

Reflecting Symbolic Model Checking in Coq

Kumar Neeraj Verma

Report on DEA internship completed under direction of
Jean Goubault-Larrecq

at
GIE Dyade and INRIA

September 2000

Abstract

We describe an implementation and a proof of correctness of a symbolic model checker for the μ -calculus using BDDs, completely formalized in the Coq proof assistant. This gives us a certified model checker which can run as a sub-system of Coq and provides a safe way of integrating symbolic model checking techniques inside the Coq proof assistant. Coq's extraction mechanism also gives us a certified model checker running in Caml.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 3 |
| 2 | Binary Decision Diagrams | 4 |
| 3 | The μ-Calculus | 5 |
| 4 | Coq | 8 |
| 5 | Implementation of BDDs | 11 |
| 6 | Other BDD algorithms: Quantification, Substitution | 16 |
| 7 | Implementation of μ-calculus | 21 |
| 7.1 | μ -Calculus Formulae | 21 |
| 7.2 | Semantics of μ -Calculus Formulae | 23 |
| 7.3 | Monotonic Functions on Boolean Expressions | 28 |
| 7.4 | Conversion from μ -Calculus Formulae to BDDs | 35 |
| 8 | Conclusion | 41 |
| 8.1 | Related Work | 41 |
| 8.2 | Possible Extensions | 42 |

Chapter 1

Introduction

The work described in this report was done as part of a DEA internship at GIE Dyade and INRIA between July and September 2000, under direction of Jean Goubault-Larrecq. The implementation is based upon a BDD library developed in Coq during an internship at GIE Dyade and INRIA between May and July 1999, and extended to include garbage collection as part of a BTech. project at IIT Delhi between July 1999 and May 2000.

We describe an implementation and a proof of correctness of a symbolic model checker for the μ -calculus using BDDs, completely formalized in the Coq proof assistant. This provides us a proved model checker which can run inside the Coq proof assistant. The aim is to provide symbolic model checking techniques based on BDDs inside the Coq proof assistant.

Combining the techniques of model checking and theorem proving is known to have several advantages for verification of hardware and software. Proof assistants are expressive in that they allow us not only to talk about boolean functions and finite state systems, but almost any branch of mathematics. They are also considered more reliable compared to normal model checkers whose correctness is a cause of concern. However, using proof assistants requires considerable time and effort, as well as skills. On the other hand, model checkers are completely automatic and are able to solve huge verification problems. Therefore combining these two techniques allows us to use the advantages of both of them together.

This can be done by using the technique of reflection (see Section 4) which allows us to replace proofs by computations. The Coq proof assistant is particularly suitable for this purpose since it is based on the λ -calculus, and hence contains a programming language.

The report is organized as follows. We use Binary Decision Diagrams for symbolic representation of sets of states and relations in our model checking algorithm. BDDs are described in Chapter 2 and the μ -calculus in Chapter 3. We give an overview of Coq in Chapter 4. The previous implementation of BDDs which we require for our model checker is briefly described in Chapter 5. This implementation also contains a proved garbage collector which can be invoked from within the BDD algorithms. We extend this implementation with some more operations required for our purposes, which are described in Chapter 6. We describe our implementation of the μ -calculus in Chapter 7. We conclude in Chapter 8.

Chapter 2

Binary Decision Diagrams

BDDs represent a propositional logic formulas (built from variables and the connectives $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$) as graphs. It is based on Shannon's decomposition of any formula F as $(A \Rightarrow F[A := \mathbf{1}]) \wedge (\neg A \Rightarrow F[A := \mathbf{0}])$, where $(F[A := G])$ denotes substitution of A by G in F , $\mathbf{1}$ is **true** and $\mathbf{0}$ is **false**. We shall also use the notation $F = A \longrightarrow F[A := \mathbf{1}]; F[A := \mathbf{0}]$. By choosing a variable A occurring in F and decomposing it and recursively continuing the process for $F[A := \mathbf{1}]$ and $F[A := \mathbf{0}]$, we get a binary tree (a *Shannon tree*) whose internal nodes have variables and leaf nodes are $\mathbf{1}$ and $\mathbf{0}$.

BDDs are *shared*, *reduced* and *ordered* versions of Shannon trees. *Sharing* means that all isomorphic subtrees are stored at the same address in memory. This gives us directed acyclic graphs instead of trees. It is also the main reason why BDDs are very small in many applications. Sharing is accomplished by having a sharing map which remembers all the BDDs that have been previously constructed. *Reducing* means that a BDD node with identical children represents a redundant comparison and can be replaced by a child node. *Ordering* means that we fix some ordering on the variables and have all the variables along any path in a BDD occur in that order. These conditions ensure that BDDs are canonical forms for propositional formulae up to propositional equivalence [Bry86].

One of the reasons why BDDs are interesting is that they are usually compact. But BDDs are also interesting because they are easy to work with, at least in principle: all the usual logical operations are easily definable by recursive descent on BDDs. Negation, for example, is defined by $\neg \mathbf{1} =_{\text{df}} \mathbf{0}$, $\neg \mathbf{0} =_{\text{df}} \mathbf{1}$, $\neg(A \longrightarrow F; G) =_{\text{df}} A \longrightarrow \neg F; \neg G$, and all other Boolean operations \oplus ($\vee, \wedge, \Rightarrow, \Leftrightarrow$, etc.) are defined by $(A \longrightarrow F_1; G_1) \oplus (A \longrightarrow F_2; G_2) =_{\text{df}} A \longrightarrow (F_1 \oplus F_2); (G_1 \oplus G_2)$, with the usual definitions on $\mathbf{1}$ and $\mathbf{0}$. This is called *orthogonality* of \oplus .

By remembering in a cache the results of previous computations, negation can be computed in linear time, and binary operations in quadratic time w.r.t. the sizes of the input BDDs. (This is sometimes called *memoizing* or *tabulating*.)

BDDs are usable in symbolic model checking applications because of the fact that we can represent finite sets of states, as well as relations on them, symbolically using BDDs [McM93]. The precise representation that we use in our implementation is described in Chapter 7. The symbolic representation of sets of states and relations has in practice been found to be quite efficient compared to previous techniques where the complete graph of the transition relations needed to be built. This is because of the fact that the symbolic representation exploits the regularities in the hardware systems, and are thus usually smaller in size.

Chapter 3

The μ -Calculus

The set of formulae of the μ -calculus is defined with respect to a set AP of *atomic propositions*, a set VAR of *relational variables* and a set T of *transition variables* as follows.

Definition 1 *Syntax of μ -calculus formulae:*

- If $p \in AP$, then p is a formula.
- If $P \in VAR$ then P is a formula.
- If f is a formula, then $\neg f$ is a formula.
- If f and g are formulae, then $f \wedge g$ is a formula.
- If f is a formula and $t \in T$, then $[t]f$ is a formula.
- If f is a formula and $P \in VAR$, then $\mu P.f$ is a formula.

In the formula $\mu P.f$, all free occurrences of P in f must be under an even number of negations.

The formulae of μ -calculus are interpreted with respect to a transition system $M = (S, APenv, Tenv)$, where S is a finite set of states, $APenv : AP \rightarrow 2^S$ and $Tenv : T \rightarrow 2^{S \times S}$.

A μ -calculus formula is supposed to represent a set of states in S where the formula holds. Atomic propositions are mapped to a set of states as defined by the environment $APenv$. \neg and \wedge are interpreted as complementation and intersection respectively. The formula $[t]f$ represents the set of all those states s such that all states s' reachable from s in one step (as defined by the transition relation $Tenv(t)$) satisfy the formula f . Relational variables also represent a set of states, however they differ from atomic propositions in that they can be bound by the fixpoint operator μ . This is the reason why we introduce two different kinds of variables for representing set of states. The formula $\mu P.f$ intuitively represents $f(\phi) \cup f(f(\phi)) \cup f(f(f(\phi))) \cup \dots$

We define below formally the interpretation $\llbracket f \rrbracket_{Me}$ of a formula f with respect to a transition system M and an environment $e : VAR \rightarrow 2^S$. It returns the set of states in which f holds.

Definition 2 *Semantics of μ -calculus formulae:*

- $\llbracket p \rrbracket_{Me} = APenv(p)$
- $\llbracket P \rrbracket_{Me} = e(P)$
- $\llbracket \neg f \rrbracket_{Me} = S \setminus \llbracket f \rrbracket_{Me}$
- $\llbracket f \wedge g \rrbracket_{Me} = \llbracket f \rrbracket_{Me} \cap \llbracket g \rrbracket_{Me}$
- $\llbracket [t]f \rrbracket_{Me} = \{s \mid \forall s' ((s, s') \in Tenv(t) \Rightarrow s' \in \llbracket f \rrbracket_{Me})\}$
- $\llbracket \mu P.f \rrbracket_{Me}$ is the least fixpoint of the function $\tau : 2^S \rightarrow 2^S$ defined as $\tau(A) = \llbracket f \rrbracket_{Me}[P \mapsto A]$.

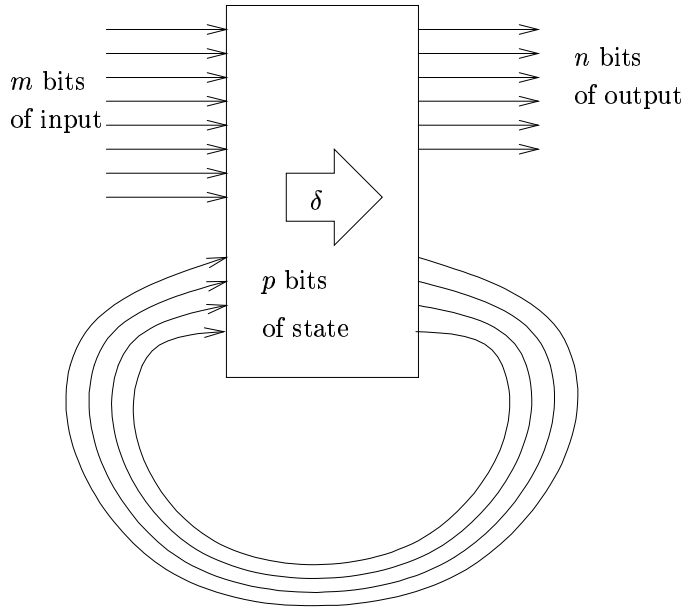


Figure 3.1: A sequential machine

The semantics of the fixpoint operator is well defined provided that the use of negations in formulae is restricted as mentioned above. This guarantees that the function τ defined above is monotonic and guarantees existence of the least fixed point (by Tarski's theorem.) Also S is finite. This ensures that the least fixed point of τ is equal to $\tau^i(\phi)$ for some integer i and hence can be computed iteratively starting from the empty set ϕ , and successively applying the function τ and checking at each step whether a fixed point has been reached. The bound on the number of iterations required to reach this fixpoint is equal to the cardinality of S .

The prototypical example of model-checking comes from the domain of hardware verification. Implementations, or circuits, are described as sequential machines with m bits of input, n bits of output, and p bits of internal state. Sequential machines are driven by an external clock, and at each clock tick, the transition relation δ of the sequential machine is computed from the current input and the current state, to yield the next output and the next internal state, as depicted in Figure 3.1. The transition δ may map input/current state combinations to several output/next state combinations, thus expressing non-determinism.

Formally, the transition is a relation δ from $m + p$ -tuples of booleans to $n + p$ -tuples of booleans. A sequential machine defines a transition system whose states are $(m + n + p)$ -tuples of booleans, or bits, representing the current input, the current output and the current state, and whose transition relation R is given by $(i_1, \dots, i_m, s_1, \dots, s_p, o_1, \dots, o_n) R (i_1, \dots, i_m, s'_1, \dots, s'_p, o_1, \dots, o_n)$ if and only if $(i_1, \dots, i_m, s_1, \dots, s_p) \delta (s'_1, \dots, s'_p, o_1, \dots, o_n)$.

We now describe what the technique of *symbolic model-checking* is all about. This technique, in combination with BDDs, has been widely successful in hardware verification. For more information, see [McM93]. The transition relation has 2^{m+n+p} states, and so it is impossible to represent it as a graph in memory for even small values of m , n , and p . But we can represent it as a propositional formula, which we'll denote by δ again, built on $m + n + 2p$ variables $i_1, \dots, i_m, s_1, \dots, s_p, s'_1, \dots, s'_p, o_1, \dots, o_n$.

The variables s_1, \dots, s_p are called *state variables*. A state of the sequential machine is identified with an assignment to the state variables, of the input variable and of the output variables. Then, every boolean expression over the set of input, state and output variables denotes a set of input/state/output configurations (or states): the set of assignments that satisfy the formula. For example, if A and B are two state variables, then $A \vee B$ represents the set of all states in which A is true or in which B is true.

The basic idea of symbolic model-checking is, given a transition system represented as a propositional formula δ on input, current state, next state and output variables, and given a μ -calculus formula Φ to be checked on the transition system, to compute a boolean expression Ψ on the input, state and output variables representing the set of states where the formula Φ holds. Then, Φ holds on the given transition system if and only if Ψ is valid. Since BDDs can represent boolean expressions, we can use them for model checking in this way.

The μ -calculus is expressive enough to describe the usual properties of finite state systems which we require in model checking. Formulas of other logics which are commonly used in model checking (like CTL and CTL*) can be translated in a straightforward way to formulas to μ -calculus.

Chapter 4

Coq

Coq is a proof assistant based on the Calculus of Inductive Constructions (CIC), a type theory that is powerful enough to formalize large parts of mathematics. It properly includes higher-order intuitionistic logic, augmented with definitional mechanisms for inductively defined types, sets, propositions and relations. CIC is also a typed λ -calculus, and can therefore also be used as a programming language. For a gentle introduction, see [HKPM98]. We shall describe here the main features of Coq that we shall need later.

The *sorts* of CIC are Prop, the sort of all propositions; Set, the sort of all specifications, programs and data types; and Type_i , $i \in \mathbb{N}$. The typing rules for sorts include $\text{Prop} : \text{Type}_0$, $\text{Set} : \text{Type}_0$, $\text{Type}_i : \text{Type}_{i+1}$.

If F and G are propositions, $F \rightarrow G$ is a proposition (read: “ F implies G ”), and $(x : T)G$ is a proposition, for any type T (read: “for every x of type T , G ”). In turn, any object $\pi : F$ is a *proof* π of F . Internally, proofs are represented as λ -terms, but Coq provides for interactive development of proofs in a top-down manner using a set of *tactics* which allow goals to be broken into subgoals.

Any object $t : \text{Set}$ denotes a data type, like the data type `nat` of natural numbers, or the type `nat -> nat` of all functions from naturals to naturals. Data types can be defined inductively. The standard definition of the type `nat` of natural numbers in Coq reads:

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

Abstractions are written using the notation $[x:T]f$, which denotes a function with formal parameter x of type T and body f . Application of a function p to an argument q is written as $(p\ q)$ and $(\dots((p\ q_1)\ q_2)\ \dots\ q_n)$ is abbreviated as $(p\ q_1\ \dots\ q_n)$. Addition of natural numbers can be defined as follows. It also introduces the syntax for defining recursive functions using the `Fixpoint` keyword. We can do case analysis on elements of an inductive type (like `nat` defined above) using the `Cases` keyword.

```
Fixpoint plus [n:nat] : nat -> nat :=
  [m:nat] Cases n of
    0 => m
    | (S p) => (S (plus p m))
  end
```

`Fixpoint` definitions require the recursive function to be defined using structural induction on the last argument. This is the mechanism used by Coq to ensure that only total functions (which terminate on all arguments) can be defined, although this conditions sometimes leads to problems because many functions are impossible to define even though their termination may be obvious.

In general, propositions and programs can be mixed. We shall exploit this mixing of propositions and programs in a very limited form: when π and π' are programs (a.k.a., descriptions of data), then $\pi = \pi'$ is a proposition, expressing that π and π' have the same value. Reflection can then be implemented as follows: given a property $P : t \rightarrow \text{Prop}$ write a program $\pi : t \rightarrow \text{bool}$. Then prove the correctness lemma: $(x : t) (\pi\ x) = \text{true} \rightarrow (P\ x)$ (“for every x of type t , if π returns `true` on input x , then P holds of x ”).

Note that `bool` is defined by `Inductive bool : Set := true : bool | false : bool.` and should be distinguished from `Prop`. To show that P holds of some concrete data x_0 , you can then compute (πx_0) in Coq, viewed as a programming language. If the result is `true`, then $(\pi x_0)=\text{true}$ holds, and the correctness lemma then allows you to conclude that $(P x_0)$ holds in Coq, viewed as a logic. Our use of reflection will roughly follow these lines.

The development of BDDs in Coq will also rest on several predefined libraries of proofs and code. This includes the theories of Peano arithmetic, Booleans, Lists and also a library of maps [GL98]. This is a theory of finite sets and maps, and their implementation in Coq as radix-exchange tries [Knu73], following closely the implementation in [Gou94]. The fundamental data types are:

```
Inductive option [A:Set] : Set :=
  NONE : (option A)
  | SOME : A -> (option A).
```

which is intended to represent the presence of some data $a : A$ (this is `(SOME A a)`, of type `(option A)`), or the absence of any data of type A (this is `(NONE A)`, again of type `(option A)`); and:

```
Inductive ad : Set :=
  ad_z : ad
  | ad_x : positive -> ad.
```

which is a type of *addresses*. The `ad_z` constant denotes 0, while `(ad_x p)` denotes the strictly positive number p . The `positive` data type is defined as part of the ZARITH standard Coq theory of arithmetic in \mathbb{Z} , and is defined as:

```
Inductive positive : Set :=
  xI : positive -> positive
  | x0 : positive -> positive
  | xH : positive.
```

The intended semantics is that `xH` denotes 1, `x0` is multiplication by 2 (appending a 0 bit at the end of a number, in binary), and `xI` is $x \mapsto 2x + 1$ (appending a 1 bit at the end of x in binary). Elements of `ad` are viewed either as finite lists of bits (in a contorted way) or as natural numbers (in an unusual format).

Equality on addresses is decidable, and there is a function `ad_eq` of type `ad -> ad -> bool` such that `(ad_eq a a')` equals `true` if $a=a'$, and `false` otherwise. Similarly, `ad_le` : `ad -> ad -> bool` is such that `(ad_le a a')` returns `true` whenever $a \leq a'$, and `false` otherwise.

The maps of [GL98] always map addresses (of type `ad`) to elements of some type A . Given $A : \text{Set}$, the type of maps from `ad` to A is `(Map A)`. The following functions are provided, which we shall be interested in:

- `(M0 A)` or `(newMap A)` is the empty map from `ad` to A .
- `MapGet` : $(A:\text{Set})(\text{Map } A) \rightarrow \text{ad} \rightarrow (\text{option } A)$; `(MapGet A m x)` reads the image y of x under the map m , and either returns `(SOME A y)` if it exists, or else `(NONE A)`; we also use this function to define the *semantics* of a map m as the function `(MapGet A m)`. The latter is then an encoding as a total function f of type `ad -> (option A)` of the partial function from `ad` to A mapping every x such that $f(x)$ is of the form `(SOME A y)` to y .
- `MapPut` : $(A:\text{Set})(\text{Map } A) \rightarrow \text{ad} \rightarrow A \rightarrow (\text{Map } A)$; `(MapPut A m x y)` adds the binding from x to y to the map m , and returns the new map. The newly added binding hides any previous binding of x in m . Semantically, it returns the partial function mapping x to y , and every $z \neq x$ to $m(z)$.
- `MapRemove` : $(A:\text{Set})(\text{Map } A) \rightarrow \text{ad} \rightarrow (\text{Map } A)$; `(MapRemove A m x)` removes x from the domain of m , that is, it returns a map mapping every $z \neq x$ to $m(z)$.
- `in_dom` : $(A:\text{Set})\text{ad} \rightarrow (\text{Map } A) \rightarrow \text{bool}$; `(in_dom A x m)` checks whether $m(x)$ is defined, i.e., it returns `true` if and only if `(MapGet A m x)` does not return `(NONE A)`.

Note that, although we have to provide the type `A` explicitly in `(MapPut A m x y)`, it is also possible to let Coq infer it, by replacing the required type by a question mark, as in `(MapPut ? m x y)`. The `map` library contains also a few other useful functions on maps and generic iterators over finite maps, which we won't use, and a collection of theorems on maps, which we shall mention when we need them. Finite sets of addresses are considered to be just maps from `ad` to the one-element type `unit`, that is, sets are represented as the domains of maps with trivial codomain. Finally, all these functions are defined as programs, they are not just axiomatized; therefore they provide a running implementation of finite maps and sets in Coq.

There is also a characterization of canonical maps, defined by the predicate `mapcanon`. If two canonical maps represent the same function, then they are equal.

Our proofs also make use of a theory of sets and cardinalities of finite sets, in the library `SETS`. It defines an *ensemble* on a type `U` as a predicate on `U`.

Definition `Ensemble := [U:Type]U->Prop.`

Ensembles with finite cardinalities are characterized by the predicate `Finite`. `(Finite U A)` means that `A` is finite.

`Finite : (U:Type)(Ensemble U)->Prop`

Cardinalities of finite sets are defined by a relation. `(cardinal U A n)` means that the ensemble `A` has cardinality `n`.

`cardinal : (U:Type)(Ensemble U)->nat->Prop`

Images of functions are defined by `Im`. `(Im U V A f)` is the ensemble containing all elements of the form `(f x)` where `x` is of type `U` and is in the ensemble `A`.

`Im : (U:Type; V:Type)(Ensemble U)->(U->V)->(Ensemble V)`

There are several results related to cardinalities of finite ensembles, like, all finite ensembles (i.e. satisfying the predicate `Finite`) have a (finite) cardinality (i.e. satisfy the `cardinal` predicate for some integer,) and the converse, images of functions over finite ensembles are finite and have cardinalities less than the cardinality of the domain, etc.

Chapter 5

Implementation of BDDs

We describe here the earlier implementation of BDDs [VGL00, VGLPAK] which we use for our μ -calculus model checker. In this implementation, BDDs are modeled inside Coq, the logical operations $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ are implemented and their correctness is proved. This gives a completely proved tautology checker for boolean expressions. This implementation is parametrized by a garbage collector, i.e., the tautology checker and its proof of correctness are based upon some assumptions about the garbage collector algorithm. A garbage collector is then implemented which is shown to satisfy these assumptions. This modular approach also gives the flexibility to change the garbage collection algorithm at any point (like changing the conditions when the garbage collector should be called) independently of the BDD algorithms. We briefly describe this implementation here. We give the main definitions and theorems that we require for implementing the model checker. See [VGL00] for a more detailed description, although it describes a somewhat older version of the implementation which moreover does not use garbage collection. We describe the more recent version in [VGLPAK], although it is a bit short.

We also implement some more BDD operations which are required for the implementation of the model checker. These are described in Chapter 6

BDDs are modeled using the map library described in Chapter 4. To define BDDs, we first need to model a store, which represents the memory in which all the BDDs are to be stored. The store is modeled as a finite map from addresses to triples (x, l, r) , where x is a variable and l and r are addresses representing children BDD nodes. Both variables and addresses belong to the type `ad`.

Definition 3

Definition `BDDvar := ad`.

Definition `BDDstate := (Map (BDDvar * (ad * ad)))`.

Thus, an address in the BDD state represents a BDD whose root is at that address. The other option was not to model BDDs using stores, but to define an inductive data type for BDDs, so that BDDs would have been trees. However, in that case, it would have been impossible to do sharing. The constants `BDDzero` and `BDDone` are the first two addresses in the BDD state, and are reserved for the leaf nodes. Sharing is implemented by having a sharing map which maps triples (x, l, r) to addresses. This is defined by the type `BDDsharing_map` defined below.

Definition 4

Definition `BDDsharing_map := (Map (Map (Map ad)))`.

A sharing map as defined above, represents in curried form, maps from left hand addresses `l` to maps from right hand addresses `r` to maps from variables `x` to addresses storing the triple $(x, (l, r))$. We need to define it in this complex way because of the fact that the domain of maps can only be addresses and not tuples of addresses.

Memoization tables are similarly implemented in curried form using maps. For negation, they map addresses to addresses. For disjunction, they map pairs of addresses to addresses.

Definition 5

Definition `BDDneg_memo := (Map ad)`.

Definition `BDDor_memo := (Map (Map ad))`.

Since we allow garbage collection, we also need to have a list of nodes that are freed during garbage collection, so that they can be reused later. Also there is a counter of type `ad`, which is an address in the BDD state after which all addresses are unused.

Definition 6

Definition `BDDfree_list := (list ad)`.

Later on, we also need to implement quantification over BDDs (Chapter 6). For this purpose we add an additional memoization table for universal quantification, which maps a pair of a node and a variable to another node.

Definition 7

Definition `BDDuniv_memo := (Map (Map ad))`.

Definition 8

Definition `BDDconfig := BDDstate * BDDsharing_map * BDDfree_list * ad * BDDneg_memo * BDDor_memo * BDDuniv_memo`.

The invariants on the various parts of the configuration are defined by the predicate `BDDconfig_OK` which says that the BDDs are well formed (i.e. they are directed acyclic graphs together with reducedness, ordering conditions) and that the various parts of the configuration are consistent (the memoization maps refer to nodes which are OK, the sharing map and the BDD state represent inverse functions, etc.) A node in a configuration is said to be OK if it is in the domain of the BDD state or is a leaf node. This is defined by the predicate `config_node_OK`. The BDD algorithms also take in as argument, a list of nodes representing BDDs that should not be freed by the garbage collector. This is of type `(list ad)`. The predicate `used_list_OK` is used to define the invariant that all the nodes in this list are OK.

`BDDconfig_OK : BDDconfig -> Prop`

`config_node_OK : BDDconfig -> ad -> Prop`

`used_list_OK : BDDconfig -> (list ad) -> Prop`

Boolean functions are represented as maps from environments to booleans, where an environment maps variables to booleans.

Definition 9

Definition `var_env := BDDvar->bool`.

Definition `bool_fun := var_env->bool`.

Extensional equality is defined by `bool_fun_eq`. The constant boolean functions are defined by `bool_fun_zero` and `bool_fun_one`. The usual logical operations are defined by `bool_fun_neg`, `bool_fun_or`, `bool_fun_and`, `bool_fun_impl` and `bool_fun_iff`.

BDDs are interpreted as boolean functions in the usual manner by the function `bool_fun_of_BDD`.

```
bool_fun_of_BDD : BDDconfig -> ad -> bool_fun
```

The canonicity of BDDs is stated by the lemma `BDDunique` which says that if two nodes represent the same boolean function then they are equal (so the two BDDs are not only isomorphic, but are stored at the same address.)

Lemma 1

Lemma `BDDunique` :

```
(cfg:BDDconfig; node1,node2:ad)
(BDDconfig_OK cfg)
-> (config_node_OK cfg node1)
-> (config_node_OK cfg node2)
-> (bool_fun_eq (bool_fun_of_BDD cfg node1) (bool_fun_of_BDD cfg node2))
-> (ad_eq node1 node2)=true.
```

The operations on BDDs (`BDDneg`, `BDDor`, `BDDand`, `BDDimpl`, `BDDiff`) are defined and proved correct with respect to the semantics of BDDs. These algorithms make use of the function `BDDmake` which takes as input a variable x and nodes l and r and returns a new node (in a new configuration) representing the boolean function " $x \rightarrow bfl; bfr$ ", where bfl and bfr are the boolean functions represented by l and r respectively.

These algorithms are parametrized by the garbage collector, giving the flexibility to change the garbage collector independently. The type of a garbage collector is `BDDconfig->(list ad)->BDDconfig`. It takes in an input configuration and input list of used nodes, representing BDDs which should not be removed by the garbage collector, and returns a new configuration.

The proofs of the BDD algorithms are based on some assumptions about the garbage collector. These assumptions are defined by the predicate `gc_OK`. It says that provided the input configuration and list of used nodes are OK, the new configuration returned by the garbage collector is OK, the contents of the nodes reachable in the old configuration from the input used list are preserved, and no new nodes are added into the configuration.

The type of the functions `BDDneg` and `BDDor` are given below. `BDDneg` takes a garbage collector of type `BDDconfig->(list ad)->BDDconfig`, a configuration of type `BDDconfig`, an input list of used nodes of type `(list ad)`, an input node of type `ad` (representing the BDD whose negation is to be computed,) and returns a new configuration, and a new node representing the output BDD in that configuration.

```
BDDneg : (BDDconfig->(list ad)->BDDconfig)
-> BDDconfig -> (list ad) -> ad -> BDDconfig*ad
```

```
BDDor : (BDDconfig->(list ad)->BDDconfig)
-> BDDconfig -> (list ad) -> ad -> ad -> BDDconfig*ad
```

Since these algorithms change the store, we need to prove that the new configuration returned is OK, the used nodes from the old configuration are preserved, etc. For example, the lemmas proved about `BDDneg` are as follows. (`used_node' cfg ul node`) means that `node` is either reachable from some node in the list `ul` or is a leaf node. The predicate `used_nodes_preserved` means that the contents of all nodes reachable from the used list in the old configuration are preserved in the new configuration.

```
Lemma BDDneg_config_OK : (gc:(BDDconfig->(list ad)->BDDconfig))
(gc_OK gc)
->(cfg:BDDconfig; ul:(list ad); node:ad)
(BDDconfig_OK cfg) ->(used_list_OK cfg ul) ->(used_node' cfg ul node)
->(BDDconfig_OK (Fst (BDDneg gc cfg ul node))).
```

```
Lemma BDDneg_node_OK : (gc:(BDDconfig->(list ad)->BDDconfig))
(gc_OK gc)
->(cfg:BDDconfig; ul:(list ad); node:ad)
```

```
(BDDconfig_OK cfg) ->(used_list_OK cfg ul) ->(used_node' cfg ul node)
->(config_node_OK (Fst (BDDneg gc cfg ul node)) (Snd (BDDneg gc cfg ul node))).
```

```
Lemma BDDneg_used_nodes_preserved : (gc:(BDDconfig->(list ad)->BDDconfig))
(gc_OK gc)
->(cfg:BDDconfig; ul:(list ad); node:ad)
(BDDconfig_OK cfg) ->(used_list_OK cfg ul) ->(used_node' cfg ul node)
->(used_nodes_preserved cfg (Fst (BDDneg gc cfg ul node)) ul).
```

```
Lemma BDDneg_is_neg : (gc:(BDDconfig->(list ad)->BDDconfig))
(gc_OK gc)
->(cfg:BDDconfig; ul:(list ad); node:ad)
(BDDconfig_OK cfg) ->(used_list_OK cfg ul) ->(used_node' cfg ul node)
->(bool_fun_eq (bool_fun_of_BDD (Fst (BDDneg gc cfg ul node))
(Snd (BDDneg gc cfg ul node)))
(bool_fun_neg (bool_fun_of_BDD cfg node))).
```

Boolean expressions are defined as follows.

Definition 10

```
Inductive bool_expr : Set :=
  Zero : bool_expr
| One : bool_expr
| Var : BDDvar -> bool_expr
| Neg : bool_expr -> bool_expr
| Or : bool_expr -> bool_expr -> bool_expr
| And : bool_expr -> bool_expr -> bool_expr
| Impl : bool_expr -> bool_expr -> bool_expr
| Iff : bool_expr -> bool_expr -> bool_expr.
```

These are interpreted as boolean functions in the obvious way by the function `bool_fun_of_bool_expr`. The tautology checker for boolean expressions is `is_tauto` which builds the BDD for the boolean expression and checks whether the resulting node is `BDDone`.

```
is_tauto : (BDDconfig->(list ad)->BDDconfig)->bool_expr->bool
```

```
Lemma is_tauto_lemma : (gc:(BDDconfig->(list ad)->BDDconfig))
(gc_OK gc)
->(be:bool_expr)
(is_tauto gc be)=true<->(bool_fun_eq bool_fun_one (bool_fun_of_bool_expr be)).
```

As can be seen, all the above definitions and proofs work for an arbitrary garbage collector which satisfies the conditions mentioned in `gc_OK`.

A garbage collector is then separately implemented which is shown to satisfy the conditions in `gc_OK`.

```
gc_0 : BDDconfig ->(list ad) -> BDDconfig
```

```
Lemma gc_0_OK : (gc_OK gc_0).
```

This is a mark and sweep garbage collector (similar to the one described in [Gou94]) which works by first marking all the nodes in the BDD state which are reachable from the used list. In the first sweep phase, the BDD state is scanned and all unmarked nodes are removed and added to the free list. In the next sweep phases, the sharing and memoization maps are scanned to remove all references to nodes that have been freed.

The BDD algorithms use the garbage collector by calling the garbage collection routine (which is passed as an argument to the algorithms) every time a node needs to be allocated. Of course, this does not mean that the whole BDD state is rescanned every time the allocation function is called, because the function `gc` passed as argument may choose not to do anything. For example, we have the following such function which also satisfies all the required conditions in `gc_OK`.

Definition `gc_inf := [cfg:BDDconfig; _:(list ad)]cfg`.

Lemma `gc_inf_OK : (gc_OK gc_inf)`.

In practice, we define some condition on the configuration and free list. The garbage collector function then calls `gc_0` if these conditions are satisfied, otherwise it calls `gc_inf`. This function can again be easily shown to satisfy the condition `gc_OK`. For example, the condition could be that there are no free nodes and the counter is greater than a certain threshold value.

Chapter 6

Other BDD algorithms: Quantification, Substitution

In this chapter, we describe our implementation of some other operations on BDDs that are required for the model checker.

We define substitution of a variable x by a boolean expression bex in another boolean expression be as follows.

Definition 11

```
Fixpoint subst [x:BDDvar;bex,be:bool_expr] : bool_expr :=
  Cases be of
    Zero => Zero
  | One => One
  | (Var y) => if (ad_eq x y) then bex else be
  | (Neg be1) =>(Neg (subst x bex be1))
  | (Or be1 be2) => (Or (subst x bex be1) (subst x bex be2))
  | (And be1 be2) => (And (subst x bex be1) (subst x bex be2))
  | (Impl be1 be2) => (Impl (subst x bex be1) (subst x bex be2))
  | (Iff be1 be2) => (Iff (subst x bex be1) (subst x bex be2))
end.
```

The corresponding operation on boolean functions is defined as follows.

Definition 12

```
Definition bool_fun_subst := [x:BDDvar; bfx,bf:bool_fun]
  [ve:var_env](bf (augment ve x (bfx ve))).
```

The function `augment` in the above definition is used to change the value to which a variable is mapped in an environment.

Restriction of a variable in a boolean expression or a boolean function to some value (i.e. $F[A := 0]$ or $F[A := 1]$) is defined as follows.

Definition 13

```
Definition bool_to_be := [b:bool]Cases b of true => One | false => Zero end.
```

```
Definition restrict := [x:BDDvar; b:bool; be:bool_expr]
  (subst x (bool_to_be b) be).
```

```

Definition bool_fun_restrict := [bf:bool_fun; x:BDDvar; b:bool]
  [vb:var_env](bf (augment vb x b)).

```

We first implement universal quantification on BDDs. The corresponding definitions on boolean expressions and boolean functions are as follows.

Definition 14

```

Definition bool_fun_forall := [x:BDDvar; bf:bool_fun]
  (bool_fun_and (bool_fun_restrict bf x true)
    (bool_fun_restrict bf x false)).

```

```

Definition forall := [x:BDDvar; be:bool_expr]
  (And (restrict x true be) (restrict x false be)).

```

The following function `BDDuniv_1` computes the universal quantification over a variable `y` of a BDD represented by the node `node` in the configuration `cfg`. As in other BDD algorithms mentioned in Chapter 5, the garbage collector `gc` is passed as parameter. Recursion over BDDs is accomplished by passing a separate argument of type `nat` which is a bound on the maximum possible depth of recursion we expect. This bound can be easily computed by looking at the variable at the root node of the BDD, yielding the required function `BDDuniv`. We have a memoization table for universal quantification on BDDs. This is one of the components of the BDD configuration and can be obtained by the helper function `un_of_cfg`. The function `BDDuniv_1` first looks up this table and if a node corresponding to `node` and `y` is found, returns it. (The memoization map can be looked up by the function `MapGet2` which reads from a curried 2-argument map.) Otherwise it computes the universal quantification. It reads the contents of the BDD state (which is equal to `(Fst cfg)`.) If `node` is a leaf node, it returns the same node as result. Otherwise, if the variable `x` at `node` is smaller than `y`, then it again returns the same node. If `x` is equal to `y`, then it returns the conjunction of the two child nodes. If `x` is greater than `y`, then the quantification is recursively computed for the left and right children, `BDDmake` is called with the variable `x`, and the final result is memoized (using the function `BDDuniv_memo_put`.)

Definition 15

```

Fixpoint BDDuniv_1 [gc:BDDconfig->(list ad)->BDDconfig;
  cfg:BDDconfig; ul:(list ad); node:ad; y:BDDvar; bound:nat]
  : BDDconfig*ad :=
Cases bound of
0 => (* Error *) (initBDDconfig,BDDzero)
| (S bound') =>
Cases (MapGet2 ? (un_of_cfg cfg) node y) of
(SOME node') => (cfg,node')
| NONE =>
Cases (MapGet ? (Fst cfg) node) of
NONE => (cfg, node)
| (SOME (x,(l,r))) =>
Cases (BDDcompare x y) of
INFERIEUR => (cfg,node)
| EGAL => (BDDand gc cfg ul l r)
| SUPERIEUR =>
Cases (BDDuniv_1 gc cfg ul l y bound') of (cfgl,node1) =>
Cases (BDDuniv_1 gc cfgl (cons node1 ul) r y bound') of (cfgr,noder) =>
Cases (BDDmake gc cfgr x node1 noder (cons noder (cons node1 ul))) of
(cfg',node') => ((BDDuniv_memo_put cfg' y node node'),node')
end
end

```

```

end
end end end end.

```

Definition 16

```

Definition BDDuniv := [gc:(BDDconfig->(list ad)->BDDconfig); cfg:BDDconfig;
                      ul:(list ad); y:BDDvar; node:ad]
  (BDDuniv_1 gc cfg ul node y (S (nat_of_ad (var' cfg node)))).

```

The function `BDDuniv_1` (as well as the other BDD operations) is defined using *state threading*. Since each call to a function modifies the state, the state needs to be passed around through this sequence of calls, and the state returned by each call is used in the next call. Universal quantification over a list of variables is defined as follows.

Definition 17

```

Fixpoint bool_fun_univl [bf:bool_fun;la:(list ad)] : bool_fun :=
  Cases la of
  nil => bf
  | (cons a la') => (bool_fun_forall a (bool_fun_univl bf la'))
  end.

```

```

Fixpoint univl [be:bool_expr;la:(list ad)] : bool_expr :=
  Cases la of
  nil => be
  | (cons a la') => (forall a (univl be la'))
  end.

```

The corresponding BDD algorithm is defined as follows.

Definition 18

```

Fixpoint BDDunivl [gc:BDDconfig->(list ad)->BDDconfig;
                  cfg:BDDconfig; ul:(list ad); node:ad; ly:(list BDDvar)]
  : BDDconfig*ad :=
  Cases ly of
  nil => (cfg,node)
  | (cons y ly') => Cases (BDDunivl gc cfg ul node ly') of (cfg1,node1) =>
    (BDDuniv gc cfg1 (cons node1 ul) y node1)
  end
  end.

```

The substitution operation on BDDs is defined by the following function `BDDsubst`. The idea is the classical equivalence $F[x := G] \Leftrightarrow \forall x.(x \Leftrightarrow G) \Rightarrow F$.

Definition 19

```

Definition BDDsubst := [gc:BDDconfig->(list ad)->BDDconfig;cfg:BDDconfig;
                      ul:(list ad); node1:ad; x:BDDvar; node2:ad]
  Cases (BDDvar_make gc cfg ul x) of (cfg1,nodex) =>
  Cases (BDDiff gc cfg1 (cons nodex ul) nodex node2) of (cfg2,node3) =>
  Cases (BDDimpl gc cfg2 (cons node3 ul) node3 node1) of (cfg3,node4) =>
  (BDDuniv gc cfg3 (cons node4 ul) x node4)
  end
  end
  end.

```

The function `BDDvar_make` in the above definition is used to build the BDD corresponding to a single variable.

We also need replacement of a variable by another variable in a BDD later. This is defined by the function `BDDreplace` as defined below.

Definition 20

```
Definition BDDreplace := [gc:(BDDconfig->(list ad)->BDDconfig);
                        cfg:BDDconfig; ul:(list ad); node,x,y:ad]
Cases (BDDvar_make gc cfg ul y) of (cfg1,nodey) =>
  (BDDsubst gc cfg1 (cons nodey ul) node x nodey)
end.
```

Replacing a vector of variables $lx = (x_1 \dots x_N)$ by another vector of variables $ly = (y_1 \dots y_N)$, i.e. computing $F[x_1 := y_1 \dots x_N := y_N]$, is defined by the following function `BDDreplacel`.

Definition 21

```
Fixpoint BDDreplacel [gc:BDDconfig->(list ad)->BDDconfig;
                    cfg:BDDconfig; ul:(list ad); node:ad; lx,ly:(list BDDvar)]
: BDDconfig*ad :=
Cases lx of
  nil => (cfg,node)
| (cons x lx') =>
  Cases ly of
    nil => (* Error *) (cfg,node)
  | (cons y ly') => Cases (BDDreplacel gc cfg ul node lx' ly') of
    (cfg1,node1) => (BDDreplace gc cfg1 (cons node1 ul) node1 x y)
  end
end
end.
```

The results that we prove about these algorithms are similar to the ones we proved for the previous operations (negation, disjunction, implication, etc.: see Chapter 5.) We show that all these operations preserve the invariants of the configuration, used BDDs from the old configuration are preserved, the nodes returned by the operations are OK. Besides, we show the semantics of these operations in terms of the corresponding operations on boolean functions. The proofs are also along similar lines.

We need some extra assumptions in some of the operations. For example, in `(BDDsubst gc cfg ul node1 x node2)`, we have the assumptions that the (boolean function corresponding to the) BDD represented by `node2` is independent of the variable `x`. In Coq:

```
Hypothesis node2_ind : (bool_fun_independent (bool_fun_of_BDD cfg node2) x).
```

Similarly, in `(BDDreplace gc cfg ul node x y)`, which is defined using `BDDsubst`, we have the assumption that the variables `x` and `y` are distinct. In Coq:

```
Hypothesis xy_neq : (ad_eq x y)=false.
```

Similarly, in `BDDreplacel`, we have the assumption that the variables in the list `lx` are distinct from the corresponding variables in the list `ly`. In Coq:

```
Fixpoint ad_list_neq [l1,l2:(list ad)] : bool :=
Cases l1 l2 of
  nil _ => true
| _ nil => true
```

```

| (cons a1 l1') (cons a2 l2') => (andb (negb (ad_eq a1 a2))
                                     (ad_list_neq l1' l2'))
end.

```

Hypothesis `lxy_neq` : `(ad_list_neq lx ly)=true`.

Here, `andb` is conjunction on `bool`, `negb` is negation on `bool`.

As an example we show the semantics of `BDDreplacel` that we proved. We can see that the hypothesis `(ad_list_neq lx ly)` mentioned above is present as a condition in the theorem.

Lemma 2

```

Lemma BDDreplacel_is_replacel :
  (gc:(BDDconfig->(list ad)->BDDconfig))
  (gc_OK gc)
  ->(cfg:BDDconfig; ul:(list ad); node:ad; lx,ly:(list BDDvar))
  (BDDconfig_OK cfg)
  ->(used_list_OK cfg ul)
  ->(used_node' cfg ul node)
  ->(ad_list_neq lx ly)=true
  ->(bool_fun_eq (bool_fun_of_BDD (Fst (BDDreplacel gc cfg ul node lx ly))
                                (Snd (BDDreplacel gc cfg ul node lx ly))))
    (bool_fun_replacel (bool_fun_of_BDD cfg node) lx ly))

```

Chapter 7

Implementation of μ -calculus

In this chapter we describe how we implemented a symbolic model checker for the μ -calculus using the BDD implementation described in Chapters 5 and 6. We define the formal syntax of μ -calculus formulae (Definition 22) and define the monotonicity condition on the formulae (Definitions 25 and 26), which ensures that the semantics is well defined. We define the semantics of these formulae as boolean expressions (Definition 34.) We face a problem while defining this semantics in the case of the fixpoint operator, since it requires proving simultaneously that the semantics is monotonic, so that we can talk about a least fixpoint. We instead define the semantics of the fixpoint operator in terms of an iterative computation (Definitions 31 and 33), with a bound on the number of iterations. We then show that this semantics is monotonic (Lemma 3,) which allows us to prove that the semantics of the fixpoint operator is actually a least fixed point (Lemma 22.) In the process, we need to define a canonical form for environments for a finite set of variables (Definitions 44 and 45) and show that it has cardinality at most 2^N (Lemma 11,) where N is the cardinality of the set of variables. We then define the algorithm for converting a μ -calculus formula to a BDD (Definition 55) which gives us a model checking algorithm, as mentioned in Chapter 3. This definition is along similar lines as the definition of the semantics of μ -calculus formulae. We then prove that this algorithm is correct with respect to the semantics (Lemma 23.)

7.1 μ -Calculus Formulae

We define the inductive type `mu_form` of μ -calculus formulae as follows. The constructors `mu_ap` and `mu_rel_var` are for atomic propositions and relational variables respectively. Atomic propositions and relational variables are represented by the type `ad`. Transition variables are also represented by the type `ad`. `(mu_all t f)` represents the formula " $[t]f$ ", which is true in those states such that all states reachable in one step from them using the transition represented by `t` satisfy `f`. `(mu_mu P f)` represents the formula " $\mu P.f$ ". The constructors `mu_neg` and `mu_and` represent the connectives \neg and \wedge respectively.

Definition 22

```
Inductive mu_form : Set :=
  mu_ap : ad -> mu_form
| mu_rel_var : ad -> mu_form
| mu_neg : mu_form -> mu_form
| mu_and : mu_form -> mu_form -> mu_form
| mu_all : ad -> mu_form -> mu_form
| mu_mu : ad -> mu_form -> mu_form.
```

The definition is along similar lines as Definition 1. We have different constructors for atomic propositions and relational variables although both of them represent sets of states. This is to distinguish variables which can be bound by the μ operator (relational variables) from those which can not (atomic propositions.) The relational variables occurring free in a μ -calculus formula are defined by the function `mu_rel_free`.

$(\text{mu_rel_free } P \ f)$ returns true if there is a free occurrence of the relational variable P in the formula f , otherwise it returns false.

Definition 23

```
Fixpoint mu_rel_free [P:ad;f:mu_form] : bool :=
Cases f of
  (mu_ap _) => false
| (mu_rel_var Q) => (ad_eq P Q)
| (mu_neg g) => (mu_rel_free P g)
| (mu_and g h) => (orb (mu_rel_free P g) (mu_rel_free P h))
| (mu_all t g) => (mu_rel_free P g)
| (mu_mu Q g) => (andb (negb (ad_eq P Q)) (mu_rel_free P g))
end.
```

Similarly, the function mu_t_free defines the transition variables which occur in a μ -calculus formula.

Definition 24

```
Fixpoint mu_t_free [t:ad;f:mu_form] : bool :=
Cases f of
  (mu_ap _) => false
| (mu_rel_var P) => false
| (mu_neg g) => (mu_t_free t g)
| (mu_and g h) => (orb (mu_t_free t g) (mu_t_free t h))
| (mu_all u g) => (orb (ad_eq t u) (mu_t_free t g))
| (mu_mu Q g) => (mu_t_free t g)
end.
```

As mentioned in Chapter 3, for the semantics of μ -calculus formulae to be well-defined, we need to impose some syntactic restrictions on the formulae. The condition is that if there is a sub-formula of the form $(\text{mu_mu } P \ f)$, then inside the formula f , every occurrence of the variable P should be under an even number of negations.

We inductively define when a variable occurs inside a formula an even or odd number of times, using the predicate f_P_even . $(\text{f_P_even } P \ f \ \text{true})$ means that all free occurrences of the relational variable P in the formula f are under an even number of negations. $(\text{f_P_even } P \ f \ \text{false})$ means that all free occurrences of the relational variable P in the formula f are under an odd number of negations.

Definition 25

```
Inductive f_P_even [P:ad] : mu_form -> bool -> Prop :=
  mu_ap_even : (p:ad)(f_P_even P (mu_ap p) true)
| mu_ap_odd : (p:ad)(f_P_even P (mu_ap p) false)
| mu_rel_var_even : (f_P_even P (mu_rel_var P) true)
| mu_neg_odd : (f:mu_form)(f_P_even P f true) -> (f_P_even P (mu_neg f) false)
| mu_neg_even : (f:mu_form)(f_P_even P f false) -> (f_P_even P (mu_neg f) true)
| mu_and_even : (f,g:mu_form)(f_P_even P f true) -> (f_P_even P g true)
  -> (f_P_even P (mu_and f g) true)
| mu_and_odd : (f,g:mu_form)(f_P_even P f false) -> (f_P_even P g false)
  -> (f_P_even P (mu_and f g) false)
| mu_all_even : (t:ad;f:mu_form)(f_P_even P f true)
  -> (f_P_even P (mu_all t f) true)
| mu_all_odd : (t:ad;f:mu_form)(f_P_even P f false)
  -> (f_P_even P (mu_all t f) false)
| mu_mu_P_even : (f:mu_form)(f_P_even P (mu_mu P f) true)
```

```

| mu_mu_P_odd : (f:mu_form)(f_P_even P (mu_mu P f) false)
| mu_mu_Q_even : (Q:ad;f:mu_form)(ad_eq P Q)=false -> (f_P_even P f true)
                -> (f_P_even P (mu_mu Q f) true)
| mu_mu_Q_odd  : (Q:ad;f:mu_form)(ad_eq P Q)=false -> (f_P_even P f false)
                -> (f_P_even P (mu_mu Q f) false).

```

`mu_ap_even` and `mu_ap_odd` say that P occurs under an even as well as an odd number of negations in an atomic proposition (since actually there are no occurrences.) `mu_rel_var_even` says that P occurs under an even number of negations in P . `mu_neg_odd` (`mu_neg_even`) says that P occurs under odd (even) number of negations in $(\text{mu_neg } f)$ if it occurs under even (odd) number of negations in f . `mu_and_even` (`mu_and_odd`) says that P occurs under even (odd) number of negations in $(\text{mu_and } f \ g)$ if it occurs under even (odd) number of negations in both f and g . `mu_all_even` (`mu_all_odd`) says that P occurs under even (odd) number of negations in $(\text{mu_all } t \ f)$ if it occurs under even (odd) number of negations in f . `mu_mu_P_even` and `mu_mu_P_odd` say that P occurs under even as well as odd number of negations in $(\text{mu_mu } P \ f)$ (since actually there are no free occurrences.) `mu_mu_Q_even` (`mu_mu_Q_odd`) says that P occurs under even (odd) number of negations in $(\text{mu_mu } Q \ f)$ if P is distinct from Q and P occurs under even (odd) number of negations in f .

The important point is that we don't have a clause for the case $(f_P_even \ P \ (\text{mu_rel_var } Q) \ true)$ or $(f_P_even \ P \ (\text{mu_rel_var } Q) \ false)$ when P is not equal to Q . This means that in this case, P occurs neither under an even nor under an odd number of negations in Q .

This allows us to define the required predicate `f_ok`. $(f_ok \ f)$ means that if f has a sub-formula of the form $(\text{mu_mu } P \ g)$ then $(f_P_even \ P \ g \ true)$ holds (meaning that all free occurrences of P in g are under even number of negations.)

Definition 26

```

Inductive f_ok : mu_form -> Prop :=
  mu_ap_f_ok : (p:ad)(f_ok (mu_ap p))
| mu_rel_var_f_ok : (P:ad)(f_ok (mu_rel_var P))
| mu_neg_f_ok : (f:mu_form)(f_ok f)->(f_ok (mu_neg f))
| mu_and_f_ok : (f,g:mu_form)(f_ok f)->(f_ok g)->(f_ok (mu_and f g))
| mu_all_f_ok : (t:ad;f:mu_form)(f_ok f)->(f_ok (mu_all t f))
| mu_mu_f_ok : (P:ad;f:mu_form)(f_ok f)->(f_P_even P f true)
                ->(f_ok (mu_mu P f)).

```

7.2 Semantics of μ -Calculus Formulae

The formulae of μ -calculus are used to represent properties of states in a finite state system. For our purposes we suppose that the states of the finite state system are n -tuples of booleans. Each n -tuple represents a state of the system, so that there are 2^n states in all. Thus, we can denote a subset of this set of states by a boolean expression on n variables. A boolean expression over these n variables would denote the possible assignments to these n variables which make the boolean expression true, and hence would be a subset of the set of 2^n n -tuples.

Transition relations over this set of states can then be represented by boolean expressions over two disjoint sets of n variables. Such a boolean expression would represent a relation consisting of pairs of tuples (t_1, t_2) such that the boolean expression is true in an environment which maps the first set of variables to the tuple t_1 and the second set of variables to the tuple t_2 .

For example, in our case, since boolean variables are represented by the type `ad` (essentially integers,) we represent a set of states by a boolean expression over the variables $0 \dots n - 1$ and relations by boolean expressions over the variables $0 \dots 2n - 1$.

We define the semantics of μ -calculus formulae as boolean expressions. This is done in an environment where every relational variable is mapped to a set of states (here, boolean expression) and every transition variable is mapped to a relation between states (which again, is a boolean expression.) These two environments are of the type `ad->bool_expr`.

Definition 27

Definition `rel_env` := `ad -> bool_expr`.

Definition `trans_env` := `ad-> bool_expr`.

Since the type `bool_expr` contains boolean expressions over all possible (infinite) variables, for the purpose of writing the proofs, we need to separately have the condition that these boolean expressions are over a certain range. For this we have the following predicate `be_ok`, defined w.r.t. a function `vf` from variables to booleans (representing some condition on the variables.) `(be_ok vf be)` means that for any variable `x` (of type `ad`) occurring in the boolean expression `be`, `(vf x)=true` holds.

Definition 28

```
Inductive be_ok [vf:ad->bool] : bool_expr->Prop :=
  zero_ok : (be_ok vf Zero)
| one_ok : (be_ok vf One)
| var_ok : (x:BDDvar)(vf x)=true->(be_ok vf (Var x))
| neg_ok : (be:bool_expr)(be_ok vf be)->(be_ok vf (Neg be))
| or_ok : (be1:bool_expr; be2:bool_expr) (be_ok vf be1)->(be_ok vf be2)
  ->(be_ok vf (Or be1 be2))
| and_ok : (be1:bool_expr; be2:bool_expr) (be_ok vf be1) ->(be_ok vf be2)
  ->(be_ok vf (And be1 be2))
| impl_ok : (be1:bool_expr; be2:bool_expr) (be_ok vf be1) ->(be_ok vf be2)
  ->(be_ok vf (Impl be1 be2))
| iff_ok : (be1:bool_expr; be2:bool_expr) (be_ok vf be1) ->(be_ok vf be2)
  ->(be_ok vf (Iff be1 be2)).
```

Then, to specify the condition that a boolean expression contains only variables over a certain range, we use the following function `var_lu`. `(var_lu L U x)` returns true if the variable `x` is in the range `L .. U-1`.

Definition 29

```
Definition var_lu := [L,U:nat; x:ad]
  (andb (nat_le L (nat_of_ad x)) (nat_le (S (nat_of_ad x)) U)).
```

We also require the condition that the atomic propositions in the μ -calculus formulae are over a certain range. For this purpose, we define the following similar predicate on μ -calculus formulae. `(mu_form_ap_ok vf f)` holds if for all atomic propositions occurring in the formula `f`, `(vf x)=true` holds.

Definition 30

```
Inductive mu_form_ap_ok [vf:ad->bool] : mu_form -> Prop :=
  mu_ap_ok : (p:ad)(vf p)=true -> (mu_form_ap_ok vf (mu_ap p))
| mu_rel_var_ok : (P:ad)(mu_form_ap_ok vf (mu_rel_var P))
| mu_neg_ok : (f:mu_form)(mu_form_ap_ok vf f) -> (mu_form_ap_ok vf (mu_neg f))
| mu_and_ok : (f,g:mu_form)(mu_form_ap_ok vf f) -> (mu_form_ap_ok vf g)
  -> (mu_form_ap_ok vf (mu_and f g))
| mu_all_ok : (t:ad;f:mu_form)(mu_form_ap_ok vf f)
  -> (mu_form_ap_ok vf (mu_all t f))
| mu_mu_ok : (P:ad;f:mu_form)(mu_form_ap_ok vf f)
  -> (mu_form_ap_ok vf (mu_mu P f)).
```

The semantics of μ -calculus formulae is defined by the function `mu_eval` (Definition 34). It takes as argument an environment `te` for transition variables, a μ -calculus formula `f`, an environment `re` for relational variables, and returns a boolean expression denoting the set of states at which the formula holds in the given

environments. The main difficulty here is in defining the semantics of the fixpoint operator. Normally, the formula $(\mu_{\mu} P f)$, in an environment e mapping a relational variable to a set of states, is defined to represent the least set of states s such that f represents the same set of states s in the environment $e[P := s]$ (where $e[P := s]$ denotes augmentation of the environment e by binding P to s .) However, to be able to define the semantics of μ -calculus formulae in this way, we would first need to prove that such a least set exists. The only option would be to simultaneously define the semantics of μ -calculus formulae and a proof that the existence of least fixpoints for such a semantics. This would involve numerous complications, having to simultaneously prove that the semantics we define is in way monotonic (using the fact that the μ -calculus formulae contain atomic propositions over a finite range, and the semantics returns a boolean expression over the same finite range, etc.)

Instead, we define the semantics of the fixpoint operator as an iterative computation. We separately prove that such an iterative computation yields a least fixpoint. We use the following function `iter` which applies a function f , of some type $A \rightarrow A$, at most n times on an argument a of type A . It stops if a fixpoint is reached earlier (found by the input function `A_eq` for testing equality between elements of A .)

Definition 31

```
Fixpoint iter [A:Type;A_eq:A->A->bool;f:A->A;a:A;n:nat] : A :=
  Cases n of
  0 => a
  | (S m) => if (A_eq a (f a)) then (f a) else (iter ? A_eq f (f a) m)
end.
```

We have to give a bound here on the number of iterations, since it is necessary for guaranteeing termination. The function takes as argument a function for deciding equality between elements of type A . In the case of boolean expressions, the equality `be_eq` we define is the same as propositional equivalence. This is decidable because we have a tautology checker using BDDs to decide the validity of a boolean expression. Thus we have the following function `be_eq_dec` for deciding equality of two boolean expressions, which we use for iterative application of a function of type `bool_expr->expr`.

The tautology checker also takes in as argument a garbage collector function. For that, we simply use the function which does nothing (returns the input configuration.)

Definition 32

```
Definition be_eq_dec := [be1,be2:bool_expr]
  (is_tauto [x:BDDconfig; _:(list ad)]x (Iff be1 be2)).
```

Definition 33

```
Definition be_iter := [f:(bool_expr->bool_expr); be:bool_expr; n:nat]
  (iter bool_expr be_eq_dec f be n).
```

Defining the semantics of the fixpoint operator as an iterative computation has the additional advantage that the algorithm for converting μ -calculus formulae to BDDs (Definition 55) is also defined in a similar way, so it becomes easier to prove the correctness of this algorithm with respect to this semantics. It allows us to postpone talking about fixpoints. Any other definition of the semantics, involving fixpoints, would have made the proof of correctness of this algorithm much more complicated.

The following function `mu_eval` defines the semantics of a μ -calculus formula f as a boolean expression, in environments `te` and `re`, where `te` maps every transition variable (of type `ad`) to boolean expressions (representing a transition relation), and `re` maps every relational variable (of type `ad`) to a boolean expression (representing a set of states.) The input N is the number of variables we are working with (i.e. the atomic propositions in the μ -calculus formulae range from $0 \dots N - 1$). As said above, the fixpoint operator is interpreted as an iterative computation of fixpoint, starting from the boolean expression `Zero`. We provide a bound of 2^N on the number of iterations. Of course, it needs to be proven that we can obtain the required fixpoint in this many iterations, provided the input is well formed.

Definition 34

```
Fixpoint mu_eval [N:nat;te:trans_env;f:mu_form] : rel_env -> bool_expr :=
[re:rel_env]
Cases f of
  (mu_ap p) => (Var p)
| (mu_rel_var P) => (re P)
| (mu_neg g) => (Neg (mu_eval N te g re))
| (mu_and g h) => (And (mu_eval N te g re) (mu_eval N te h re))
| (mu_all t g) => (mu_all_eval N (te t) (mu_eval N te g re))
| (mu_mu P g) =>
  (iter ? be_eq_dec [be](mu_eval N te g (re_put re P be)) Zero (two_power N))
end.
```

The function `re_put` in the above definition is used to change the boolean expression to which a variable is mapped in an environment.

In the above definition we have interpreted the formula `(mu_all t g)` using the function `mu_all_eval` which is defined as follows. $(lx \ N)$ is the vector of variables $0 \dots N - 1$ and $(lx' \ N)$ is the vector of variables $N \dots 2N - 1$ (we represent vectors of variables as lists.) `ad_of_nat` converts natural numbers (of type `nat`) to elements of type `ad`.

Definition 35

```
Definition ap := [n:nat](ad_of_nat n).
Definition ap' := [N,n:nat](ad_of_nat (plus N n)).
```

```
Fixpoint lx_1 [n:nat] : (list ad) :=
Cases n of
  0 => (nil ?)
| (S m) => (cons (ap m) (lx_1 m))
end.
```

```
Fixpoint lx'_1 [N,n:nat] : (list ad) :=
Cases n of
  0 => (nil ad)
| (S m) => (cons (ap' m) (lx'_1 m))
end.
```

```
Definition lx := [N:nat](lx_1 N).
Definition lx' := [N:nat](lx'_1 N).
```

```
Definition mu_all_eval : nat -> bool_expr -> bool_expr -> bool_expr :=
[N;t,be](univ1 (Impl t (replacel be (lx N) (lx' N))) (lx' N)).
```

The corresponding operation on boolean functions is also defined similarly.

Definition 36

```
Definition bool_fun_mu_all : nat -> bool_fun -> bool_fun -> bool_fun :=
[N;bft,bfg]
(bool_fun_univ1 (bool_fun_impl bft (bool_fun_replacel bfg (lx N) (lx' N))) (lx' N)).
```

To be able to prove that the semantics of the fixpoint operator is as expected, we first need to show that `mu_eval` is monotonic. This is stated in Lemma 3 below. We first require a few definitions.

Definition 37 A function abe from ad to $bool_expr$ is OK with respect to a function vf from ad to $bool$ if every boolean expression be in the range of abe is OK (i.e. $(be_ok\ vf\ be)$ holds.)

```
Definition ad_to_be_ok := [vf:(ad->bool); abe:(ad->bool_expr)]
(x:ad)(be_ok vf (abe x)).
```

The above predicate is used for defining the appropriate conditions on the environments.

Definition 38 Two functions $f1$ and $f2$ from ad to $bool_expr$ are equal if for every address x , $(f1\ x)$ is equivalent to $(f2\ x)$.

```
Definition ad_to_be_eq := [f1,f2:(ad->bool_expr)](x:ad)(be_eq (f1 x) (f2 x)).
```

The relation be_le used below is a pre-order on boolean expressions. $(be_le\ be1\ be2)$ means that any environment satisfying the boolean expression $be1$ also satisfies the boolean expression $be2$.

Definition 39 A function f from relational environments (of type rel_env) to boolean expressions is said to be monotonically increasing with respect to the variable P if for any boolean expressions $be1$ and $be2$ such that $be1$ is less than $be2$, and under any environment re , the boolean expression to which f evaluates when P is bound to $be1$ in re , is less than the boolean expression to which f evaluates when P is bound to $be2$ in re .

```
Definition re_to_be_inc := [f:(rel_env->bool_expr); P:ad]
(re:rel_env; be1,be2:bool_expr)(be_le be1 be2)
->(be_le (f (re_put re P be1)) (f (re_put re P be2))).
```

We have a similar definition for monotonically decreasing functions from rel_env to $bool_expr$.

Definition 40

```
Definition re_to_be_dec := [f:(rel_env->bool_expr); P:ad]
(re:rel_env; be1,be2:bool_expr)(be_le be1 be2)
->(be_le (f (re_put re P be2)) (f (re_put re P be1))).
```

We now prove the lemma which says that the semantics of μ -calculus formulae is monotonic. We prove that if the variable P occurs under even number of negations in the formula f then the semantics is monotonically increasing with respect to P , and if P occurs under odd number of negations, then the semantics is monotonically decreasing with respect to P . We also simultaneously show that the boolean expression returned by the semantics is OK (as defined by the be_ok predicate.) We also prove that if two relational environments are equivalent (as defined by the predicate $ad_to_be_eq$), then the boolean expressions returned under those environments are equivalent. The hypotheses are that the formula is OK (as defined by f_ok) and only contains atomic propositions in the range $0 .. N - 1$. Also, the environments are assumed to be OK (as defined by the predicate $ad_to_be_ok$.)

Lemma 3

```
Lemma mu_eval_lemma1 : (N:nat; te:trans_env)
(ad_to_be_ok (var_lu (0) (mult (2) N)) te)
->(f:mu_form)(f_ok f)
->(mu_form_ap_ok (var_lu (0) N) f)
->((re:rel_env)(ad_to_be_ok (var_lu (0) N) re)
->(be_ok (var_lu (0) N) (mu_eval N te f re)))
/\((P:ad)(f_P_even P f true)->(re_to_be_inc (mu_eval N te f) P))
/\((P:ad)(f_P_even P f false)->(re_to_be_dec (mu_eval N te f) P))
/\((re1,re2:rel_env)(ad_to_be_eq re1 re2)
->(be_eq (mu_eval N te f re1) (mu_eval N te f re2))).
```

Proof: The proof is by induction on the formula f . The difficult case is when f is of the form $(\mu_{\mu} a g)$ for some relational variable a and μ -calculus formula g . We make use of the following results about be_iter .

Lemma 4 *If $P(\text{be})$ holds and $P(\text{be}') \Rightarrow P(f(\text{be}'))$ for every be' , then P holds on every iterate $f^n(\text{be})$.*

```
Lemma be_iter_prop_preserved :
  (n:nat;be:bool_expr;f:bool_expr->bool_expr;pr:bool_expr->Prop)
  (pr be)
  -> ((be':bool_expr)(pr be')->(pr (f be')))
  -> (pr (be_iter f be n)).
```

Lemma 5 *If f_1 and f_2 are compatible w.r.t. be_eq and compute logically equivalent functions, then so do f_1^n and f_2^n .*

```
Lemma be_iter_eq_preserved_1 :
  (n:nat;be1,be2:bool_expr;f1,f2:bool_expr->bool_expr)
  (be_eq be1 be2)
  ->((be1',be2':bool_expr)(be_eq be1' be2')->(be_eq (f1 be1') (f1 be2')))
  ->((be1',be2':bool_expr)(be_eq be1' be2')->(be_eq (f2 be1') (f2 be2')))
  ->((be1',be2':bool_expr)(be_eq be1' be2')->(be_eq (f1 be1') (f2 be2')))
  -> (be_eq (be_iter f1 be1 n) (be_iter f2 be2 n)).
```

Lemma 6

```
Lemma be_iter_le_preserved :
  (n:nat;be1,be2:bool_expr;f1,f2:bool_expr->bool_expr)
  (be_le be1 be2)
  ->((be1',be2':bool_expr)(be_eq be1' be2')->(be_eq (f1 be1') (f1 be2')))
  ->((be1',be2':bool_expr)(be_eq be1' be2')->(be_eq (f2 be1') (f2 be2')))
  -> ((be1',be2':bool_expr)(be_le be1' be2')->(be_le (f1 be1') (f2 be2')))
  -> (be_le (be_iter f1 be1 n) (be_iter f2 be2 n)).
```

We require the fact the operator And is monotonically increasing and Neg is monotonically decreasing. Also we need to show that mu_all_eval is monotonically increasing, which involves showing that forall , univl , subst , replace , replacel , impl , etc. are monotonically increasing. We also need to talk about the variables occurring in the input and output of these boolean expressions. For example, we show that if a variable occurs in the boolean expression $(\text{forall } \text{be } x)$, then it occurs in the boolean expression be and is distinct from x . We prove similar results about univl , subst , replace , replacel etc.

All this allows us to complete the proof of Lemma 3. \square

7.3 Monotonic Functions on Boolean Expressions

We need to show that the semantics of the fixpoint operator as an iterative computation, as defined by the function mu_eval , corresponds to the usual semantics as least fixpoints.

Fixpoints are defined by the following relation fp . $(\text{fp } \text{bef } \text{be})$ means that the boolean expression be is equivalent to the boolean expression $(\text{bef } \text{be})$, where bef is a function from boolean expressions to boolean expressions.

Definition 41

```
Definition fp := [bef:(bool_expr->bool_expr); be:bool_expr]
  (be_eq be (bef be)).
```

Least fixpoints are defined by the relation lfp . $(\text{lfp } \text{bef } \text{be})$ means that be is a fixpoint of the function bef and is smaller than every fixpoint of bef .

Definition 42

```
Definition lfp := [bef:(bool_expr->bool_expr); be:bool_expr]
  (fp bef be)/\((be':bool_expr)(fp bef be')->(be_le be be')).
```

We also have the following definition `lfp_be` to define the least fixpoint greater than a certain value.

Definition 43

```
Definition lfp_be := [bef:(bool_expr->bool_expr); be1,be:bool_expr]
  (fp bef be)
  /\(be_le be1 be)
  /\((be':bool_expr)(fp bef be')->(be_le be1 be')->(be_le be be')).
```

Lemma 7

```
Lemma lfp_be_lfp : (bef:(bool_expr->bool_expr); be:bool_expr)
  (lfp bef be)<->(lfp_be bef Zero be).
```

We use this characterization of fixpoints to show that the semantics of the fixpoint operator defined by `mu_eval` is equivalent to the semantics in terms of least fixpoints. The main result that needs to be proven is that if we take any monotonic function on boolean expressions then we can obtain its least fixpoint by iterating this function atmost 2^N times and stopping earlier if a fixpoint is reached (in other words, by using the function `be_iter`.) Here, N is the cardinality of the set of variables which occur in the boolean expressions. Intuitively, the argument is that if a boolean function `be1` is less than the boolean expression `be2` (as defined by `be_le`,) then the set of N -tuples of booleans which satisfy `be2` is bigger than the set of N -tuples which satisfy `be1`. Also, there are only 2^N such N -tuples, hence we must reach a fixpoint in 2^N iterations.

For talking about finite sets, we use the library of *Ensembles* which is a part of the standard libraries of Coq. This provides several definitions and results related to cardinalities of finite sets.

The environments for boolean variables that we had been using till now were total functions from the (infinite) set of variables to booleans. However, this is an infinite set, and for our purposes, we need an exact characterization of those environments which map a given finite set of variables to booleans. One option is to use the same environments as above, but enforce the condition that they are constant over all except that finite set of variables. However, this again leads to problems because of the fact that functions in Coq are not extensional. We instead represent these environments using maps. An environment is represented as a map from addresses to the `unit` type. Such a map is supposed to represent an environment which maps every variable in its domain to `true` and the other variables to `false`. Essentially, this map represents the finite set of variables which are supposed to be mapped to `true`. The advantage with this representation is that there is a characterization of maps which are canonical. Thus if two such maps represent the same environment, then they are equal according to Coq's equality.

`var_env''` is the type of such environments. The exact characterization of canonical environments for variables in a certain range is given by `Evar_env''`. (`Evar_env'' L U`) is the ensemble which contains all those environments (represented by maps) which are canonical and whose domains only contain variables which are greater than or equal to `L`, and strictly less than `U`. `mapcanon` is the predicate characterizing maps which are canonical.

Definition 44

```
Definition var_env'' := (Map unit).
```

Definition 45

```
Definition Evar_env'' := [L,U:nat]
  [ve:var_env'']
```

```
(mapcanon unit ve)
/\((x:ad)(var_lu L U x)=false->(in_dom unit x ve)=false).
```

```
Evar_env'' : nat -> nat -> (Ensemble var_env'').
```

As mentioned in Chapter 4, (Ensemble var_env'') is just a predicate on var_env''.

The (total) environment represented by such a map is defined by the following function var_env''_to_env.

Definition 46

```
Definition var_env''_to_env := [ve:var_env'';x:ad](in_dom unit x ve).
```

```
var_env''_to_env : var_env'' -> var_env.
```

We first need to prove that the cardinality of the ensemble defined above is at most 2^{U-L} .

We first show that if (minus U L)=(0) then this ensemble has cardinality 1.

Lemma 8

```
Lemma var_env''_cardinal_one : (L,U:nat)(minus U L)=(0)->
(cardinal var_env'' (Evar_env'' L U) (1)).
```

Proof: This requires us to show that the ensemble is exactly a singleton containing the empty map. The first part is to show that the empty map is present in this ensemble. This is because of the fact that the empty map is canonical and does not have any variable in its domain. Next we assume that some map *ve* is present in this ensemble and show that it is same as the empty map (i.e. they are equal according to Coq's equality.) This is because U-L is equal to zero, and hence the domain of *ve* is empty. To prove that it is equal to the empty map (M0 unit), we use the fact that *ve* is canonical (by definition of Evar_env''). As mentioned, if two canonical maps denote the same function, then they are equal. \square

Next, we need to show that if we increase the range U-L by one, then the cardinality of the corresponding ensemble gets doubled. For convenience, we name these two ensembles as Evar_env'' LU and Evar_env'' LSU.

Definition 47

```
Definition Evar_env''LU : (Ensemble var_env'') := [L,U:nat](Evar_env'' L U).
```

```
Definition Evar_env''LSU : (Ensemble var_env''):= [L,U:nat](Evar_env'' L (S U)).
```

We define two functions *f1'* and *f2'*, which are supposed to map environments in the ensemble Evar_env''LU to environments in the ensemble Evar_env''LSU. That however would need to be proven. *imagef1'* and *imagef2'* are the corresponding images of these two functions. Intuitively, *f1'* takes any environment and makes the variable number U true. This is done using the function Evar_env''ntoSn. *f2'* is the identity function.

Definition 48

```
Definition Evar_env''ntoSn : nat->var_env''->var_env'' :=
[U:nat; ve:var_env''](MapPut ? ve (ad_of_nat U) tt).
```

```
Definition f1' : nat->var_env''->var_env'' := [U:nat](Evar_env''ntoSn U).
```

```
Definition f2' = var_env''->var_env'' := [x:var_env'']x.
```

```
Definition imagef1' : (_,_:nat)(Ensemble var_env'') :=
[L,U:nat](Im ?? (Evar_env''LU L U) (f1' U)).
```

```
Definition imagef2' : (_,_:nat)(Ensemble var_env'') :=
[L,U:nat](Im ?? (Evar_env''LU L U) f2').
```

Next we show that $\text{Evar_env}'\text{'LSU}$ is a subset of the union of the ensembles $\text{imagef1}'$ and $\text{imagef2}'$.

Lemma 9

Definition $\text{imagef1}'\text{orf2}' := [L,U:\text{nat}] (\text{Union } ? (\text{imagef1}' L U) (\text{imagef2}' L U))$.

Lemma $\text{imagef1}'\text{orf2}'\text{lemma} : (L,U:\text{nat}; \text{ve}:\text{var_env}'')$
 $(\text{In } ? (\text{Evar_env}'\text{'LSU } L U) \text{ve})$
 $\rightarrow (\text{In } ? (\text{imagef1}'\text{orf2}' L U) \text{ve})$.

Proof: We take any environment ve belonging to the ensemble $\text{Evar_env}'\text{'LSU}$. In case it maps the variable U to `false`, then it clearly belongs to the ensemble $\text{Evar_env}'\text{'LU}$ and hence belongs to $\text{imagef2}'$, since $\text{f2}'$ is the identity function. The conditions related to canonicity are automatically satisfied.

In case ve maps the variable U to `true` then we consider the map ve' equal to $(\text{MapRemove } ? \text{ve } (\text{ad_of_nat } U))$. ve' is the same as ve , except that the variable U is mapped to `false`. ve' is also canonical since MapRemove preserves canonicity. Thus ve' belongs to the ensemble $\text{Evar_env}'\text{'LU}$. It remains to show that ve is equal to the image of ve' under the function $\text{f1}'$. This is because, firstly, they represent the same environment (from the semantics of MapPut and MapRemove ,) and secondly, both of them are canonical (since MapPut and MapRemove preserve canonicity.) If two canonical maps represent the same function then they are equal. \square

This allows us to show that if the cardinality of $\text{Evar_env}'\text{'LU}$ is n then $\text{Evar_env}'\text{'LSU}$ is finite and has cardinality less than $(\text{mult } (2) n)$.

Lemma 10

Lemma $\text{Evar_env}'\text{'LSU_finite} : (L,U,n:\text{nat})$
 $(\text{cardinal } \text{var_env}'\text{' } (\text{Evar_env}'\text{'LU } L U) n)$
 $\rightarrow (\text{Finite } \text{var_env}'\text{' } (\text{Evar_env}'\text{'LSU } L U))$.

Lemma $\text{card_Evar_env}'\text{'LSU_lemma} : (L,U,n:\text{nat})$
 $(\text{cardinal } \text{var_env}'\text{' } (\text{Evar_env}'\text{'LU } L U) n)$
 $\rightarrow (m:\text{nat})(\text{cardinal } \text{var_env}'\text{' } (\text{Evar_env}'\text{'LSU } L U) m)$
 $\rightarrow (1 \leq m (\text{mult } (2) n))$.

Proof: We first use the property of images to derive that $\text{imagef1}'$ and $\text{imagef2}'$ are finite and have cardinalities less than n . We prove an auxiliary result that if the cardinalities of two ensembles A and B are c_1 and c_2 respectively, then their union is finite and has cardinality less than or equal to $c_1 + c_2$. From this we derive that $\text{imagef1}'\text{orf2}'$ is finite and has cardinality less than or equal to $(\text{mult } (2) n)$. We then derive the required results from the fact that $\text{Evar_env}'\text{'LSU}$ is a subset of $\text{imagef1}'\text{orf2}'$. \square

Finally, we prove that the cardinality of the ensemble $(\text{Evar_env}'\text{' } L U)$ is less than or equal to 2^{U-L} . We do not prove the other side of the inequality since it is not required.

Lemma 11

Lemma $\text{Eenv_var}'\text{'LU_finite} :$
 $(n,L,U:\text{nat})(n=(\text{minus } U L)) \rightarrow (\text{Finite } ? (\text{Evar_env}'\text{'LU } L U))$.

Lemma $\text{Eenv_var}'\text{'LU_card} :$
 $(n,L,U,c:\text{nat})(n=(\text{minus } U L)) \rightarrow (\text{cardinal } ? (\text{Evar_env}'\text{'LU } L U) c)$
 $\rightarrow (1 \leq c (\text{two_power } n))$.

Proof: The proof follows in a straightforward manner by induction on n using Lemma 8 and Lemma 10. \square

We then define the ensemble of environments represented by a boolean expression. This is defined with respect to the range $L .. U - 1$ over which the variables of the boolean expression range. This ensemble contains all those environments which firstly are in the ensemble $(\text{Evar_env}'' L U)$ and secondly satisfy the boolean expression.

Definition 49

```
Definition bool_expr_to_var_env'' : nat->nat->bool_expr->(Ensemble var_env'') :=
  [L,U:nat; be:bool_expr; ve:var_env'']
  (eval_be' be (var_env''_to_env' ve))=true
  /\(In var_env'' (Evar_env'' L U) ve).
```

The ensemble of environments corresponding to a boolean expression has cardinality less than 2^{U-L} .

Lemma 12

```
Lemma bool_expr_to_var_env''_finite : (L,U:nat;be:bool_expr)
  (be_ok (var_lu L U) be)
  -> (Finite ? (bool_expr_to_var_env'' L U be)).
```

```
Lemma bool_expr_to_var_env''_card : (L,U,n:nat;be:bool_expr)
  (be_ok (var_lu L U) be)
  -> (cardinal ? (bool_expr_to_var_env'' L U be) n)
  -> (le n (two_power (minus U L))).
```

Proof: This follows from the fact that by definition, this ensemble is a subset of the ensemble $(\text{Evar_env}'' L U)$ whose cardinality is less than 2^{U-L} . \square

We define a new pre-order on boolean expressions using finite environments. $(\text{be_le1 } L U \text{ be1 } \text{be2})$ means that the ensemble $(\text{bool_expr_to_var_env}'' L U \text{ be1})$ is a subset of the ensemble $(\text{bool_expr_to_var_env}'' L U \text{ be2})$.

Definition 50

```
Definition be_le1 := [L,U:nat;be1,be2:bool_expr]
  (ve:var_env'')(In ? (bool_expr_to_var_env'' L U be1) ve)
  -> (In ? (bool_expr_to_var_env'' L U be2) ve).
```

We then need to show that this new order is equivalent to the order be_le . For this, we first need the following auxiliary lemma. It says that for every total environment ve (of type $\text{var_env}'$) there is a finite environment ve'' of type $\text{var_env}''$, which belongs to the ensemble $(\text{Evar_env}'' L U)$ and is equivalent to ve as far as variables in the range $L .. U-1$ are concerned.

Lemma 13

```
Lemma var_env'_to_var_env''_lemma1 : (n,L,U:nat;ve:var_env')
  n=(minus U L)
  -> (EX ve'':var_env'' |(In ? (Evar_env'' L U) ve'')
    /\ (x:ad)(var_lu L U x)=true
    ->(in_dom ? x ve'')=(ve (nat_of_ad x))).
```

Proof: The proof is by induction on n (equal to $(\text{minus } U L)$.) In the case where n is zero, the required ve'' is the empty map. In the case where n is equal to $(S n0)$, we have by induction hypothesis a map $\text{ve1}''$ which belongs to $(\text{Evar_env}'' L (\text{plus } L n0))$ and which is equivalent to ve for variables in the range $L .. L + n0 - 1$. In case the environment ve maps the variable $L + n0$ to false , the required map is $(\text{MapRemove } ? \text{ve1}'' (\text{ad_of_nat } (\text{plus } L n0)))$. In case ve maps the variable $L + n0$ to true , the

required maps is $(\text{MapPut } ? \text{ ve1}'' (\text{ad_of_nat } (\text{plus } L \text{ n0})))$. We require the facts that MapPut and MapRemove preserve canonicity of maps. \square

We also require the fact that the value of a boolean expression with variables in a certain range, is independent of the values to which the variables outside this range are mapped. This is stated in the following lemma. $(\text{seq_eq } L \ U \ ? \ (\text{eq } ?) \ \text{ve1} \ \text{ve2})$ means that the total environments ve1 and ve2 are equivalent as far as variables in the range $L .. U - 1$ are concerned.

Lemma 14

```
Lemma eval_be_independent : (L,U:nat; ve1,ve2:var_env')
  (seq_eq L U ? (eq ?) ve1 ve2)
  ->(be:bool_expr) (be_ok (var_lu L U) be)->(eval_be' be ve1)=(eval_be' be ve2).
```

Proof: The proof is straightforward using induction on the boolean expression be . \square

We next show the equivalence between be_le and be_le1 . The first part is straightforward:

Lemma 15

```
Lemma be_le_le1 : (L,U:nat;be1,be2:bool_expr)(be_le be1 be2) -> (be_le1 L U be1 be2).
```

The second part is to show the converse: $(\text{be_le } \text{be1} \ \text{be2})$ implies $(\text{be_le1 } \text{be1} \ \text{be2})$. Proving this requires the condition that the boolean expressions only contain variables in the range $L .. U - 1$.

Lemma 16

```
Lemma be_le1_le : (L,U:nat;be1,be2:bool_expr)
  (be_ok (var_lu L U) be1)
  -> (be_ok (var_lu L U) be2)
  -> (be_le1 L U be1 be2)
  -> (be_le be1 be2).
```

Proof: We take any total environment ve satisfying the boolean expression be1 . We need to show that ve also satisfies be2 . For this we first convert ve to a finite environment ve'' in the range $L .. U - 1$ using Lemma 13. By using Lemma 14, we get that ve'' also satisfies be1 and hence satisfies be2 , by definition of be_le1 . Then we use Lemma 14 again to show that ve satisfies be2 . \square

We next show that an increasing sequence of boolean expressions terminates. We first prove the following lemma about a sequence of decreasing (with respect to inclusion) ensembles with finite cardinalities. $(\text{Included } ? \ A \ B)$ means that the ensemble A is a subset of the ensemble B .

Lemma 17 *For any sequence f of decreasing (with respect to inclusion) ensembles with finite cardinalities, there exists a m less than or equal to the cardinality of $(f \ 0)$ such that cardinality of $(f \ m)$ is equal to the cardinality of $(f \ (S \ m))$.*

```
Lemma decreasing_ens_seq : (U:Type)
  (n:nat)
  (f:nat->(Ensemble U))
  (cardinal ? (f 0) n)
  -> ((k:nat)(Finite ? (f k)))
  -> ((k:nat)(Included ? (f (S k)) (f k)))
  -> (EX m:nat | (EX c:nat | (le m n) /\ (cardinal ? (f m) c)
    /\ (cardinal ? (f (S m)) c))).
```

Proof: The proof uses well founded induction on n (the cardinality of $(f \ 0)$.) If the cardinality of $(f \ 0)$ is equal to the cardinality of $(f \ (1))$ then the required m is 0 . Otherwise clearly $(f \ (1))$ has cardinality strictly less than the cardinality of $(f \ 0)$, because the sequence of ensembles is decreasing. We then apply the induction hypothesis on the sequence starting from 1 (i.e. excluding $(f \ 0)$.) \square

From this, we derive the lemma about decreasing (with respect to `be_le`) sequence of boolean expressions.

Lemma 18 *For any sequence f of decreasing (with respect to `be_le`) boolean expressions with variables in the range $L .. U-1$, there exists an m less than or equal to 2^{U-L} , such that $(f \ m)$ is equivalent to $(f \ (S \ m))$ (with respect to the relation `be_eq`.)*

```
Lemma decreasing_be_seq_1 : (L,U:nat)
  (f:nat->bool_expr)
  ((k:nat)(be_ok (var_lu L U) (f k)))
-> ((k:nat)(be_le (f (S k)) (f k)))
-> (EX m:nat|(le m (two_power (minus U L))) /\ (be_eq (f m) (f (S m))))).
```

Proof: The sequence of corresponding ensembles is clearly decreasing (with respect to inclusion.) Also the ensemble corresponding to $(f \ 0)$ has cardinality less than 2^{U-L} . Therefore, from Lemma 17, there exists an m less than or equal to 2^{U-L} such that the ensembles corresponding to $(f \ m)$ and $(f \ (S \ m))$ have the same cardinalities. From this, we derive the required result. We use the correspondence between `be_le` and `be_le1` in this proof. \square

We also get the corresponding lemma for increasing sequence of boolean expressions.

Lemma 19

```
Lemma increasing_be_seq_1 : (L,U:nat)
  (f:nat->bool_expr)
  ((k:nat)(be_ok (var_lu L U) (f k)))
-> ((k:nat)(be_le (f k) (f (S k))))
-> (EX m:nat|(le m (two_power (minus U L))) /\ (be_eq (f m) (f (S m))))).
```

Proof: We apply Lemma 18 on the sequence of negations of the boolean expressions in the given sequence. \square

We define the following function `be_iter1` which is composition of a function on boolean expressions, a given number of times. It is different from `be_iter` in that it does not check if a fixpoint is reached earlier.

Definition 51

```
Fixpoint be_iter1 [bef:bool_expr->bool_expr;be:bool_expr;n:nat] : bool_expr :=
Cases n of
  0 => be
| (S m) => (be_iter1 bef (bef be) m)
end.
```

We have the following lemma `be_iter1_fix_ex` which says that if we take a monotonically increasing function on boolean expressions, then we can obtain a fixpoint by composing that function atmost 2^N times. The assumption is that the variables in the boolean expressions range from $0 .. N-1$, that `bef` returns boolean expressions with variables in the same range. `(be_to_be_inc bef)` means that the function `bef` is monotonically increasing.

Definition 52

```
Definition be_to_be_inc := [f:(bool_expr->bool_expr)]
  (be1,be2:bool_expr)(be_le be1 be2)->(be_le (f be1) (f be2)).
```

Lemma 20

```
Lemma be_iter1_fix_ex : (N:nat; bef:(bool_expr->bool_expr))
  (be_to_be_inc bef)
  ->((be:bool_expr)(be_ok (var_lu (0) N) be)->(be_ok (var_lu (0) N) (bef be)))
  ->(be:bool_expr)
    (be_ok (var_lu (0) N) be)
    ->(be_le be (bef be))
    ->(EX m:nat | (le m (two_power N))
      /\(be_eq (be_iter1 bef be m) (be_iter1 bef be (S m)))).
```

Proof: We apply Lemma 19 on the sequence of boolean expressions s defined as $s_k = (\text{be_iter1 } \text{bef } \text{be } k)$ which is a sequence boolean expressions since bef is monotonic. We first require to prove by induction that the variables in all the boolean expressions in this sequence range between $0 \dots N - 1$ (straightforward, using the corresponding property of bef .) \square

We next show when does be_iter (and not be_iter1) return a least fixpoint of a monotonic function.

Lemma 21

```
Lemma be_iter_is_lfp_be
  : (bef:(bool_expr->bool_expr); n:nat; be:bool_expr)
    (be_le be (bef be))
    ->((be1,be2:bool_expr)(be_eq be1 be2)->(be_eq (bef be1) (bef be2)))
    ->((be1,be2:bool_expr)(be_le be1 be2)->(be_le (bef be1) (bef be2)))
    ->(EX m:nat |(le m n)/\ (be_eq (be_iter1 bef be m) (be_iter1 bef be (S m))))
    ->(lfp_be bef be (be_iter bef be n))
```

Proof: The proof is using induction on n . \square

Finally, we show that the semantics of the fixpoint operator is a least fixpoint.

Lemma 22

```
Lemma mu_eval_mu_is_lfp : (N:nat; te:trans_env)
  (ad_to_be_ok (var_lu (0) (mult (2) N)) te)
  ->(P:ad; f:mu_form; re:rel_env)
    (f_ok (mu_mu P f))
    ->(mu_form_ap_ok (var_lu (0) N) (mu_mu P f))
    ->(ad_to_be_ok (var_lu (0) N) re)
    ->(lfp [be:bool_expr](mu_eval N te f (re_put re P be))
      (mu_eval N te (mu_mu P f) re)).
```

Proof: We apply Lemma 21 with the variable bef bound to the function $[\text{be:bool_expr}](\text{mu_eval } N \text{ te } f \text{ (re_put re P be)})$. This function is shown to satisfy the assumptions in Lemma 21, namely that it is monotonic and preserves equality, using Lemma 3. We also use Lemma 20 to show that be_iter1 returns a fixpoint of this function. \square

7.4 Conversion from μ -Calculus Formulae to BDDs

In this section we describe the algorithm for converting a μ -calculus formula to BDDs, and prove the correctness of this algorithm with respect to the semantics of μ -calculus formulae.

The algorithm is along similar lines as the definition of the semantics of μ -calculus formulae. Similar to be_iter , we have the following function BDDiter to iterate a function on BDDs a certain number of times, and stopping earlier if a fixpoint is reached (this is checked by checking equality of nodes.) In the input, we

have a function f which takes a configuration and a node and returns a new pair of a configuration and a node.

Definition 53

```

Fixpoint BDDiter [f:BDDconfig->ad->BDDconfig*ad;cfg:BDDconfig;node:ad;n:nat]
  : BDDconfig * ad :=
Cases n of
  0 => (cfg,node)
| (S m) => Cases (f cfg node) of (cfg1,node1) =>
      if (ad_eq node node1) then (cfg1,node1) else
      (BDDiter f cfg1 node1 m)
      end
end.

```

Similar to μ_eval the algorithm for converting a μ -calculus formula to BDDs uses environments for the relational and transition variables. These are of type $(Map\ ad)$ and map a (relational or transition) variable to an address (representing a BDD node.)

Definition 54

```

Definition Brel_env := (Map ad).
Definition Btrans_env := (Map ad).

```

The following function $BDDmu_eval$ computes the BDD corresponding to a μ -calculus formula. For evaluating the BDD for $(\mu_mu\ P\ f)$ we call $BDDiter$ which iterates the function $BDDmu_eval$ 2^N times.

Definition 55

```

Variable N:nat.
Variable gc:BDDconfig->(list ad)->BDDconfig.
Variable te:Btrans_env.

Fixpoint BDDmu_eval [f:mu_form]
  : BDDconfig -> (list ad) -> Brel_env -> BDDconfig*ad :=
[cfg:BDDconfig;ul:(list ad);bre:Brel_env]
Cases f of
  (mu_ap p) => (BDDvar_make gc cfg ul p)
| (mu_rel_var P) => Cases (MapGet ? bre P) of
      NONE => (cfg,BDDzero)
      | (SOME node) => (cfg,node)
      end
| (mu_neg g) => Cases (BDDmu_eval g cfg ul bre) of (cfgg,nodeg) =>
      (BDDneg gc cfgg (cons nodeg ul) nodeg)
      end
| (mu_and g h) => Cases (BDDmu_eval g cfg ul bre) of (cfgg,nodeg) =>
      Cases (BDDmu_eval h cfgg (cons nodeg ul) bre) of (cfgh,nodeh) =>
      (BDDand gc cfgh (cons nodeh (cons nodeg ul)) nodeg nodeh)
      end
      end
| (mu_all t g) => Cases (MapGet ? bre t) of
      NONE => (cfg,BDDzero)
      | (SOME nodet) =>
      Cases (BDDmu_eval g cfg ul bre) of (cfgg,nodeg) =>
      (BDDmu_all N gc cfgg (cons nodeg ul) nodet nodeg)

```

```

        end
      end
    | (mu_mu P g) =>
      (BDDiter [cfg;node] (BDDmu_eval g cfg (cons node ul) (MapPut ? bre P node))
        cfg BDDzero (two_power N))
  end.

```

For evaluating the formula $(\mu_{\text{all}} t g)$ we use the function `BDDmu_all` defined below.

Definition 56

```

Definition BDDmu_all := [N:nat;gc:BDDconfig->(list ad)->BDDconfig;cfg:BDDconfig;
  ul:(list ad); nodet,nodeg:ad]
Cases (BDDreplacel gc cfg ul nodeg (lx N) (lx' N)) of (cfgr,noder) =>
  Cases (BDDimpl gc cfgr (cons noder ul) nodet noder) of (cfgi,nodei) =>
    (BDDunivl gc cfgi (cons nodei ul) nodei (lx' N))
  end
end.

```

That is, we compute $\forall l'. t(l, l') \Rightarrow G[l := l']$ where l and l' are the vectors of variables $(lx\ N)$ and $(lx'\ N)$. The bound supplied to `BDDiter` in the function `BDDmu_eval` is 2^N . We are guaranteed to obtain the fixpoint in at most these many iterations. However if Coq uses call-by-value strategy for evaluating expressions, then it would first have to compute the bound 2^n in unary representation. This is wasteful, as we may actually obtain the fixpoint in a much smaller number of iterations. So to be efficient, we would need to use an evaluation strategy in which the expression of an inductive type is simplified only if required (when doing "Case" analysis.) This kind of problem is of course not encountered in implementation of normal model checkers because they do not need to supply a bound on the depth of recursion.

Also, we have used `BDDiter` in such a way that the BDDs computed during one iteration can be freed during the next iterations. At the beginning of each iteration, the node corresponding to the current approximation to the fixed point is added to the used list. In the next iteration, it is replaced by the result of this iteration.

To be able to prove the correctness of this algorithm with respect to the semantics of μ -calculus formulae, we need to define some conditions when the environments are well formed.

`(cfg_ul_bre_ok cfg ul bre)` means that all the BDD nodes referred to by the relational environment are used nodes (i.e. are reachable from some node in the used list `ul` or are leaf nodes.) `(cfg_re_bre_ok cfg re bre)` means that the relational environment `bre` using BDDs is consistent with the relational environment `re` using boolean expressions. `(f_bre_ok f bre)` means that all relational variables occurring free in the μ -calculus formula `f` are in the domain of the relational environment `bre`. We also have similar definitions for transition environments.

Definition 57

```

Definition cfg_ul_bre_ok := [cfg:BDDconfig;ul:(list ad);bre:Brel_env]
  (P,node:ad) (MapGet ? bre P)=(SOME ? node)->(used_node' cfg ul node).

```

Definition 58

```

Definition cfg_re_bre_ok := [cfg:BDDconfig;re:rel_env;bre:Brel_env]
  (P,node:ad) (MapGet ? bre P)=(SOME ? node)
  -> (bool_fun_eq (bool_fun_of_BDD cfg node) (bool_fun_of_bool_expr (re P))).

```

Definition 59

```

Definition f_bre_ok := [f:mu_form;bre:Brel_env]
  (P:ad) (mu_rel_free P f)=true->(in_dom ? P bre)=true.

```

Definition 60

```
Definition cfg_ul_bte_ok := [cfg:BDDconfig;ul:(list ad);bte:Btrans_env]
  (t,node:ad)(MapGet ? bte t)=(SOME ? node)->(used_node' cfg ul node).
```

Definition 61

```
Definition cfg_te_bte_ok := [cfg:BDDconfig;te:trans_env;bte:Btrans_env]
  (t,node:ad)(MapGet ? bte t)=(SOME ? node)
  -> (bool_fun_eq (bool_fun_of_BDD cfg node) (bool_fun_of_bool_expr (te t))).
```

Definition 62

```
Definition f_bte_ok := [f:mu_form;bte:Btrans_env]
  (t:ad)(mu_t_free t f)=true->(in_dom ? t bte)=true.
```

We next prove the required correctness lemma for `BDDmu_eval`. We have the well formedness assumptions, described above, about the environments, as well as the conditions that the configuration and used list are well formed. We show that the output configuration and node are well formed, and the used nodes from the previous configuration are preserved, besides proving that the semantics of the new node as boolean function is consistent with the semantics of μ -calculus formulae.

Lemma 23

```
Lemma BDDmu_eval_ok : (N:nat; gc:(BDDconfig->(list ad)->BDDconfig))(gc_OK gc)
  ->(te:trans_env; bte:Btrans_env; f:mu_form; cfg:BDDconfig;
    ul:(list ad); re:rel_env; bre:Brel_env)
  (BDDconfig_OK cfg)
  ->(used_list_OK cfg ul)
  ->(cfg_ul_bre_ok cfg ul bre)
  ->(cfg_re_bre_ok cfg re bre)
  ->(cfg_ul_bte_ok cfg ul bte)
  ->(cfg_te_bte_ok cfg te bte)
  ->(f_bre_ok f bre)
  ->(f_bte_ok f bte)
  ->(BDDconfig_OK (Fst (BDDmu_eval N gc bte f cfg ul bre)))
    /\(config_node_OK (Fst (BDDmu_eval N gc bte f cfg ul bre))
      (Snd (BDDmu_eval N gc bte f cfg ul bre)))
    /\(used_nodes_preserved cfg (Fst (BDDmu_eval N gc bte f cfg ul bre)) ul)
    /\(bool_fun_eq (bool_fun_of_BDD
      (Fst (BDDmu_eval N gc bte f cfg ul bre))
      (Snd (BDDmu_eval N gc bte f cfg ul bre)))
      (bool_fun_of_bool_expr (mu_eval N te f re))).
```

Proof: The proof uses induction on the formula f . The proof is mostly straightforward, and uses the corresponding results about the BDD functions `BDDmake`, `BDDand`, etc. We need to show that the function `BDDmu_all_eval` returns a well formed configuration and nodes, preserves used nodes from the old configuration and has the expected semantics in terms of boolean functions (according to the corresponding function `bool_fun_mu_all` on boolean functions.) The main difficulty is in the case of the fixpoint operator. We require to prove a corresponding result for the function `BDDiter`. We prove that if the function (in this case it is `BDDmu_eval` applied on the appropriate formula) in the input of `BDDiter` is OK (i.e. returns configurations and nodes which are OK, has the required semantics etc.), then `BDDiter` applied to that function also has all the required properties. The precise lemma which we prove is stated below.

Lemma 24

```

Lemma BDDiter_lemma1 :
(te:trans_env;bte:Btrans_env;g:mu_form)
(Bf:mu_form->BDDconfig->(list ad)->Brel_env->BDDconfig*ad;f:mu_form->rel_env->bool_expr)
(
  (cfg:BDDconfig; ul:(list ad); re:rel_env; bre:Brel_env)
    (BDDconfig_OK cfg)
    ->(used_list_OK cfg ul)
    ->(cfg_ul_bre_ok cfg ul bre)
    ->(cfg_re_bre_ok cfg re bre)

    ->(cfg_ul_bte_ok cfg ul bte)
    ->(cfg_te_bte_ok cfg te bte)
    ->(f_bre_ok g bre)
    ->(f_bte_ok g bte)

    ->(BDDconfig_OK (Fst (Bf g cfg ul bre)))
    /\(config_node_OK (Fst (Bf g cfg ul bre)) (Snd (Bf g cfg ul bre)))
    /\(used_nodes_preserved cfg (Fst (Bf g cfg ul bre)) ul)
    /\(bool_fun_eq
      (bool_fun_of_BDD (Fst (Bf g cfg ul bre)) (Snd (Bf g cfg ul bre)))
      (bool_fun_of_bool_expr (f g re)))
  )
-> (n:nat;P:ad;cfg:BDDconfig;node:ad;ul:(list ad);be:bool_expr;re:rel_env;
  bre : Brel_env)
(BDDconfig_OK cfg)
-> (config_node_OK cfg node)
-> (used_list_OK cfg ul)
-> (cfg_ul_bre_ok cfg ul bre)
-> (cfg_re_bre_ok cfg re bre)

-> (cfg_ul_bte_ok cfg ul bte)
-> (cfg_te_bte_ok cfg te bte)
-> ((x:ad)(f_bre_ok g (MapPut ? bre P x)))
-> (f_bte_ok g bte)
-> (bool_fun_eq (bool_fun_of_BDD cfg node) (bool_fun_of_bool_expr be))
-> (BDDconfig_OK
  (Fst (BDDiter [cfg0:BDDconfig; node0:ad]
    (Bf g cfg0 (cons node0 ul) (MapPut ad bre P node0))
    cfg node n)))
/\(config_node_OK
  (Fst (BDDiter [cfg0:BDDconfig; node0:ad]
    (Bf g cfg0 (cons node0 ul) (MapPut ad bre P node0))
    cfg node n))
  (Snd (BDDiter [cfg0:BDDconfig; node0:ad]
    (Bf g cfg0 (cons node0 ul) (MapPut ad bre P node0))
    cfg node n)))
/\(used_nodes_preserved cfg
  (Fst (BDDiter [cfg0:BDDconfig; node0:ad]
    (Bf g cfg0 (cons node0 ul) (MapPut ad bre P node0))
    cfg node n)) ul)
/\(bool_fun_eq (bool_fun_of_BDD
  (Fst (BDDiter [cfg0:BDDconfig; node0:ad]
    (Bf g cfg0 (cons node0 ul) (MapPut ad bre P node0))

```

```

      cfg node n))
(Snd (BDDiter [cfg0:BDDconfig; node0:ad]
  (Bf g cfg0 (cons node0 ul) (MapPut ad bre P node0))
  cfg node n))
(bool_fun_of_bool_expr
  (iter ? be_eq_dec [be:bool_expr] (f g (re_put re P be)) be n))).

```

Proof: The proof follows in a straightforward manner by induction on n . □

□

□

Chapter 8

Conclusion

We have described an implementation and a proof of correctness of a symbolic model checker for the μ -calculus using BDDs, formalized inside the Coq proof assistant. This not only proves the correctness of μ -calculus algorithm, but also allows us to run it inside Coq. This provides a safe way of integrating symbolic model checking techniques within a proof assistant, since the results provided by the model checker can directly be introduced as theorems inside the proof assistant.

We first extended the earlier BDD implementation with operations like universal quantification over a list of variables, substitution, and replacement of a list of variables by another list of variables. We then used this BDD implementation to implement a model checking algorithm for the μ -calculus by defining a translation from μ -calculus formulae to BDDs.

This required tackling a number of problems. Formalizing the semantics of μ -calculus formulae as boolean expressions was problematic in the case of the fixpoint operator because of the fact that to be able to define it as a fixed point, we needed to prove simultaneously that the semantics was monotonic. We instead chose to define the semantics of the fixpoint operator as an iterative computation, and showed separately that it actually returned a fixed point provided some well-formedness conditions were satisfied.

We also had a problem because of the restriction that Coq imposes on definitions recursive functions, namely, that they need to be primitive recursive. So to compute the BDD for $\mu P.f$ by iteration, we need to supply a bound on the number of iterations, which is 2^N , where N is the number of boolean variables we are working with. However if Coq follows a simple call-by-value strategy for reduction of expressions, then it would have to compute 2^N in unary representation before performing the iterations, even if the actual number of iterations required is much less. This problem can be overcome by using an appropriate reduction strategy.

Proving the correctness of the function `BDDmu_eval` (which computes the BDD for a μ -calculus formula) was a bit involved because of the way the function has been defined – we recursively define `BDDmu_eval`, inside which we call `BDDiter` which iterates the function `BDDmu_eval` a certain number of times. It required us to figure out the appropriate lemma (Lemma 24) about `BDDiter`, whose statement is rather long, even though the proof follows in a straightforward way.

The whole implementation is a rather large piece of work. For example, the BDD implementation we used contained 9000 lines of Coq code. The implementation of quantification, substitution and the implementation of the model checker took around 3700 lines more.

8.1 Related Work

One piece of work very much related to ours is John Harrison's interfacing of a BDD library with the HOL prover [Har95]. The author's goal is not to do any model-checking, rather to solve the validity problem for propositional logic formulae. Because the logical language of HOL does not contain a programming sublanguage as NQTHM or Coq, reflection is not an option: the BDD library must log every BDD reduction step and translate each as a proper HOL inference, which HOL will then check. This logging process is

difficult to implement, produces huge logs, and it is necessary to use a few tricks in the HOL implementation to keep the HOL proof-checking phase reasonably efficient.

There has already been some work done on the subject of integrating model checking and theorem proving. It is one of the main aims of PVS [ORS92] to do so; but the internal model-checker of PVS is itself not checked formally. One of the closest works to ours in the literature is Yu and Luo's paper on LegoMC [YL97]. They build a model-checker, LegoMC, for checking μ -calculus formulae against CCS programs; on success, LegoMC returns a proof-term expressing why it believes the given formula to hold on the given program. Safety is achieved through Lego checking the latter proof-term. Interesting as it may be for simple verification conditions, this technique does not scale up much to more complex hardware circuit problems. In fact, for such problems, BDDs are mostly the only choice, and we have already seen that, although Harrison had managed to interface BDDs with HOL in such a way [Har95], the results were not entirely satisfactory.

Gordon and his colleagues have combined the HOL proof assistant with an external BDD package [Gor99, GL99]. They allow the HOL prover to accept results from an external BDD package, without any rechecking by HOL. While this is likely to work faster, its safety is dependent on the reliability of the external BDD package as well as the mechanisms for communication between HOL and the BDD package. Our approach involves more work but is completely safe.

Christoph Sprenger [Spr98] verified a model checker for the μ -calculus in Coq using Winskel's local model checking algorithm [Win91]. Since experimental results are not available, it is difficult to conclude anything about how well it works in practice. Our work differs in that we use symbolic model checking using BDDs, while Sprenger uses Winskel's local model checking algorithm [Win91]. In addition, we wanted to be able to run the model checker inside Coq, besides running the extracted Caml programs. For this purpose, we have tried to keep our executable programs separate from their proofs, so that we don't have to carry around proof terms while executing the programs in Coq (although these proof terms are removed in the extracted Caml programs.)

8.2 Possible Extensions

There are several possibilities for this work being continued in the future. First, although the BDD implementation has been extensively benchmarked and has been seen to work on problems of realistic size [VGL00, VGLPAK], the usability of the model checker based on it has yet to be examined.

Following this first step, it would also be useful to implement model checkers for other logics (CTL, CTL*, etc.) as well as implement more sophisticated and efficient algorithms.

Experiments with the BDD implementation [VGL00, VGLPAK] show a tremendous improvement in speed and space requirements on running the extracted Caml programs instead of running the programs inside Coq. For example, the extracted Caml programs run around 150 times faster than the programs in Coq. This is the price that has to be paid for running the programs inside Coq's interpreted λ -calculus. However, the problem with the extracted programs is that, as of now, there is no way of allowing Coq to accept the results provided by the extracted programs. Thus it would be of great use (not just for this application) to examine ways in which Coq could be allowed to accept results provided by its extracted programs. This would have to be done in a such a way that the safety of Coq is not compromised.

On the subject of integrating model checking and theorem proving, it would be useful to examine general proof techniques for program verification that work on top a model checker implemented inside a proof assistant.

Bibliography

- [Bry86] Randall E. Bryant. Graph-based algorithms for boolean functions manipulation. *IEEE Transactions on Computers*, C35(8):677–692, August 1986.
- [GL98] Jean Goubault-Larrecq. Satisfiability of inequality constraints and detection of cycles with negative weight in graphs. Part of the Coq contribs, available at <http://pauillac.inria.fr/coq/contribs/graphs.html>, 1998.
- [GL99] Mike Gordon and Ken Friis Larsen. Combining the Hol98 proof assistant with the BuDDy BDD package. Technical Report 481, University of Cambridge Computer Laboratory, December 1999.
- [Gor99] Mike Gordon. Programming combinations of deduction and BDD-based symbolic calculation. Technical Report 480, University of Cambridge Computer Laboratory, December 1999.
- [Gou94] Jean Goubault. Standard ML with fast sets and maps. In *5th ACM SIGPLAN Workshop on ML and its Applications (ML'94)*. ACM Press, Orlando, FL, USA, June 1994.
- [Har95] John Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38:162–170, 1995.
- [HKPM98] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. *The Coq Proof Assistant, A Tutorial*. Coq Project, Inria, 1998. Draft, version 6.2.4. Available at <http://coq.inria.fr/doc/tutorial.html>.
- [Knu73] D. E. Knuth. *The Art of Computer Programming III: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973.
- [McM93] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
- [ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *CADE'92*, pages 748–752. Springer Verlag LNAI 607, Saratoga Springs, June 1992.
- [Spr98] Christoph Sprenger. A verified model checker for the modal μ -calculus in Coq. In *TACAS '98*, LNCS. Springer Verlag, 1998.
- [VGL00] Kumar Neeraj Verma and Jean Goubault-Larrecq. Reflecting BDDs in Coq. Research Report RR-3859, INRIA, 2000.
- [VGLPAK] Kumar Neeraj Verma, Jean Goubault-Larrecq, Sanjiva Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *ASIAN 2000*, LNCS. Springer Verlag.
- [Win91] Glynn Winskel. A note on model checking the modal ν -calculus. *Theoretical Computer Science*, 83(1):157–167, 21 June 1991.
- [YL97] Shenwei Yu and Zhaohui Luo. Implementing a model checker for LEGO. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97*, volume 1313 of *LNCS*, pages 442–458. Springer-Verlag, September 1997.