Safe external calls from formally verified functional code: exiting the monad, and a BDD case study

David Monniaux, Sylvain Boulmé

▶ **To cite this version:**

David Monniaux, Sylvain Boulmé. Safe external calls from formally verified functional code: exiting the monad, and a BDD case study. 36es Journées Francophones des Langages Applicatifs (JFLA 2025), Jan 2025, Roiffé, France. hal-04859400

HAL Id: hal-04859400

https://inria.hal.science/hal-04859400v1

Submitted on 30 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

Safe external calls from formally verified functional code: exiting the monad, and a BDD case study

David Monniaux[1] and Sylvain Boulmé[2]

[1]Univ. Grenoble Alpes, CNRS, Grenoble INP*, VERIMAG, 38000 Grenoble, France

It is commonplace, when developing formally verified programs in a proof assistant, to have these programs make external calls to unverified code, for instance for I/O or expensive computations that can be implemented more efficiently and more easily in a general-purpose programming language.

Is this practice safe? In this paper, we discuss issues arising from the assumption that all constructs are functional and deterministic, which may not be the case of external code. We argue that (a) much external functionality, including computational oracles and hash tables, can be wrapped in a "may return" monad (b) many desirable properties are simply ensured by the type safety of the external code. As working permanently in a nondeterministic monad is cumbersome, and absurd if a result is actually deterministic, we introduce a "deterministic exit" monadic operator.

We illustrate this with the example of a library for computing over BDDs (binary decision diagrams).

1 Introduction

Proof assistants[1] typically operate on the assumption that functions are functions in the mathematical sense, that is, that when they are applied to a value v, they always return a value (they are *total*) and this value is always the same for the same v (they are *deterministic*). This is at odds with the notion of function in most programming languages, which may fail to return (thus may be *partial*), or may return different values for the same input (thus may be *nondeterministic*).

Some proof assistants provide an extraction facility, which turns computations expressed in the language of the proof assistant into programs in a general-purpose language, perhaps discarding definitions and computations that pertain only to proofs.[2] Typically, the resulting code is linked with code handwritten in the target language, which takes care of issues such as command-line parsing, opening of files etc., then calls the formally verified part, and then handles printing of output.

In some cases, the formally verified program calls handwritten, unverified code, for instance for performing expensive computations (this code may use machine integers, imperative data

*Institute of Engineering Univ. Grenoble Alpes

[1]This is the case for instance of Coq, PVS, Lean, and Isabelle.

[2]For instance, Coq may extract to OCaml (the most common case), Haskell and Scheme [Let04; Let08]. Isabelle may extract to SML, OCaml, Haskell and Scala.

structures, calls to external solvers or high-performance computational kernels). Typically this goes through a mechanism that asserts that there exists a certain function ϕ left undefined,[3] and that this function is implemented as function f in the target language: the extraction process will replace calls to ϕ by calls to f. For instance, the CompCert[4] formally verified compiler [Ler09],developed in Coq, calls several oracles programmed in OCaml, most notably the iterated register coalescing algorithm for register allocation [RL10]. The Chamois branch of CompCert[5] [MB24] adds more oracles, notably the computation of schedules for instruction scheduling, with a choice between several algorithms and even external solvers.

There are at this point obvious possibilities for wrongdoing [MB22, §3.2]: it is for instance possible to specify a function to return a list of length at most 2 (a dependent pair of a list l and a proof that $|l| \leq 2$), and link to an external function sometimes returning a list of length 3. No warning will be given: during extraction, the type of lists of elements from A of length at most 2, after erasure of proof information, is extracted to the type of lists of A, which includes lists of length 3. For this reason, it is generally advised to restrict types of values returned from external calls to types that do not convey proof information.

External code may be impure and may thus return different values for the same input. This may happen, for instance, when calling a SAT-solver, which uses a pseudorandom number generator and may return different satisfying instances for the same problem. It may also happen by accident if a supposedly deterministic program depends on mutable data structures that carry information over between calls; or one may want to use stateful data structures such as hash tables or databases.

For oracles that solve algorithmic problems, a common solution (e.g. register allocation in CompCert) is to assume the external function to be pure even without a formal proof that it is. Should it produce different answers to the same input, it would not in practice cause a problem, for the program does not intentionally call the oracle twice with the same input and depend on the outputs to be identical (recall that the oracle solves expensive problems and calls are thus kept to the minimum).

A common idiom is *hash-consing*: all objects of certain datatypes, at least those still in use, are stored in a global hash table, which is searched before creating a new object from these datatypes, so that no two distinct but isomorphic objects exist in the system at the same time. This allows, among other properties, unit-time equality tests, and giving unique names to objects, which in turns allows easy *memoization* of functions from these datatypes. Hash-consing is, however, uneasily integrated into fully functional settings. [BJM14]

In this article, we explain how *may-return monad* [Bou21, §2.2.1] combined with polymorphic typing at the interface between the verified and unverified code [Bou21, §2.2.3] can be used to gracefully implement hash-consing and memoization. We introduce an operator of *deterministic exit* from the may-return monad. As an example, we describe our implementation of an emptiness checker for propositional formulas using binary decision diagrams.

2 Working with nondeterminism

2.1 Nondeterminism can be dangerous

Consider the following Coq program, which calls the `flipper` function twice and computes the exclusive-or of the returned values:

```
Axiom flipper : unit → bool
Definition should_be_false (_ : unit):= xorb (flipper tt) (flipper tt)
```

[3] ϕ is an *uninterpreted function* in the logical sense: we do not know anything about its result, except that the function has a value for every input, and is functional: if $x = y$ then $\phi(x) = \phi(y)$.

[4] `https://compcert.org/ https://github.com/AbsInt/CompCert`

[5] `https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert`

We prove that "`should_be_false tt`" always evaluates to false, because both calls to "`flipper tt`" are supposed to evaluate to the same Boolean. Yet, if we set, through extraction mechanisms, `flipper` to be "`let s = ref true in fun () -> s := not !s; !s`", which alternates between true and false at every call, the extracted code for `should_be_false` will evaluate to true.

It is even possible to provoke a segmentation fault![6]

```
Definition disj_type (b : bool) : Type :=
  if b then (bool * bool) else bool
Definition prepare (_ : unit) : disj_type (flipper tt) :=
  if flipper tt as x return disj_type x then (false, true) else true
Definition extract (x : bool) : disj_type x → bool :=
  if x return disj_type x → bool then snd else fun x ⇒ x
Definition extract_prepare (_ : unit) :=
  extract (flipper tt) (prepare tt)
```

We prove that "`extract_prepare tt = true`" always evaluates to true. Yet the resulting OCaml code crashes with a segmentation fault! The reason is that in "`prepare tt`", "`flipper tt`" evaluates to false, thus "`prepare tt`" evaluates to a single true Boolean, which is encoded as a single integer 3 in the OCaml runtime, whereas in the call to `extract`, "`flipper tt`" evaluates to true, thus `extract` expects a pointer to a block consisting in a pair of Booleans, and then tries loading a value from a block at address 3, thus the crash.

Another issue are operations that may seem deterministic in the computational model of the target language, but behave nondeterministically according to the specification language, because they distinguish between values that are semantically equal in the specification language. Such is the case of the pointer equality test (== in OCaml), which distinguishes between several semantically identical values. Consider

```
Axiom naive_phys_eq : ∀ {A}, A → A → bool
Definition test1 (_: unit) := let a := true::nil in naive_phys_eq a a
Definition test2 (_: unit) :=
  let a := true::nil in let b := true::nil in naive_phys_eq a b
```

"`test1 tt`" and "`test2 tt`" both evaluate to "`naive_phys_eq (true::nil) (true::nil)`" and thus are provably equal. Because "`test1 tt`" passes the same pointer within == arguments, it produces true; thus OCaml "`Printf.printf "%b␣%b\n" (test1 ()) (test2 ())`" should produce "true true". Yet, if one extracts `naive_phys_eq` to == and runs the resulting program through the OCaml interpreter, one gets "true false". The reason is that the second procedure constructs two semantically equivalent lists in two different memory locations. Note that the same applies to the bytecode compiler, upon which the interpreter is based. In contrast, with the optimizing native code compiler, one gets "true true". This illustrates that the exact behavior of OCaml code embedding == may depend on compilation choices.

To avoid this problem, Jourdan [Jou16, §9.1, p. 183], instead of naively introducing a pointer equality test, introduced a primitive that takes two inputs a and b, two computations, one "slow", one "fast" under the condition that a and b are equal, and a proof that the "slow" and the "fast" computations have equal results, and executes the "fast" one if a and b point to the same data. The result is always equal to that of the "slow" computation.

2.2 Exiting from the may-return monad

2.2.1 The may-return monad

Boulmé [Bou21, §2.2.1] suggests wrapping all impure calls in a *may-return monad*: instead of an external call to a function $A \to B$, use an external call to a function $A \to ??B$ where $??B$ is a type of "maybe returned" values of type B. Intuitively, $??B$ may be thought

[6] This is a variant of an example proposed by Matthieu Sozeau.

about as a set of elements of type B representing all possible results of a nondeterministic computation. This monad is equipped with the usual *bind* operation for composition: if one has $f : A \to ??B$ and $g : B \to ??C$, then one can compose both functions into a function of type $A \to ??C$. Formal reasoning on the monad computations is enabled through an axiomatized relation, called *may-return*, and written "`k ⇝ r`" where `k:??A` represents a nondeterministic computation and `r:A` a possible result of this computation. The axioms over this relation express *partial correctness properties* over nondeterministic computations, i.e. properties of the form "`k ⇝ r → P r`". It is no longer possible to reach inconsistent configurations or break type safety, as described in §2.1, because this relies on successive calls returning identical values to the same inputs. The may-return monad is extracted to trivial operations in the OCaml code, essentially sequential control flow.

```
Axiom t: Type → Type.                                Notation "??" := t
Axiom mayRet: ∀ {A:Type}, ?? A → A → Prop.           Infix "⇝" := mayRet
Axiom ret: ∀ {A}, A → ?? A
Axiom bind: ∀ {A B}, (?? A) → (A → ?? B) → ?? B
Axiom mayRet_ret: ∀ A (a b:A),  (ret a) ⇝ b → a=b
Axiom mayRet_bind: ∀ A B k1 k2 (b:B),
     (bind k1 k2) ⇝ b → ∃ a:A, k1 ⇝ a ∧ (k2 a) ⇝ b
Axiom mk_annot: ∀ {A} (k: ?? A), ?? { a: A | k ⇝ a }
```

Operator "`DO b ←  .;; `" is notation for `bind`. Operator `mk_annot` is a primitive operator (extracted to identity) for introducing may-return annotations within types.

This approach has a drawback: it forces the rest of the computation into the may-return monad, which is cumbersome and increases the proof effort. This seems absurd when the final result of a computation is deterministic: for instance, if we implement a decision procedure for checking whether a propositional formula can be satisfied, even though this procedure may locally use nondeterministic features, its final result is provably unique.

2.2.2 Improvement: the deterministic exit

We introduce a "deterministic exit" operator[7], which allows retrieving the result of a nondeterministic computation (e.g. Coq code with OCaml oracles) if it is provably equal (in Coq) to that of a deterministic computation:

```
Axiom det_coerce: ∀ {A} (k: ?? A) (v: unit → A)
  (DET: ∀ r, k ⇝ r → r = v tt), A
Axiom det_coerce_correct: ∀ A (k: ?? A) v
  (DET: ∀ r, k ⇝ r → r = v tt), (det_coerce k v DET)=(v tt)
```

Operator `det_coerce` is extracted to return the result of `k` (if any). Indeed, the `DET` precondition of `det_coerce` forbids `k` to falsify at runtime the determinism of `det_coerce`, provable in Coq from its type.[8] Computation `v` is there to ensure consistency; it is not evaluated in the extracted code.[9] It also enables running the computation—without extraction—within Coq (even if this might be terribly inefficient in practice).

In the may-return monad, the pointer equality test [Bou21, §2.2.4] is expressed as

```
Axiom phys_eq: ∀ {A}, A → A → ?? bool
Axiom phys_eq_true : ∀ {A} (x y: A), phys_eq x y ⇝ true → x=y
```

Jourdan's primitive for fast-path computations if pointer equality is satisfied can be implemented with `phys_eq` and `det_coerce` (see Figure 1). As we shall see, `phys_eq`

[7] Gourdin's Phd[Gou23, §2.4.4] introduced another exit called "`has_returned`", discussed in Section 4.

[8] Actually, `k` could still nondeterministically fail (or diverge). Coq theorems must indeed be interpreted in partial correctness, even for pure Coq without external code [Bou21, §1.1.1].

[9] It ensures a trivial implementation "`det_coerce k v DET := v tt`" guaranteeing that these axioms are compatible with anything that base Coq is compatible with.

```
Program Definition jourdan_physEq {A B} (x y: A)
  (fast_path: x=y → B) (slow_path: unit → B)
  (Hdet: ∀ (Heq: x=y), fast_path Heq = slow_path())
  : {r: B | r = slow_path()}
:=  det_coerce (DO b ← mk_annot (phys_eq x y);;
      match b with | true ⇒ RET (fast_path _)
                   | false ⇒ RET (slow_path ()) end)  slow_path _
```

Figure 1. Implementing Jourdan's physical equality within a may-return monad

and `det_coerce` offer more flexibility: Jourdan's approach needs local convergence of the slow and fast computation; in contrast our approach allows running a whole long "nondeterministic" computation in the monad and retrieve the final deterministic result.

3 Case study: a BDD library

In order to demonstrate the practicability of our approach, we implement in Coq and OCaml computations involving very "imperative" traits (weak hash tables, hash tables, pointer equality, generation of unique identifiers) over a widely used data structure, prove them correct in Coq, and show how deterministic results can be safely recovered. There have of course been previous formalizations of BDDs in Coq [VGL00; BJM14]. The main difficulty is to maintain in an efficient way some global and local tables; one has to choose whether to reimplement hash tables in Coq, at the expense of efficiency and good interplay with the garbage collector, or to use OCaml's native hash tables. We demonstrate in this section how to do the latter safely and efficiently.

3.1 A short introduction to BDDs

Binary decision diagrams BDDs [Knu11, §7.1.4] express functions from a set of Boolean variables to the Booleans as successive case distinctions according to the variables; a node (v, b_0, b_1) in the diagram expresses "*if variable v is* true*, descend into branch b_1, else into b_0*", and the leaves of the diagram are 0 or 1, standing for return values false and true. Any Boolean function depending only on a finite set of variables can be expressed as a BDD over these variables. Ordered BDDs enforce that the variables on each path from the root to the leaves follow a certain ordering $\prec$ (not all variables need be present on each path). Reduced ordered BDDs (ROBDDs) impose that there are no isomorphic and distinct sub-diagrams (that is, these should be merged) and that there are no useless nodes (v, b, b), distinguishing between identical branches. Given an ordering on variables, a Boolean function depending on a finite number of variables is defined by a unique ROBDD (*canonicity*).

Testing if two ROBDDs define the same Boolean function reduces to testing if they are the same, through pointer equality: two ROBDDs define the same function if and only if they are isomorphic, and two ROBDDs are isomorphic if and only if they are the same in memory. ROBDDs are directly built reduced by building every node (v, b_0, b_1) using a "smart constructor" that checks if b_0 and b_1 point to the same diagram (returning b_0 in this case), and then if a node (v, b_0, b_1) is already present in the system (returning it if present), and finally constructs the node. All constructed nodes are kept in a global hash table (thus the phrase *hash-consing*), often called *unique table*. If implemented naively in a language with garbage collection, the unique table may prevent nodes from being collected, even if no references to them exist from outside the table. A *weak hash table* is used instead.[10]

It is straightforward to implement a Boolean and operation over two ROBDDs: assume $a = (v_a, a_0, a_1)$ and $b = (v_b, b_0, b_1)$, $a \wedge b$ is $(v_a, a_0 \wedge b, a_1 \wedge b)$ if $v_a \prec v_b$, $(v_b, a \wedge b_0, a \wedge b_1)$

[10] E.g., Weak in OCaml or WeakHashMap in Java.

if $v_b \prec v_a$, and $(v_a, a_0 \wedge b_0, a_1 \wedge b_1)$ if $v_a = v_b$, where $a_0 \wedge b$ etc. are the recursive calls. However, such a naive implementation would take time proportional to the products of the sizes of the unfoldings of the two BDDs (meaning that the subdiagrams are "unshared"), thus exponential time in the worst case. In order to recover time proportional to the products of the sizes of the diagrams, the usual approach is to build a hash table that *memoizes* the result for each pair (a, b) of subdiagrams encountered in the current computation, and queries the table before recursing into a branch. For extra efficiency, if a symmetric operation is memoized, the pair of arguments a, b is sorted before being stored or retrieved from hash tables, according to unique identifiers (UIDs): either the pointer to the node itself if it is stable, or UIDs within the nodes themselves.

For our case study, we implemented a variant of the BDD scheme known as *typed BDDs* (or *signed BDDs*) [MB88][Mad90, §3.1 pp.45–46]: the only leaf is 0, and each inner node carries, in addition to v, b_0, b_1, a Boolean saying whether the truth value of the b_1 branch is to be inverted. Furthermore, an extra Boolean at the root indicates whether to invert the truth value of the whole BDD. Typed BDDs thus use a single data structure to store a Boolean function and its negation, and negation is (almost) free. Typed BDDs are also canonical, and algorithms are similar to those for untyped BDDs. We implemented the "and", "exclusive-or" and "restrict" (substitution of a variable by a constant) operations, each needing memoization.[11] Proofs and explanations about typed BDDs are complicated by the extra Booleans; thus in this article we will keep explanations to vanilla BDDs.

3.2 Hash-consing and memoization

Hash-consing in Coq has long been recognized to be a thorny topic [BJM14]. One option is to implement hash tables in Coq using pure, functional data structures, in a "pure deep" approach [BJM14, §5.1][CBN24]. This approach may be improved by implementing hash tables using native arrays and integers [May23][12]. For efficiency, we however would prefer using the native (possibly weak) hash tables of OCaml. May-return monads enabled such a native hash-consing implementation, thanks to cheap defensive tests involving `phys_eq`, in a symbolic execution case study [Bou21, §3.3 pp.60–65]. Our BDD case study extends this technique with memoizing fixpoints (not needed in our previous case study). Moreover, it demonstrates how to formally verify a complete satisfiability test based on hash-consed BDD **without** formally proving that their semantic equality reduces to their physical equality.

Because OCaml moves objects in memory, we cannot use pointers as stable identifiers for objects. In addition to v, b_0, b_1, each node contains a UID u, obtained from a global counter incremented at every object creation. A nondeterministic function `uid_next : unit → ??uid` returns the next available UID; no assumption is made that it never returns the same identifier twice.[13] $u(n)$ denotes the UID in node n.

A global weak hash table, from the OCaml standard library, contains all currently allocated nodes. The hash table is equipped with a store and a retrieve operation: a node (v, b_0, b_1) is associated to the key $(v, u(b_0), u(b_1))$ (note the shallow hashing: 3 integers are hashed without data structure traversal). The retrieve operation is nondeterministic; no assumption is made that the node retrieved matches the requested (v, b_0, b_1): two pointer comparisons and a variable comparison are used to check that the node is correct. Hence, we neither assume that an existing node will be found in the able, nor that a node returned by the table corresponds to what was sought.[14]

For memoization, we proceed similarly to Boulmé's memoized fixpoints [Bou21, §2.3.3]. We use normal hash tables; the table is created at the beginning of the operation (an

[11] All 16 different possible Boolean operations over two Booleans are obtained by negation, and or exclusive-or.
[12] `https://github.com/valoran-M/diqt`
[13] This happens if more than $2^{31}/2^{63}$ identifiers are allocated. This may lead to inefficiency, but not to incorrectness.
[14] This absence of assumption of correctness is desirable given how delicately weak hash tables interact with OCaml's garbage collector, including the concurrent implementation in version 5.

alternative would be to retain the same table across operations, with periodic flushing). If an operation, such as $\wedge$, takes two BDDs as inputs, then the table is indexed by pairs of UIDs; the elements stored in the table are quadruples (a, b, c, P) where a, b are the inputs to the operation, c is the output, and P is a proof that c is a valid result for the application of the operation to a and b. The retrieval operation is modeled as nondeterministic, and no assumption is made that it returns the last value stored that corresponds to the key. After a quadruple (a', b', c, P) is retrieved from the table for key $u(a), u(b)$, pointer equality is checked for $a = a'$ and $b = b'$, which ensures that c is truly a correct answer for c.

We reach here a crucial point. In the OCaml runtime, the hash table stores triples (a, b, c); the proof term P is elided. Were the hash table declared explicitly at the OCaml level to store such triples, a buggy hash table implementation could, for instance, mix up triples belonging to the "and" and "exclusive-or" tables, or otherwise forge triples. Instead, we use *parametricity by invariants* [Bou21, §2.2 pp.32–33]: the hash table must have a polymorphic type $table(\alpha)$, where it stores elements of universally quantified type α; and we argue (without a formal proof) that this unary parametricity (which results from OCaml type safety) extends to Coq types through extraction. For example, if quadruples (a, b, c, P) where P is a proof term of a property $\pi(a, b, c)$ are stored into a polymorphic hash table as triples (a, b, c), then all triples (a, b, c) retrieved from the table satisfy $\pi(a, b, c)$. The intuition is that the OCaml implementation sees elements of type α as opaque, so the only objects of type α that it can provide are those that were previously given to it.[15]

The functor of OCaml hashtables, instantiated on pairs of UIDs, exposes an interface:

```
type key = uid * uid;;     type !'a t;;   val create : int -> 'a t;;
val find_opt: 'a t->key->'a option;; val replace: 'a t->key->'a->unit
```

which is seen from Coq as

```
Axiom isbdd_xor_hashtable: Type → Type
Axiom isbdd_xor_create_hashtable: ∀ U, unit → ??isbdd_xor_hashtable U
Axiom isbdd_xor_get_hash : ∀ {U} (table: isbdd_xor_hashtable U)
        (ua : uid) (ub : uid), ??(option U)
Axiom isbdd_xor_store_hash0 : ∀ {U} (table: isbdd_xor_hashtable U)
        (ua : uid) (ub : uid) (u : U), ??unit
```

3.3 Specification and implementation

We define a specification K of the "smart constructor", and then show that its implementation (checking that if the two branches are identical using pointer comparison, retrieval from the hash table, checking that the node retrieved is correct…) matches the specification. This smart constructor builds reduced BDDs. However, because of the use of pointer comparison to check that two branches are equal, this relies on the equivalence between equality and pointer equality, hence on hash-consing systematically reusing existing nodes, and hence on the correctness of the weak hash table. Hence, we cannot formally prove that this property holds (without a powerful programming logic on OCaml programs within Coq).

Functions such as "and" or "exclusive-or" operating over pairs of BDDs and creating a new BDD are somewhat complicated: they work within a nondeterministic monad, their termination argument is not a simple structural induction over one of the arguments, and they must be proved correct with respect to their specification together with their definition: the function uses the correctness of the results of its own recursive calls and stores this correctness predicate (P, below) in the memoization table.

Akin to the "Braga method" [LWM21], we introduce a relational specification for "and", written $a \,\&\, b \models c$, stating that c is an acceptable output for input a, b (i.e., equivalent to their conjunction), and expressed as an inductive type (inference rules in Fig. 2).

[15] This is not true if the OCaml code uses the Obj module, which allows unsafe type casts. We assume the unverified OCaml code has been audited not to use this module or the Marshal module.

$$\frac{}{a \,\&\, a \models a}$$ "shortcut", implemented by pointer equality $$\frac{a = \neg b}{a \,\&\, b \models 0}$$ for typed BDD

$$\frac{b \,\&\, a \models c}{a \,\&\, b \models c}$$ for swapping arguments, including when retrieving results from the hash table

$$\frac{}{a \,\&\, 1 \models a} \qquad \frac{}{a \,\&\, 0 \models 0}$$

$$\frac{b = (v_b, b_0, b_1) \qquad v_a \prec v_b \qquad a_0 \,\&\, b \models c_0 \qquad a_1 \,\&\, b \models c_1 \qquad K(v_a, c_0, c_1, c)}{(v_a, a_0, a_1) \,\&\, b \models c}$$

$$\frac{a_0 \,\&\, b_0 \models c_0 \qquad a_1 \,\&\, b_1 \models c_1 \qquad K(v, c_0, c_1, c)}{(v, a_0, a_1) \,\&\, (v, b_0, b_1) \models c}$$

Figure 2. Definition of inductive relation $a \,\&\, b \models c$

We prove by induction over this specification its *functional correctness*: the value of the Boolean function defined by c is truly the "and" of the values of the functions defined by a and b, over any valuation of the variables. We then define the monadic implementation of the "BDD and" function as a function $\forall a\ b,\ ??\{c : \texttt{bdd} \mid a \,\&\, b \models c\}$, thus together with its correctness proof, with a termination argument allowing descent into a, b, or both.[16]

3.4 Convergence to a deterministic result

We define an abstract syntax for logical formulas, and (by induction on these formulas) conversions to BDD and disjunctive normal form (DNF). The DNF is obtained by a fully deterministic computation, with no use of hash tables or other external structures. We define a nonemptiness test for DNFs (resp. BDDs) and prove its correctness: it answers `true` if and only if there is a valuation that satisfies the DNF (resp. BDDs). We thus define two satisfiability tests for formulas by composition of the conversion to BDD and DNF and these nonemptiness tests. From the correctness of the nonemptiness tests and the correctness of the conversions, we deduce that both satisfiability tests for formulas (the one using DNFs and the one using BDDs) return the same value. Thus, the apparently nondeterministic result of the satisfiability test for formulas going through BDDs is actually deterministic. We can thus exit the nondeterministic monad and obtain a test through BDDs that has the same return value as the test through DNFs, but is often much more efficient.

```
Definition test_nonemptiness (f : formula) : bool :=
 det_coerce (test_nonemptiness_bdd f) (fun _ ⇒ test_nonemptiness_dnf f)
            (test_nonemptiness_bdd_dnf_eqv f)
Theorem test_nonemptiness_correct :
  ∀ f, test_nonemptiness f = true ↔
    ∃ s, val_valid var_cmp s ∧ formula_sem s f = Some true
```

A few additional notes about the nonemptiness test. It is possible to define such a test by exploration of the branches of the BDDs. Such an algorithm would seem to be exponential, but actually has linear time in the number of variables if the BDD is reduced (which is the case here, even though we cannot formally prove it): subtrees that have no satisfying

[16] Since we work in a may-return monad, with no assumption of termination of computations, we could have dispensed with proving termination and instead use generalized fixpoints [Bou21, §2.3.3].

assignments will be just the leaf 0 and thus incur no exploration. In essence, this algorithm constructs a satisfying assignment.

A more optimized version of the emptiness test just checks at the root of the BDD or DNF. A reduced DNF (no conjunction $a \wedge \bar{a}$ containing opposite literals; we produce only reduced DNFs) has a satisfying assignment if and only if the list of disjuncts is nonempty. A reduced BDD has a satisfying assignment if and only if it is not the 0 leaf; however, even though we produce reduced BDDs, we cannot formally prove it. Instead, we can use a *shallow reduction* property: the smart constructor ensures that it never builds a node $(v, 0, 0)$. We say that a BDD is shallowly reduced if it does not contain nodes $(v, 0, 0)$. We show that a shallowly reduced BDD not reduced to a leaf 0 has a satisfying assignment.

4 Conclusion and related works

Our approach for safely using hash tables for memoization crucially relies on parametric polymorphism and type safety in the target programming language: we rely on them to show that the unverified program cannot return "logically ill-typed" values. The same is true of approaches where an unverified procedure builds a final result (say, a clause in a SAT-solver) that must be built using a family of operators supplied by the Coq code from a set S of initial components (clauses from the SAT problem, inequalities from a polyhedron): the α parameter, in essence, stands for S and ensures that the unverified procedure cannot mix values obtained from two different initial sets S and S', which would be allowed using a nonpolymorphic opaque type [Bou21, §4.2.1]. We thus advocate that polymorphic type safety should be preserved at the boundary of extracted code and external code.

The Coq to OCaml extractor attempts preserving types, including polymorphic types: for instance, the OCaml types extracted from `bool`, `list` etc. are what one would expect.[17] OCaml's type system is however too weak to represent Coq's type, even after projecting out the propositional part: System-F polymorphism cannot be represented, and neither can dependent types. This is why the extractor uses the Obj module to perform operations that are type-unsafe with respect to OCaml, but are actually safe because of the type-safety of the Coq code.[18] A recent proposal [FST24] is to drop this attempt at producing mostly well typed OCaml and to instead produce Malfunction, which is essentially untyped OCaml. We argue that this should maintain the possibility to automatically check that foreign OCaml functions satisfy their typesafe interfaces declared in Coq.

We have here introduced an exit of the may-return monad, called `det_coerce`. Gourdin's Phd [Gou23, §2.4.4] introduced another exit, without the determinism precondition, but only producing a Boolean information that the computation has returned:

```
Axiom has_returned: ∀ {A}, ?? A → bool
Axiom has_ret_correct: ∀ A k, has_returned k=true → ∃ r, k ⇝ r
```

Extraction of `has_returned` runs the computation and then returns true; it thus either returns true, or does not terminate.[19] From the Coq point of view, the function can return true or false; however the false result is never observed in extract code. In Chamois CompCert, this exit is applied to its translation validators which involve hash-consed symbolic executions [Gou23]. These nondeterministic computations check that some result is correct, terminates if it is, and loops forever or fails when it is not.[20] In summary, `has_returned` and `det_coerce` are two complementary exits of the monad.

[17] It is however commonplace to direct extraction to extract these types to the corresponding ones in OCaml's standard library, to avoid duplicating these with custom types.

[18] Perhaps the extractor could use new features in OCaml, such as GADTs or System-F polymorphism inside records, to recover some typing. This is however not implemented.

[19] These axioms have a trivial model (just return false), which ensures consistency. However, contrary to the one of `det_coerce`, it cannot be used for practical computations.

[20] The monad may be equipped with a *fail* operator (extracted to OCaml code that raises an exception) and loop and fixpoint constructs, which do not need a termination proof.

References

[BJM14] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. "Implementing and Reasoning About Hash-consed Data Structures in Coq". In: *Journal of Automated Reasoning* (June 2014), pp. 1–34. ISSN: 0168-7433. DOI: 10.1007/s10817-014-9306-0. HAL: hal-00816672.

[Bou21] Sylvain Boulmé. "Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles)". See also `http://www-verimag.imag.fr/~boulme/hdr.html`. Habilitation à diriger des recherches. Université Grenoble-Alpes, Sept. 2021. HAL: tel-03356701.

[CBN24] Clément Chavanon, Frédéric Besson, and Tristan Ninet. "PfComp: A Verified Compiler for Packet Filtering Leveraging Binary Decision Diagrams". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2024. London, UK: Association for Computing Machinery, 2024, 89–102. ISBN: 9798400704888. DOI: 10.1145/3636501.3636954. URL: `https://gitlab.inria.fr/cchavano/pfcomp/-/raw/main/paper/cpp2024-preprint.pdf?ref_type=heads&inline=false`.

[FST24] Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. "Verified Extraction from Coq to OCaml". In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: 10.1145/3656379. HAL: hal-04329663. URL: `https://doi.org/10.1145/3656379`.

[Gou23] Léo Gourdin. "Formal Validation of Intra-Procedural Transformations by Defensive Symbolic Simulation". Theses. Université Grenoble Alpes, Dec. 2023. HAL: tel-04648091.

[Jou16] Jacques-Henri Jourdan. "Verasco: a Formally Verified C Static Analyzer". PhD thesis. Université Paris 7 Diderot, May 2016. HAL: tel-01327023.

[Knu11] Donald E. Knuth. *The Art of Computer Programming*. Vol. 4A. Addison-Wesley, 2011. ISBN: 0-201-03804-8.

[Ler09] Xavier Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (2009). DOI: 10.1145/1538788.1538814.

[Let04] Pierre Letouzey. "Programmation fonctionnelle certifiée : L'extraction de programmes dans l'assistant Coq. (Certified functional programming : Program extraction within Coq proof assistant)". PhD thesis. University of Paris-Sud, Orsay, France, 2004. HAL: tel-00150912.

[Let08] Pierre Letouzey. "Extraction in Coq: An Overview". In: *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*. Vol. 5028. Lecture Notes in Computer Science. Springer, 2008, pp. 359–369. DOI: 10.1007/978-3-540-69407-6_39. HAL: hal-00338973.

[LWM21] Dominique Larchey-Wendling and Jean-François Monin. "The Braga Method: Extracting Certified Algorithms from Complex Recursive Schemes in Coq". In: *Proof and Computation II*. World Scientific, Aug. 2021, pp. 305–386. DOI: 10.1142/9789811236488_0008. HAL: hal-03338785v1.

[Mad90] Jean-Christophe Madre. "PRIAM : un outil de vérification formelle de circuits intégrés digitaux". French. PhD thesis. École nationale supérieure des télécommunications, June 5, 1990.

[May23] Valeran Maytié. "Formalisation des dictionnaires en Coq". MA thesis. Université Paris Saclay, July 2023. URL: `https://valeran-maytie.fr/pdf/stage_L3.pdf`.

[MB22] David Monniaux and Sylvain Boulmé. "The Trusted Computing Base of the CompCert Verified Compiler". In: *European Symposium on Programming Languages and Systems (ESOP '22)*. Vol. 13240. Lecture Notes in Computer Science. Springer, 2022, pp. 204–233. DOI: 10.1007/978-3-030-99336-8_8. HAL: hal-03541595v1.

[MB24] David Monniaux and Sylvain Boulmé. "Chamois: agile development of CompCert extensions for optimization and security". In: *Journées francophones des langages applicatifs (JFLA)*. Jan. 2024. HAL: hal-04406465.

[MB88] Jean Christophe Madre and Jean-Paul Billon. "Proving Circuit Correctness Using Formal Comparison Between Expected and Extracted Behaviour". In: *Proceedings of the 25th ACM/IEEE Conference on Design Automation, DAC '88, Anaheim, CA, USA, June 12-15, 1988*. Ed. by Dennis W. Shaklee and A. Richard Newton. ACM, 1988, pp. 205–210. DOI: 10.1109/DAC.1988.14759.

[RL10] Silvain Rideau and Xavier Leroy. "Validating register allocation and spilling". In: *Compiler Construction (CC 2010)*. Vol. 6011. Springer, 2010, pp. 224–243. URL: http://gallium.inria.fr/~xleroy/publi/validation-regalloc.pdf.

[VGL00] Kumar Neeraj Verma and Jean Goubault-Larrecq. *Reflecting BDDs in Coq*. Research Report RR-3859. Projet COQ. INRIA, 2000. HAL: inria-00072797.