation objective. In such cases, a deep embedding is fully justified – see for example the development of a sound and complete theorem prover for first-order logic verified in *Isabelle* proposed in [8].

A deep embedding of the B logic in *Coq* is described in [9] (using notations with names), to validate the *base rules* used by the prover of *Atelier-B* – yet not checking standard B results, and without implementation goal. As far as the implementation of a trusted B prover is concerned, we can also mention the encoding of the B logic as a rewriting system proposed in [10].

Deep embeddings have also the advantage to clearly separate the host and the guest logics: in *Bicoq*, excluded middle, provable in B, is not promoted to *Coq*. This improves readability, and allows one to study meta-theoretical questions such as consistency. Furthermore, the host logic consistency is not endangered.

4   Formalising the B Logic in *Coq*

In this section, we present our embedding of the B logic in the *Coq* system; the embedding uses a *De Bruijn* representation that avoids ambiguities and constitutes an efficient solution w.r.t. the implementation objective (see [11,12]). Deviations between B and its formalisation are described and justified.

Notation. B *definitions use upper case letters with standard notations.* BiCoq *uses lower case letters, and mixes* B *and* Coq *notations; standard notations are used for* Coq *(e.g. $\forall$ is the universal quantification) while dotted notations are used for the embedded* B *(e.g. $\dot{\forall}$ is the universal quantification constructor).*

Notation. $[T]$ *denotes the type of the lists whose elements have type T.*

4.1   Syntax

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

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

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. The comprehension set operator, while syntactically defined by $\{V|P\}$, is rejected at *wf-checking* if not of the form $\{V|V \in S \wedge P\}$, with V a variable not free in S

Definition. *Other connectors are defined from the previous ones, $P{\Leftrightarrow}Q$ is defined as $P{\Rightarrow}Q \wedge Q{\Rightarrow}P$, $P \vee Q$ as $\neg P{\Rightarrow}Q$, and $\exists\, V \cdot P$ as $\neg \forall\, V \cdot \neg P$.*

The first design choice of *BiCoq* is to use a pure nameless *De Bruijn* notation (see [11,13]), where variables are represented by indexes giving the position of their binder – here the universal quantifier and the comprehension set. When an index exceeds the number of parent binders, it is said to be *dangling* and represents a *free variable*, whose name is provided by a scope (left implicit in this paper), so that any syntactically correct term is semantically valid, and there is no need for well-formedness condition[4]. In this representation, proofs of side conditions related to name clashing are replaced by computations on indexes, but the index representing a variable is not constant in a term.

The B syntax is formalised in *Coq* by two mutually inductive types with the following constructors, $\mathbb{I}$ being the set of indexes (that is, $\mathbb{N}\backslash\{0\}$) and $\mathbb{J}$ an infinite set of names with a decidable equality:

$$\mathbb{P} \quad := \quad \mathbb{P}\dot{\wedge}\mathbb{P} \quad | \; \mathbb{P}\dot{\Rightarrow}\mathbb{P} \; \mid \; \dot{\neg}\mathbb{P} \quad | \; \dot{\forall}\,\mathbb{P} \quad | \; \mathbb{E}\dot{=}\mathbb{E} \; \mid \; \mathbb{E}\dot{\in}\mathbb{E}$$
$$\mathbb{E} \quad := \quad \dot{\chi}\mathbb{I} \qquad | \; \mathbb{E}\dot{\mapsto}\mathbb{E} \; \mid \; \dot{\downarrow}\mathbb{E} \quad | \; \dot{\Omega} \qquad | \; \dot{\uparrow}\mathbb{E} \qquad | \; \mathbb{E}\dot{\times}\mathbb{E} \; \mid \; \{\mathbb{E}\dot{|}\mathbb{P}\} \; \mid \; \dot{\omega}\mathbb{J}$$

$\mathbb{P}$ represents B predicates, while $\mathbb{E}$ merges B expressions, sets and variables.

Using a *De Bruijn* representation, binders $\dot{\forall}$ and $\{\dot{|}\}$ have no attached names and only bind (implicitly) a single variable. Binding over list of variables can be eliminated without loss of expressivity, as illustrated by the following example:

Example 5. $\{V \,|\, V \in S_1 {\times} S_2 \wedge \exists V_1 \cdot (V_1 \in S_1 \wedge \exists V_2 \cdot (V_2 \in S_2 \wedge V_1 \mapsto V_2 = V \wedge P))\}$ represents $\{V_1, V_2 \,|\, V_1, V_2 \in S_1 {\times} S_2 \wedge P\}$ [5]

The constructor $\{\dot{|}\}$ is further modified to be parameterised by an expression, to keep in the syntax definition only wf-checkable terms. Indeed, only comprehension sets of the form $\{V \mid V \in E \wedge P\}$, with V not free in E, are valid. The *BiCoq* representation of this set is $\{e\dot{|}p\}$; to reflect the non-freeness condition, $\{e\dot{|}p\}$ only binds variables in its predicate parameter p. By these design choices, we bridge the gap between syntactically correct terms and wf-checkable ones, while being conservative.

$\dot{\Omega}$ represents the constant set **BIG**, $\dot{\chi}$ unary (*De Bruijn*) variables. The constructor $\dot{\omega}$ is without B equivalent, and provides elements of $\dot{\Omega}$ (cf. Par. 4.3).

Notation. *$\dot{\chi}_i$ denotes the application of constructor $\dot{\chi}$ to $i\,{:}\,\mathbb{I}$ and $\dot{\omega}_j$ of constructor $\dot{\omega}$ to $j\,{:}\,\mathbb{J}$. By abuse of notation the variable $\dot{\chi}_i$ is also denoted simply by i.*

---

[4] An alternative approach to avoid well-formedness conditions is described in [14].

[5] This second representation, while standard in B, appears to be an illegal binding over the expression $x \mapsto y$ rather than over the variable x, y, but the same notations are used for both in [1] and such confusions are frequent.

Finally, the elementary substitution is not considered in *BiCoq* as a syntactical construct but is replaced by functions on terms – substitution being introduced earlier in *B* only to be used in the description of inference rules. Note however that the full *GSL* of *B* can still be formalised by additional terms constructors (the *explicit substitution* approach, see [15,16]).

Notation. $p_1 \dot{\Leftrightarrow} p_2$ *is defined as* $p_1 \dot{\Rightarrow} p_2 \dot{\wedge} p_2 \dot{\Rightarrow} p_1$, $p_1 \dot{\vee} p_2$ *as* $\dot{\neg} p_1 \dot{\Rightarrow} p_2$, *and* $\dot{\exists} p$ *as* $\dot{\neg} \dot{\forall} \dot{\neg} p$.

Notation. $\mathbb{T}$ *denotes the type of terms, that is the union of* $\mathbb{P}$ *and* $\mathbb{E}$.

4.2 Dealing with the *De Bruijn* Notation

De Bruijn notation is an elegant solution to avoid complex name management, and it has numerous merits. But it also has a big drawback, being an unusual representation for human readers:

Example 6. If $x \in y$ is the interpretation of the term $1 \dot{\in} 2$, the interpretation of the term $\dot{\forall}(1 \dot{\in} 2)$ is $\forall t \cdot t \in x$; because of the binder, the scope has shifted (so 2 now represents x), and (likely) the semantic has been distorted.

In this paragraph, we illustrate some of the consequences of using a *De Bruijn* notation, as well as how to mask such consequences from the users.

Induction When defining type $\mathbb{T}$, *Coq* automatically generates the associated structural induction principle. As illustrated in Ex. 6, it is however not semantically adequate, because it does not reflect *De Bruijn* indexes scoping. A more interesting principle is derived in *BiCoq* by using the syntactical depth function $\mathcal{D}$ of a term as a well-founded measure:

$$\forall (P : \mathbb{T} \to Prop), (\forall (t : \mathbb{T}), (\forall (t' : \mathbb{T}), \mathcal{D}(t') < \mathcal{D}(t) \to P \, t') \to P \, t) \to \forall (t : \mathbb{T}), \, P \, t$$

With this principle, for the term $\dot{\forall}(1 \dot{\in} 3)$ (that is, $\forall t \cdot t \in y$) we can choose to use an induction hypothesis on $1 \dot{\in} 2$ (that is, $x \in y$) instead of $1 \dot{\in} 3$ (that is, $x \in z$).
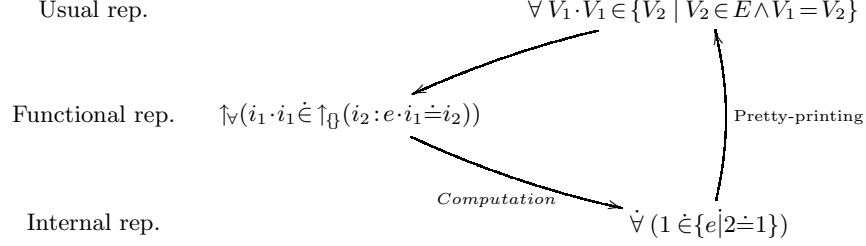
Non-Freeness The *B* notation $V \backslash T$ means that the variable V does not appear free in T. Non-freeness is defined in *BiCoq* as a type $\dot{\searrow} : \mathbb{I} \to \mathbb{T} \to Prop$ (a relation between $\mathbb{I}$, representing the variables, and $\mathbb{T}$), with the following rules[6]:

$$\frac{}{i \dot{\searrow} \dot{\Omega}} \qquad \frac{}{i \dot{\searrow} \dot{\omega}_k} \qquad \frac{i_1 \neq i_2}{i_1 \dot{\searrow} i_2} \qquad \frac{(i+1) \dot{\searrow} p}{i \dot{\searrow} \dot{\forall} p} \qquad \frac{i \dot{\searrow} e \quad (i+1) \dot{\searrow} p}{i \dot{\searrow} \{e | p\}}$$

The two first rules are axioms, the associated constructors are atomic and do not interact with variables. The rules for $\dot{\forall}$ and $\{ | \}$ reflect the fact that the associated constructors are binders and therefore shift the scope.

---

[6] The rules for the other constructors are trivial and can be obtained by straightforward extension, e.g. here $i \dot{\searrow} p$ and $i \dot{\searrow} q$ allow to derive $i \dot{\searrow} p \dot{\Rightarrow} q$.

Binding, Instantiation and Substitution It is possible to define functions to simulate B binding (that is the use of $\forall$ or $\{\}$, representing λ-abstraction). These functions constitute a built-in user interface to produce *De Bruijn* terms while using the usual representation, making *De Bruijn* indexes and their management invisible to the user (see also [17] for a similar approach):

Usual rep. $\qquad\qquad\qquad\qquad\qquad\qquad$ $\forall\, V_1 \cdot V_1 \in \{V_2 \mid V_2 \in E \wedge V_1 = V_2\}$

Functional rep. $\quad \uparrow_\forall(i_1 \cdot i_1 \dot\in \uparrow_{\{\}}(i_2 : e \cdot i_1 \dot= i_2))$ $\qquad\qquad\qquad$ Pretty-printing

Internal rep. $\qquad\qquad\qquad$ *Computation* $\qquad\qquad$ $\dot\forall\,(1 \dot\in \{e \dot| 2 \dot= 1\})$

The binding functions are defined by:

$$\uparrow_\forall(i \cdot p) := \dot\forall\, \mathbf{Bind}\, i\, 1\, p \qquad \uparrow_{\{\}}(i : e \cdot p) := \{e | \mathbf{Bind}\, i\, 1\, p\} \qquad \uparrow_\exists(i \cdot p) := \dot\exists \mathbf{Bind}\, i\, 1\, p$$

$\mathbf{Bind}(i_1\, i_2 : \mathbb{I})(t : \mathbb{T}) : \mathbb{T} := match\, t\, with$
$\quad | \quad \dot\Omega \quad | \quad \dot\omega_{j'} \Rightarrow t$
$\quad | \quad \dot\chi_{i'} \qquad\quad \Rightarrow t \text{ if } i' < i_2, \text{ or else } \dot\chi_{i_2} \text{ if } i' = i_1, \text{ or else } \dot\chi_{i'+1}$
$\quad | \quad \dot\forall\, p' \qquad\quad \Rightarrow \dot\forall\,(\mathbf{Bind}\,(i_1 + 1)\,(i_2 + 1)\,p')$
$\quad | \quad \{e' | p'\} \qquad \Rightarrow \{\mathbf{Bind}\, i_1\, i_2\, e' \mid \mathbf{Bind}\,(i_1 + 1)\,(i_2 + 1)\,p'\}$
$\quad | \quad \ldots \qquad\qquad \Rightarrow \ldots \text{ (straightforward extension)}$

On the same principles, the definition of instantiation functions (for elimination of $\forall$ or $\{\}$, representing β-reduction and denoted by $\downarrow_\forall(p \leftarrow e) : \mathbb{P} \to \mathbb{E} \to \mathbb{P}$ and $\downarrow_{\{\}}(e_1 \leftarrow e_2) : \mathbb{E} \to \mathbb{E} \to \mathbb{P})$ is straightforward – being partial, these functions just require in *Coq* an additional proof parameter (omitted in this paper) that the term is of the expected form. Finally, it is also possible to define a substitution function[7]:

$\langle i := e \rangle t : \mathbb{I} \to \mathbb{E} \to \mathbb{T} \to \mathbb{T} := match\, t\, with$
$\quad | \quad \dot\Omega \quad | \quad \dot\omega_{j'} \Rightarrow t$
$\quad | \quad \dot\chi_{i'} \qquad\quad \Rightarrow if\, i' = i\, then\, e\, else\, t$
$\quad | \quad \dot\forall\, p' \qquad\quad \Rightarrow \dot\forall\, \langle i+1 := \mathbf{Lift}(e) \rangle p'$
$\quad | \quad \{e' | p'\} \qquad \Rightarrow \{\langle i := e \rangle e' | \langle i+1 := \mathbf{Lift}(e) \rangle p'\}$
$\quad | \quad \ldots \qquad\qquad \Rightarrow \ldots \text{ (straightforward extension)}$

where $\mathbf{Lift}$, not detailed in this paper, increments dangling *De Bruijn* indexes. Remember that substitution is introduced early in B as a syntactical construct, but only to be used in inference rules. We consider that such rules are better represented using the resulting term (that is, the reduction of the application of the substitution).

Once these functions are defined, numerous lemmas are proved, such as the (in)famous ones describing all possible interactions between lifting, binding, instantiation and substitution. The following results are then derived, proving

---

[7] Substitution and instantiation may seem similar in usual notation, but their differences are emphasised when using *De Bruijn* notation.

the irrelevance of α-renaming or describing relationships between instantiation, binding and substitution (with $=$ the *Coq* term structural equality):

$$i_2 \dot{\diagdown} p \rightarrow \Uparrow_\forall (i_1 \cdot p) = \Uparrow_\forall (i_2 \cdot \langle i_1 := i_2 \rangle p) \qquad i_2 \dot{\diagdown} p \rightarrow \Uparrow_{\text{\O}} (i_1 : e \cdot p) = \Uparrow_{\text{\O}} (i_2 : e \cdot \langle i_1 := i_2 \rangle p)$$

$$\Downarrow_\forall (\Uparrow_\forall (i \cdot p) \leftarrow i) = p \qquad\qquad\qquad \Downarrow_{\text{\O}} (\Uparrow_{\text{\O}} (i : e \cdot p) \leftarrow i) = i \dot{\in} e \dot{\wedge} p$$

$$\Downarrow_\forall (\Uparrow_\forall (i \cdot p) \leftarrow e) = \langle i := e \rangle p$$

4.3 Inference Rules

Having formalised the B syntax and defined some functions and properties on terms, the next step is to encode the B inference rules. Thanks to the use of the functional representation described in the previous paragraph, *BiCoq* rules look very much like the standard B rules. The translation is therefore straightforward, merely a syntactical one, and the risk of error is very limited.

In our formalisation sets of hypothesis are represented by lists, with membership ($\in$) and inclusion ($\subseteq$) as well as the pointwise extension of non-freeness ($\dot{\diagdown}$). The B inference rules are formalised as constructors of an inductive type $\vdash\; : [\mathbb{P}] \rightarrow \mathbb{P} \rightarrow Prop$, that is $g \dot{\vdash} p$ is the *Coq* type of all B proofs of p under the assumptions g. Such a type may be inhabited (i.e. p is provable assuming g) or empty (i.e. there is no proof of p under the assumptions of g).

The B rules and their encoding as constructors are detailed in Tab. 1, universal quantifications being omitted (the types are $g, g_1, g_2 : [\mathbb{P}]$; $p, p_1, p_2 : \mathbb{P}$; $e, e_1, e_2, e_3, e_4 : \mathbb{E}$, $i, i_1, i_2 : \mathbb{I}$ and $j, j_1, j_2 : \mathbb{J}$). For most of them, translation is straightforward, only taking care to use functional substitution and binding where appropriate. On the other hand, the use of the functional representation imposes to keep the syntactical side conditions, except for the comprehension set rule, where such condition is embedded in the syntax; new rules have to be derived to benefit of the internal *De Bruijn* representation.

Only the last two B inference rules deserve discussion. The first one of these indicates that the constant set **BIG** is infinite, using the **infinite** B predicate defined by a fixpoint; unfolding this definition to produce a translation is possible, but not practical. Therefore, this rule is replaced in *BiCoq* by two different rules allowing to exhibit an infinity of elements of **BIG**, $\mathbb{J}$ being itself infinite.

The last rule, defining the semantics of pairs and products, is more interesting. A straightforward translation of this rule indeed leads to the impossibility to prove, in *BiCoq*, the following theorems from [1]:

$$\vdash (E \mapsto F) = (E' \mapsto F') \Rightarrow E = E' \wedge F = F'$$
$$\vdash S \in \uparrow U \wedge T \in \uparrow V \Rightarrow (S \times T) \in \uparrow (U \times V)$$

The proof of the first result provided in [1] is flawed, due to a confusion between pairs of expressions and lists of variables (as pointed out in [18]), both using the same notation – and cannot be corrected in the absence of a form of destructor for pairs. On the other hand, the proof of the monotonicity of cartesian product w.r.t. inclusion is not detailed in [1], being considered trivial. However, using the listed rules, one may derive predicates of the form $V \in S \times T$ but without being able to constraint V to be a pair to apply the last rule (a classical problem of the untyped λ-calculus). Basically, injectivity and surjectivity rules are lacking;

Table 1. Encoding of the B inference rules

B inference rules	$BiCoq$ formalisation
$\overline{P \vdash P}$	None, derived from $[\in]$
$\dfrac{P \text{ appears in } \Gamma}{\Gamma \vdash P}$	$p \in g \to g \dot\vdash p\ [\in]$
$\dfrac{\Gamma' \text{ includes } \Gamma \quad \Gamma \vdash P}{\Gamma' \vdash P}$	$g_1 \dot\vdash p \to g_1 \subseteq g_2 \to g_2 \dot\vdash p\ [\subseteq]$
$\dfrac{\Gamma \vdash P \quad \Gamma, P \vdash Q}{\Gamma \vdash Q}$	None, derived from $[\neg_n]\ [\neg_p]\ [\subseteq]\ [\in]$
$\dfrac{\Gamma \vdash P \Rightarrow Q}{\Gamma, P \vdash Q} \qquad \dfrac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}$	$g \dot\vdash p_1 \dot\Rightarrow p_2 \to g, p_1 \dot\vdash p_2$ $g, p_1 \dot\vdash p_2 \to g \dot\vdash p_1 \dot\Rightarrow p_2$
$\dfrac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}$	$g \dot\vdash p_1 \to g \dot\vdash p_2 \to g \dot\vdash p_1 \dot\wedge p_2\ [\wedge_i]$
$\dfrac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \qquad \dfrac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q}$	$g \dot\vdash p_1 \dot\wedge p_2 \to g \dot\vdash p_1$ $g \dot\vdash p_1 \dot\wedge p_2 \to g \dot\vdash p_2$
$\dfrac{\Gamma, Q \vdash P \quad \Gamma, Q \vdash \neg P}{\Gamma \vdash \neg Q}$	$g, p_2 \dot\vdash p_1 \to g, p_2 \dot\vdash \dot\neg p_1 \to g \dot\vdash \dot\neg p_2\ [\neg_p]$
$\dfrac{\Gamma, \neg Q \vdash P \quad \Gamma, \neg Q \vdash \neg P}{\Gamma \vdash Q}$	$g, \dot\neg p_2 \dot\vdash p_1 \to g, \dot\neg p_2 \dot\vdash \dot\neg p_1 \to g \dot\vdash p_2\ [\neg_n]$
$\overline{\Gamma \vdash E = E}$	$g \dot\vdash e \dot= e$
$\dfrac{\Gamma \vdash P \quad V \backslash \Gamma}{\Gamma \vdash \forall V \cdot P}$	$i \searrow g \to g \dot\vdash p \to g \dot\vdash \dot\uparrow_\forall (i \cdot p)\ [\forall_i]$
$\dfrac{\Gamma \vdash \forall V \cdot P}{\Gamma \vdash [V := E]P}$	$g \dot\vdash \dot\uparrow_\forall (i \cdot p) \to g \dot\vdash \langle i := e \rangle p$
$\dfrac{V \backslash S}{\vdash E \in \{V | V \in S \wedge P\} \Leftrightarrow E \in S \wedge [V := E]P}$	$\dot\vdash e_1 \dot\in \dot\uparrow_{\{\}} (i : e_2 \cdot p) \dot\Leftrightarrow e_1 \dot\in e_2 \dot\wedge \langle i := e_1 \rangle p$
$\dfrac{\Gamma \vdash E = F \quad \Gamma \vdash [V := E]P}{\Gamma \vdash [V := F]P}$	$g \dot\vdash e_1 \dot= e_2 \to g \dot\vdash \langle i := e_1 \rangle p \to g \dot\vdash \langle i := e_2 \rangle p$
$\dfrac{V \backslash S}{\vdash \exists V \cdot (V \in S) \Rightarrow \downarrow S \in S}$	$i \searrow e \to g \dot\vdash \dot\uparrow_\exists (i \cdot i \dot\in e) \dot\Rightarrow \dot\downarrow e \dot\in e$
$\dfrac{V \backslash S, T}{\vdash S \in \uparrow T \Leftrightarrow \forall V \cdot (V \in S \Rightarrow V \in T)}$	$i \searrow e_1 \to i \searrow e_2 \to g \dot\vdash e_1 \dot\in \dot\uparrow e_2 \dot\Leftrightarrow \dot\uparrow_\forall (i \cdot i \dot\in e_1 \dot\Rightarrow i \dot\in e_2)$
$\dfrac{V \backslash S, T}{\vdash \left(\begin{array}{c} \forall V \cdot (V \in S \Rightarrow V \in T) \\ \wedge \forall V \cdot (V \in T \Rightarrow V \in S) \end{array} \right) \Leftrightarrow S = T}$	$g \dot\vdash e_1 \dot\in \dot\uparrow e_2 \to g \dot\vdash e_2 \dot\in \dot\uparrow e_1 \to g \dot\vdash e_1 \dot= e_2$
$\overline{\vdash \mathbf{infinite}(BIG)}$	$g \dot\vdash \dot\omega_j \dot\in \dot\Omega$ $j_1 \neq j_2 \to g \dot\vdash \dot\neg(\dot\omega_{j_1} \dot= \dot\omega_{j_2})$
$\overline{\vdash (E \mapsto F) \in (S \times T) \Leftrightarrow (E \in S) \wedge (F \in T)}$	$g \dot\vdash e_1 \mapsto e_2 \dot= e_3 \mapsto e_4 \to g \dot\vdash e_1 \dot= e_3$ $g \dot\vdash e_1 \mapsto e_2 \dot= e_3 \mapsto e_4 \to g \dot\vdash e_2 \dot= e_4$ $i_1 \searrow e \dot\in (e_1 \dot\times e_2) \to i_2 \searrow e \dot\in (e_1 \dot\times e_2) \to i_1 \neq i_2 \to$ $g \dot\vdash \dot\uparrow_\exists (i_1 \cdot i_1 \dot\in e_1 \dot\wedge \dot\uparrow_\exists (i_2 \cdot i_2 \dot\in e_2 \dot\wedge e \dot= i_1 \mapsto i_2)) \dot\Leftrightarrow e \dot\in (e_1 \dot\times e_2)$

these observations, probably well known of the B gurus but not documented to our knowledge, have led us to replace this B rule by three new rules in order to be able to prove the expected theorems. Again, this process illustrates our conservative approach.

5 Proofs in *BiCoq*

5.1 Standard *B* Proofs

Using the definition of $\dot{\vdash}$, we formally prove in *BiCoq* all propositional calculus and predicate calculus results of [1], using the functional representation and following the proposed proof structure, e.g.:

$$i_1 \diagdown g \to i_1 \diagdown p \to g \dot{\vdash} \langle i_2 := i_1 \rangle p \to g \dot{\vdash} \dot{\uparrow}_\forall (i_2 \cdot p), \text{ that is } \frac{\Gamma \vdash [V_2 := V_1]P \quad V_1 \backslash \Gamma, P}{\Gamma \vdash \forall V_2 \cdot P}$$

To assist the proof construction *BiCoq* provides *Coq* tactics written in the *Coq* tactic language [19]. For example, the propositional calculus procedure described in [1], proposing a strategy based on propositional calculus theorems, is provided as a *Coq* tactic. More technical *Coq* tactics are also available in *BiCoq*, e.g. to obtain proved fresh variables.

An alternative form of theorems is also derived, using the internal *De Bruijn* representation; e.g. the $\dot{\forall}$-introduction rule (to be compared with $[\forall_i]$) is:

$$i \diagdown g \to i \diagdown \dot{\forall} p \to g \dot{\vdash} \textbf{Inst } i \, 1 \, p \to g \dot{\vdash} \dot{\forall} p$$

These last results are of course rather technical, not benefiting from the functional representation. Yet they have some interest, for technical lemmas or as derived rules in which only semantical side conditions remain (computations over *De Bruijn* indexes dealing with the syntactical ones).

5.2 Mixing *BiCoq* and *Coq* Logics

As it is standard in such a deep embedding (e.g. see [9]), *BiCoq* provides also results expressing relations between host and guest logics:

$$(g \dot{\vdash} p \vee g \dot{\vdash} q) \to g \dot{\vdash} p \dot{\vee} q \qquad\qquad g \dot{\vdash} p \dot{\Rightarrow} g \to (g \dot{\vdash} p \to g \dot{\vdash} p)$$
$$(g \dot{\vdash} p \wedge g \dot{\vdash} q) \leftrightarrow g \dot{\vdash} p \dot{\wedge} q \qquad\qquad (\forall (y : \mathbb{I}), \, g \dot{\vdash} \langle x := y \rangle p) \leftrightarrow g \dot{\vdash} \dot{\uparrow}_\forall (x \cdot p)$$

Asymmetrical results mark the differences between the classical B logic and the constructive *Coq* logic – e.g. a reciprocal of the first rule, combined with the excluded middle, would prove that for any predicate p either $\dot{\vdash} p$ or $\dot{\vdash} \dot{\neg} p$, which of course is not the case. This emphasises the fact that both logics are well separated, the B logic being embedded has an external theory.

By providing the best of both worlds, these results constitute efficient proof tactics. For example, the last theorem does not reflect non-freeness side conditions from B to the *Coq* logic (*Coq* taking care of such conditions automatically).

6 Developing a Proved *B* Toolkit

In this section, we detail how *BiCoq* is used as a framework for the development of formally checked *B* toolkits. *Coq* offers mechanisms to extract programs from constructive proofs (i.e. software from logical definitions and theorems), but a different approach is chosen here. Indeed, *BiCoq* includes code (in the form of functions using the *ML*-like internal language of *Coq*) which is proved correct. This code is extractible by a pure syntactical process, e.g. in *Objective Caml*, using the extraction mechanism of *Coq*. By doing so, we obtain proved *B* tools whose code is small, readable and efficient – and independent of *Coq*.

Notation. $\mathbb{B}$ *represents the booleans,* $\top$ *being true and* $\bot$ *being false.*

Notation. *Hat notations are used for boolean functions (e.g.* $\widehat{\wedge}$ *is the boolean and).*

6.1 Implementing Decidable Properties

For P and f respectively a predicate and a boolean function over a type S, we note $(P \rightsquigarrow f)$ when f decides P, i.e. when the following property is proved:

$$\forall (s\!:\!S), (f(s)\!=\!\top \rightarrow P(s)) \wedge (f(s)\!=\!\bot \rightarrow \neg P(s))$$

By defining folding as the extension of predicates and functions to lists, we prove that if f decides P, then the folding of f decides the folding of P:

$\mathbf{Fold_p}(P) := fun(L\!:\![S]) \Rightarrow \forall(s\!:\!S), s \in L \rightarrow P(s)$
$\mathbf{Fold_f}(f) := fun(L\!:\![S]) \Rightarrow if\ \mathbf{empty}(L)\ then\ \top\ else\ f(\mathbf{head}(L)) \widehat{\wedge} \mathbf{Fold_f}(f)(\mathbf{tail}(L))$
$(P \rightsquigarrow f) \rightarrow (\mathbf{Fold_p}(P) \rightsquigarrow \mathbf{Fold_f}(f))$

Example 7 (Non-freeness). Non-freeness is defined in *B* as a logical proposition and represented by the inductive type $\stackrel{.}{\backslash}$ in *BiCoq*. Our implementation strategy consists in developing a program $\widehat{\backslash} : \mathbb{I} \rightarrow \mathbb{T} \rightarrow \mathbb{B}$ and to prove that $(\stackrel{.}{\backslash} \rightsquigarrow \widehat{\backslash})$. Hence $\widehat{\backslash}$ and its extension (checking that a variable does not occur free in a list of hypotheses) are proved correct and can be extracted.

In *BiCoq* this approach is systematic; all typed equalities are implemented and proved correct (e.g. term equality), as well as non-freeness, list membership, inclusion, etc. to constitute our formally checked *B* toolkit.

6.2 A Proved Prover for the *B* Logic

In this paragraph we focus on the definition of an extractible prover to conduct first-order *B* proofs for standard *B* developments.

BiCoq includes programs, named *B tactics* in the following, to simulate the application of *B* inference rules or theorems. By providing such a dedicated piece of code for each of the inference rules listed in Tab. 1, and by proving them correct, we got a correct and complete prover (that is, any standard *B* result can be derived using this prover).

To this end, a type for *sequents* is defined as the product $[\mathbb{P}]\times\mathbb{P}$; for $g\!:\![\mathbb{P}]$ and $p\!:\!\mathbb{P}$ we denote $g\Vdash p$ the associated pair. While $g\vdash p$ is the type of B proofs of p under the assumptions g, that can be inhabited or not, $g\Vdash p$ is a syntactical construct extending $\mathbb{T}$. To interpret a sequent, we use the translation $\mathbf{Trans}_{\vdash}$ that for a pair $g\Vdash p$ returns the type $g\vdash p$ (and its extension derived by $\mathbf{Fold_p}$).

A B tactic is a function $T_B:\Vdash\,\to\,[\Vdash]$ that, provided a goal $g\Vdash p$, returns a list of subgoals $[g_1\Vdash p_1,\ldots,g_n\Vdash p_n]$ which together are sufficient to prove $g\Vdash p$; if a B tactic concludes (proves the goal) this list is empty. The following (elementary) examples give the definition of the B tactics associated respectively to the inference rules $[\in]$ and $[\wedge_i]$:

Example 8. $\mathbf{T}_{\in}(s):=let\ (g\Vdash p:=s)\ in\ if\ p\widehat{\in}g\ then\ []\ else\ [s]$

Example 9. $\mathbf{T}_{\wedge_{\mathbf{i}}}(s):=let\ (g\Vdash p:=s)\ in\ match\ p\ with\ p_1\wedge p_2\Rightarrow[g\Vdash p_1,g\Vdash p_2]\mid\_\Rightarrow[s]$

The implementation strategy described in Par. 6.1 is now particularly relevant, as $\mathbf{T}_{\in}$ uses the boolean function $\widehat{\in}$ instead of the logical proposition $\in$.

Following the same principles, numerous (much more complex) B tactics are provided in *BiCoq*, implementing theorems or strategies, such as the decision procedure for propositional calculus described in [1]. For each B tactic T_B, the correctness is ensured by a proof of the following property:

$$\forall\,(s:\Vdash),\mathbf{Trans}_{\vdash}(T_B(s))\to\mathbf{Trans}_{\vdash}(s),\text{ that is }\frac{g_1\vdash p_1\quad\ldots\quad g_n\vdash p_n}{g\vdash p}$$

Thanks to the functions defined in Par. 4.2, management of the *De Bruijn* indexes can be hidden from the users of the B tactics. With the programs already provided in *BiCoq* (such as non-freeness, binding, etc.), these B tactics constitute the core of a proved prover. This prover still lacks automation and *HMI*, and should be coupled with other tools, for example a B parser using the platform *BRILLANT* [20].

7 Higher-Order Considerations and Extensions

While the B logic is first-order, various definitions and proofs in [1] are conducted in a higher-order meta-logic: results in propositional calculus are proved by induction over terms, and refinement is defined by quantification over predicates before being transformed into an equivalent first-order definition. Using the higher-order framework provided by *Coq*, *BiCoq* can clearly be extended to integrate and to formally check such concepts.

New results can also be derived; for example, using the proof depth function $\mathcal{D}_{\vdash}:\dot{\vdash}\to\mathbb{I}$, we obtain a depth induction principle on B proof trees e.g. for results about proof rewriting. Other results, proved in higher-order logic, are applicable in first-order B logic, and implemented as B tactics for standard B proofs. This is the case for the following congruence results.

Predicate Substitution. We extend the B logic syntax with a new *predicate variable* constructor $\dot{\pi}\mathbb{K} : \mathbb{P}$ ($\mathbb{K}$ being an infinite set of names with a decidable equality), without adding any inference rules in order not to enrich the *BiCoq* logic[8]. Only limited modifications of *BiCoq* are required to deal with this new constructor, e.g. non-freeness with the additional rule $\forall\,(i\,{:}\,\mathbb{I})(k\,{:}\,\mathbb{K}),\ i\diagdown\dot{\pi}_k$.

Predicate variables play a role similar to the one of the variables – they are placeholders that can be replaced by a predicate using the substitution function $\langle k := p_1\rangle p_2 : \mathbb{K} \to \mathbb{P} \to \mathbb{P} \to \mathbb{P}$, not detailed in this paper, that mimicks the expression substitution function (see Par. 4.2). Thanks to this extension, we can prove the following congruence rules for $\dot{\Leftrightarrow}$ and implement associated B tactics that can be used e.g. to unfold a definition in a term, even under binders:

$$g\dot\vdash p_1\dot\Leftrightarrow p_2 \to g\dot\vdash \langle j:\equiv p_1\rangle p\dot\Leftrightarrow\langle j:\equiv p_2\rangle p \qquad g\dot\vdash p_1\dot\Leftrightarrow p_2 \to g\dot\vdash \langle j:\equiv p_1\rangle e\dot=\langle j:\equiv p_2\rangle e$$

Example 10. $x\dot=0, y\dot\in\mathbb{N}\dot\vdash y\leq x\dot\Leftrightarrow y\dot=0$, therefore we immediately derive (in one step) $x\dot=0, y\dot\in\mathbb{N}\dot\vdash \dot\Uparrow_\forall(v\cdot v\dot\in\Uparrow_{\dot{0}}(t:\mathbb{N}\cdot t\leq y\dot\wedge y\leq x))\dot\Leftrightarrow\dot\Uparrow_\forall(v\cdot v\dot\in\Uparrow_{\dot{0}}(t:\mathbb{N}\cdot t\leq y\dot\wedge y\dot=0))$

Note that predicate substitution and expression substitution mechanically forbid the capture of variables in the substituted subterm, by lifting dangling *De Bruijn* indexes when crossing a binder. That is, in Ex. 10, if v or t appear free in the substituted subterm, they escape capture during substitution.

Predicate Grafting. Other congruence results can be derived for *grafting* of predicates, a modified substitution (not lifting the substituted subterm) allowing for the capture of variables:

$$\langle k\triangleleft p\rangle t:\mathbb{K}\to\mathbb{P}\to\mathbb{T}\to\mathbb{T}:=match\ t\ with$$

$$
\begin{array}{lll}
|\quad \dot\Omega \quad|\quad \dot\omega_{j'} \quad|\quad \dot\chi_{i'} & \Rightarrow t\\
|\quad \dot\pi_{k'} & \Rightarrow if\ k'=k\ then\ p\ else\ t\\
|\quad \dot\forall\, p' & \Rightarrow \dot\forall\,\langle k\triangleleft p\rangle p'\\
|\quad \{e'|p'\} & \Rightarrow \{\langle k\triangleleft p\rangle e'|\langle k\triangleleft p\rangle p'\}\\
|\quad \ldots & \Rightarrow \ldots\ (straightforward\ extension)
\end{array}
$$

The associated congruence results and proofs are technical, and not detailed in this paper. We just provide for illustration a simplified version of these results:

$$\dot\vdash p_1\dot\Leftrightarrow p_2 \to g\dot\vdash \langle j\triangleleft p_1\rangle p\dot\Leftrightarrow\langle j\triangleleft p_2\rangle p \qquad \dot\vdash p_1\dot\Leftrightarrow p_2 \to g\dot\vdash \langle j\triangleleft p_1\rangle e\dot=\langle j\triangleleft p_2\rangle e$$

Example 11. $g\dot\vdash \langle k\triangleleft\dot\neg\dot\neg p\rangle q\dot\Leftrightarrow\langle j\triangleleft p\rangle q$, that is the elimination of double negations in a subterm (even if dangling *De Bruijn* indexes of p are bound in q)

Remark. Results such as the ones in Exs. 10 or 11 are provable in B, on a case-by-case basis, with a first-order proof depending on the structure of the term in which substitution or grafting is done. It is therefore conceivable to develop a specific (and likely complex) B tactic automatically building for such goals a proof using the B inference rules. On the contrary, the proposed extensions provide a new approach through results derived from a higher-order proof; the associated B tactics are therefore simpler, and produce generic (and shorter) proofs by using not only the B inference rules but also induction on $\mathbb{T}$.

---

[8] However, some new (propositional) sequents became provable, such as $\dot\pi_k\dot\vdash\dot\pi_k$.

8   Conclusion

Through an accurate deep embedding of the B logic in *Coq*, we identify shortfalls or confusions in [1] and propose amendments in order to be able to validate standard results – improving the confidence in the method and in the developments conducted with it. We describe a strategy to further benefit from this deep embedding by implementing verified B tools, extractible to be used independently of *Coq*. The approach is illustrated by the development of B tactics that constitute a complete and correct prover – usable to conduct proofs (provided further automation), or to check proofs produced by other tools. The objective, again, is to have better confidence in the developments conducted in B.

We also explain how, benefiting from the higher-order features of *Coq*, new results for B can be derived, and present an extension to derive congruence theorems related to equivalence, implemented in our prover.

All the results presented in this paper are mechanically checked; *BiCoq* currently represents about 550 definitions (i.e. types, properties, functions), 750 theorems and proofs in *Coq* – and about 6 man.months of development. It has now to be extended with the following definitions and results:

- Generation by the prover of B proof terms checkable by *Coq*.
- Use of a locally nameless *De Bruijn* representation with named free variables to derive unified congruence results (merging substitution and grafting).
- Fixpoint constructs, with application to the definition of natural numbers in the B style; on the innovative side, we expect to derive inductive B tactics, not available in current B implementations.
- *GSL* definition – either through a shallow embedding (an approach similar to the one presented in [6], but in *BiCoq*) or through a deep embedding (with higher-order and first-order refinement definitions, and proof of equivalence).

We would like to emphasise the simplicity and the efficiency of the deep embedding approach, when having both validation and implementation objectives. In a relatively short amount of time, it was possible to describe the B logic, to check its standard results, and to implement a proved prover for this logic.

References

1. Abrial, J.R.: The B-Book - Assigning Programs to Meanings. Cambridge University Press (August 1996)
2. The Coq development team: The Coq proof assistant reference manual. LogiCal Project. (2004)
3. 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

4. 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

5. 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)

6. 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

7. Chartier, P.: Formalisation of B in Isabelle/HOL. In: B '98: Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method, London, UK, Springer-Verlag (1998) 66–82

8. Ridge, T., Margetson, J.: A mechanically verified, sound and complete theorem prover for first order logic. In Hurd, J., Melham, T.F., eds.: TPHOLs. Volume 3603 of Lecture Notes in Computer Science., Springer (2005) 294–309

9. 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

10. Cirstea, H., Kirchner, C.: Using rewriting and strategies for describing the B predicate prover. In Kirchner, C., Kirchner, H., eds.: CADE-15 : Workshop on Strategies in automated deduction. Volume 1421 of Lecture Notes in Computer Science., Lindau, Germany, Springer (1998) 25–36

11. 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

12. Liang, C., Nadathur, G.: Tradeoffs in the intensional representation of lambda terms. In Tison, S., ed.: RTA. Volume 2378 of Lecture Notes in Computer Science., Springer (2002) 192–206

13. Aydemir, B., Charguéraud, A., Pierce, B.C., Weirich, S.: Engineering aspects of formal metatheory (April 2007) Manuscript.

14. Bird, R., Paterson, R.: De Bruijn notation as a nested datatype. Journal of Functional Programming **9**(1) (1999) 77–91

15. Abadi, M., Cardelli, L., Curien, P.L., Lévy, J.J.: Explicit substitutions. Journal of Functional Programming **1**(4) (1991) 375–416

16. 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

17. 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

18. Mussat, L. Private Communication (2005)

19. Delahaye, D.: A tactic language for the system Coq. In Parigot, M., Voronkov, A., eds.: Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island. Volume 1955 of Lecture Notes in Computer Science., Springer-Verlag LNCS/LNAI (November 2000) 85–95

20. Colin, S., Petit, D., Rocheteau, J., Marcano, R., Mariano, G., Poirriez, V.: BRILLANT : An open source and XML-based platform for rigourous software development. In: SEFM (Software Engineering and Formal Methods), Koblenz, Germany, AGKI (Artificial Intelligence Research Koblenz), IEEE Computer Society Press (september 2005) selectivity : 40/120.