FUNCTIONAL PEARLS

Functional satisfaction

Luc Maranget
Inria Rocquencourt, BP 105
78153 Le Chesnay Cedex, France
(*e-mail:* `Luc.Maranget@inria.fr`)

Abstract

This work presents simple decision procedures for the propositional calculus and for a simple predicate calculus. These decision procedures are based upon enumeration of the possible values of the variables in an expression. Yet, by taking advantage of the sequential semantics of boolean connectors, not all values are enumerated. In some cases, dramatic savings of machine time can be achieved. In particular, an equivalence checker for a small programming language appears to be usable in practice.

1 Introduction

In this paper we propose a simple, yet reasonably efficient decision procedure for the propositional calculus and for a simple predicate calculus. By "*simple*" we mean a technique inspired by the semantics of the propositional calculus, not a sophisticated, resource aware, technique such as binary decision diagrams. Whereas, by "*reasonably efficient*" we mean more efficient than the most naive decision procedures.

We first consider the evaluation of boolean expressions with variables. Given a boolean expression $e_1 \vee e_2$, if the evaluation of e_1 yields true, there is no need to evaluate e_2: the answer is true regardless of the truth-value of e_2. More generally it seems wise to use such a sequential (or short-circuiting) evaluation of propositions: it never hurts and may, in some circumstances, yield important savings over a direct application of the definitions of the boolean connectors.

Starting from a function E for evaluating boolean expressions, it is possible to solve more complex problems. For instance, one can check whether proposition e_2 is a tautology or not, by enumerating all the possible truth-value assignments of x_1, x_2, $\ldots x_n$, where x_1, x_2, $\ldots x_n$ are the variables of e_2. This simple idea can be expressed by the following pseudo-code:

for x_1 `in true, false do`
    `...`
    `for` x_n `in true, false do`
      `if` $E(e, x_1, x_2, \ldots, x_n)$ `= false then` e `is not a tautology`

The procedure sketched above does not take a significant advantage of sequential evaluation of boolean connectors. Let for instance consider the case when e is $x_1 \vee e_2$. Then, the procedure will blindly perform 2^n evaluations of e. However, when x_1 is true, the truth-value of $x_1 \vee e_2$ is true, regardless of the assignments of the remaining variables $x_2, \ldots x_n$ and all the inner loops are useless. More generally, while enumerating all assignments for the variables of $e_1 \vee e_2$, there is no need to enumerate the assignments for the variables of e_2, as soon as the truth-value of e_1 is true.

Intuitively, taking advantage of sequential evaluation of boolean connectors in such a context means mixing enumeration of variable assignments and evaluation of boolean expressions. This combination can be achieved quite easily in a functional language such as Objective Caml (Ocaml, 2003). The trick is to consider a continuation-based semantics of the propositional calculus. First class-functions then permit a straightforward implementation.

This paper is organized as follows. Section 2 recalls the definition of sequential evaluation of boolean connectors, and gives simple Caml code for an evaluator. Then, in section 3, we show how to turn this evaluator into an enumerator. Finally, section 4 considers extending the propositional calculus with monadic predicates.

2 Evaluation of propositions

2.1 Church booleans

In the λ-calculus, one can express the booleans true and false as $\lambda tf.t$ and $\lambda tf.f$.

```
let b_true  kt kf = kt  (* b_true  : 'a -> 'b -> 'a *)
and b_false kt kf = kf  (* b_false : 'a -> 'b -> 'b *)
```

The "if" construct being $\lambda ctf.c\ t\ f$, one expresses the boolean connectors as follows:

```
let b_not b kt kf = b kf kt
(* b_not : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c *)
let b_and b1 b2 kt kf = b1 (b2 kt kf) kf
(* b_and : ('a -> 'b -> 'c) -> ('d -> 'b -> 'a) -> 'd -> 'b -> 'c *)
and b_or b1 b2 kt kf = b1 kt (b2 kt kf)
(* b_or : ('a -> 'b -> 'c) -> ('a -> 'd -> 'b) -> 'a -> 'd -> 'c *)
```

Then, a boolean expression is written by calling the appropriate functions.

```
let b = b_and b_false (b_or b_true (b_or b_false b_false))
(* b : '_a -> '_b -> '_b  *)
```

More generally, applying the functional boolean connectors yields a function, which we call a functional boolean. One may remark that a functional boolean is either the function `b_true` or the function `b_false`, and that the inferred type of functional booleans tells us which they are — see also (Mairson, 2004) in this issue. However, one may wish to recover more traditional truth-values:

```
let eval b = b true false (* eval : (bool -> bool -> 'a) -> 'a *)
```

2.2 Variables

Let us now enrich our very basic calculus with variables. Environment lookup and construction are implemented by two functions. Function `find` takes a variable name `x` (of type `string`) and an environment `env` (of type `'a env`) as arguments and returns `Some` v if `env` binds `x` to v, or `None` when `x` is unbound, while function `add` adds or updates a binding to an environment.

```
find : string -> 'a env -> 'a option
add  : string -> 'a -> 'a env -> 'a env
```

Here, in the case of the propositional calculus, it is natural to bind propositional variables to machine booleans, and the following function `b_var` takes an environment as an extra argument.

```
let b_var x kt kf env = match find x env with
| Some true  -> kt env | Some false -> kf env
(* b_var :
   string -> (bool env -> 'a) -> (bool env -> 'a) -> bool env -> 'a *)
```

Presently, the `b_var` function does nothing but translating machine booleans into functional ones. The previous functional definitions of the boolean connectors remain unchanged. Now, the proposition $x \vee \neg x$ can be written as:

```
let bx =  b_or (b_var "x") (b_not (b_var "x"))
(* bx : (bool env -> '_a) -> (bool env -> '_a) -> bool env -> '_a *)
```

When supplemented by the definition of `b_var`, the application of functional connectors yields a variety of *propositional* functions. Notice that, in contrast to functional booleans, propositional functions all have the same type.

The following function, `eval`, evaluates a propositional function `b` w.r.t. an environment `env`; it returns a machine boolean.

```
let eval b env = b (fun _ -> true) (fun _ -> false) env
(* eval : (('a -> bool) -> ('b -> bool) -> 'c -> 'd) -> 'c -> 'd *)
```

One can see the definitions of `b_var` and of the functional connectors as a denotational, continuation-based, semantics of the propositional calculus. The evaluator directly derives from this semantics.

3 Enumeration

3.1 Intuition

A slight modification of the `b_var` function will turn our evaluating propositional functions into enumerating ones. It suffices to add a clause for unbound variables.

```
let b_var x kt kf env = match find x env with
| Some true -> kt env | Some false -> kf env
| None -> kt (add x true env) ; kf (add x false env)
(* b_var : string ->
     (bool env -> unit) -> (bool env -> unit) -> bool env -> unit *)
```

Intuitively, continuations `kt` and `kf` represent the computations still to be performed when `x` is bound to true or false respectively. If `x` is unbound, then `b_var` considers both possibilities. Notice that the sequence operator "`;`" is used, hence `kf` and `kt` are meant to be called for their side effects.

We can now list "all" possible assignments of the variables of some proposition by feeding `b` with the following two initial continuations.

```
let print_true env = print_env env ; print_endline " -> True"
and print_false env =  print_env env ; print_endline " -> False"

let enum b = b print_true print_false empty
```

Where `empty` is the empty environment.

A run of `enum` on the functional proposition that encodes $((\neg x \vee y) \vee z) \vee x$ outputs the following list:

```
x=t, y=t -> True
x=t, y=f, z=t -> True
x=t, y=f, z=f -> True
x=f -> True
```

As a consequence of the sequential evaluation of functional connectors, not all the 2^3 possible assignments for variables x, y and z are listed. However, the list is complete: a line may stand for several assignments when it does not show some variables. For instance, the first line above stands for the two assignments `x=t, y=t, z=t -> True` and `x=t, y=t, z=f -> True`, while the last line above stands for four complete assignments.

Function `enum` performs better on the equivalent proposition $(x \vee (\neg x \vee y)) \vee z$:

```
x=t -> True
x=f -> True
```

More generally, enumerating any disjunction of the four terms x, $\neg x$, y and z will yield either two or four lines. Obviously, `enum` output can be understood as disjunctive normal forms (take the disjunction of all lines seen as conjunctions), however it is important to notice that no actual terms get built.

3.2 Correctness and completeness of `enum`

A precise statement of the properties of `enum` requires a few definitions.

First, a point on the truth-value of a proposition is worth mentioning. Such truth-values are defined operationally (by the `eval` function of section 2.2). Hence, saying "the truth-value of b w.r.t. `env` is true" in fact means: "evaluating b in environment `env` by using the sequential (left-to-right) semantics of boolean connectors yields true".

Then, given two environments `env` and `env'`, we say that `env'` extends `env` when `env'` holds at least the same bindings as `env`. Observe that if the truth-value of b

w.r.t. `env` is fixed, then the truth-value of b does not change w.r.t. any environment extending `env`.

Now, we can state that `enum` is correct and complete. Given an expression b (encoded as propositional function `b`) and an environment `env`, evaluating the application `b kt kf env` will result in calling `kt` (resp. `kf`), if and only if there exists an environment `env'`, such that

1. the environment `env'` extends `env`,
2. and the truth-value of b w.r.t. environment `env'` is true (resp. false).

Given the nature of this work, we shall omit the proof, which is by induction on the structure of propositional functions.

3.3 Flexibility

Enumerating propositional functions can be used to decide various properties. For instance, the following functions check for a proposition to be a tautology and to be satisfiable.

```
exception Exit

let always_true b =
  try b (fun _ -> ()) (fun _ -> raise Exit) empty ; true
  with Exit -> false
(* always_true :
    (('a -> unit) -> ('b -> 'c) -> 'd env -> 'e) -> bool *)

let maybe_true b =
  try b (fun _ -> raise Exit) (fun _ -> ()) empty ; false
  with Exit -> true
(* maybe_true :
     (('a -> 'b) -> ('c -> unit) -> 'd env -> 'e) -> bool *)
```

The `always_true` function enumerates "all" the assignments of the variables of proposition b. If the truth-value of b w.r.t. one such assignment is false, then b is not a tautology and enumeration can stop. This is performed by raising exception `Exit`. Otherwise, there is no assignment `env` such that the truth-value of b w.r.t. `env` is false, and b is a tautology. The `maybe_true` function acts symmetrically. Moreover it could have be written as `not (always_true (b_not b))`. The correctness of these functions directly stems from section 3.2.

3.4 Avoiding side-effects

Imperative style can be avoided by considering different definitions of `b_var` for different properties. For instance, we can replace the sequence operator "`;`" by the conjunction operator "`&&`".

```
let b_var x kt kf env = match find x env with
| Some true  -> kt env | Some false -> kf env
| None -> kt (add x true env) && kf (add x false env)
(* b_var : string ->
    (bool env -> bool) -> (bool env -> bool) -> bool env -> bool *)
```

Then, a tautology checker simply is:

```
let always_true b = b (fun _ -> true) (fun _ -> false) empty
(* always_true :
     (('a -> bool) -> ('b -> bool) -> 'c env -> 'd) -> 'd *)
```

Observe that, to check satisfiability, it would suffice to replace `&&` by `||` in the definition of `b_var`.

This solution for avoiding side-effects works independently of the implementation language, either strict or lazy. However, as suggested by an anonymous referee, one can take better advantage of lazyness. In a lazy langauge, one could write a new function `enum` (cf. Section 3) that would return a list of pairs of environment and truth value: `(bool env * bool) list`, instead of printing this information. Then, to test for tautology (resp. satisfiability), the list can be lazily reduced, returning as soon as false (resp. true) is found in the second component.

4 Beyond the propositional calculus

Our technique easily extends to more complex calculi. For instance, we can extend the propositional calculus with monadic (i.e., one-argument) predicates $P(x)$, $Q(x)$, etc. Environments now bind a variable x to a list of atomic predicates, either positive ($P(x)$) or negative ($\neg P(x)$). This list represents the conjunction of its elements (a constraint on x). And an environment represents the conjunction of its bindings.

4.1 A monadic predicate calculus

We consider a monadic predicate $x = i$ where x is a variable and i is an integer. The full syntax of the calculus is:

$$C \quad ::= \quad (\texttt{EQUALS}\ x\ i) \ \ | \ \ (\texttt{OR}\ C \ldots C) \ \ | \ \ (\texttt{AND}\ C \ldots C)$$

This calculus is the language of *conditions* of the small programming language of 1999 ICFP programming contest (Ramsey & Scott, 2000). Notice that disjunctions and conjunctions take an arbitrary number of arguments. The corresponding abstract syntax is:

```
type cond =
  Equals of string * int | Or of cond list | And of cond list
```

During enumeration, variables do in fact not range over lists of atomic predicates. Instead, they range over the interpretation of these lists as finite and co-finite sets of integers (co-finite sets are sets whose complement is finite). We assume that such sets are implemented by the module `Ints`, whose signature follows:

```
type t
val empty : t
val universe : t
val singleton : int -> t
val is_empty : t -> bool
val inter : t -> t -> t
val complement : t -> t
```

This module provides the type `Ints.t` of sets, the usual functions on sets (`inter`, `is_empty` etc.), the function `complement` of type `Ints.t -> Ints.t` for complementing sets, and the value `universe` that represents the whole set of integers.

Using finite and co-finite sets slightly simplifies environments: the `option` type is no longer needed. The role of `None` is taken by `Ints.universe` and the role of `Some` v is taken by v. As a consequence, the function `find` now has the more simple type `string -> 'a env -> 'a`.

Enumeration is in practice performed at the predicate level:

```
let b_equals x i kt kf env =
  let vx = find x env in
  let vt = Ints.inter vx (Ints.singleton i) in
  if not (Ints.is_empty vt) then kt (add x vt env) ;
  let vf = Ints.inter vx (Ints.complement (Ints.singleton i)) in
  if not (Ints.is_empty vf) then kf (add x vf env)
(* b_equals :
  string -> int -> (Ints.t env -> unit) -> (Ints.t env -> unit) ->
    Ints.t env -> unit *)
```

The `b_equals` function is the analog of the `b_var` function of Section 3.1. The sets `vt` and `vf` compactly express the previously mentioned constraints on variable `x`. As an advantage of this technique, unsatisfiable constraints are detected and discarded as soon as they appear, by using the `Ints.is_empty` function.

Conditions of type `cond` are turned into enumerating functions as follows:

```
let rec compile_cond c kt kf = match c with
| Equals (x, i) -> b_equals x i kt kf
| Or []      -> kf
| Or (c::cs) ->
    b_or (compile_cond c) (compile_cond (Or cs)) kt kf
| And []      -> kt
| And (c::cs) ->
    b_and (compile_cond c) (compile_cond (And cs)) kt kf
(* compile_cond :
  cond -> (Ints.t env -> unit) -> (Ints.t env -> unit) ->
    Ints.t env -> unit *)
```

It is important to notice that, in contrast to `b_equals`, the function `compile_cond` has no "`env`" argument in last position. Indeed, the proposed `compile_cond` function is η-reduced. As a consequence, calling `compile_cond` with all its three "static" arguments yields some computations. More precisely, the connectors `b_or` and `b_and` are reduced, and compilation produces partial applications of `b_equals`. Hence, `compile_cond` arguably performs a compilation from conditions to Caml functions, provided the continuations `kt` and `kf` and known.

All functions of section 3.3 (`always_true`, etc.) are still working.

4.2 A program equivalence checker

The contest language for statements included an `if` construct, a `case` construct and a final `return` statement. We consider a minimal language, although we implemented the full contest language.

$$S \quad ::= \quad \texttt{(IF}\ C\ S\ S\texttt{)} \ \ \texttt{|} \ \ \texttt{(DECISION}\ j\texttt{)}$$

The (DECISION j) construct is the `return` statement, an integer j is returned.

The Caml data type for statements S is standard, like their semantics. As a consequence, the compilation function for statements is quite simple:

```
type stm = If of cond * stm * stm | Decision of int

let rec compile_stm s k = match s with
| If (c, st, sf) ->
    compile_cond c (compile_stm st k) (compile_stm sf k)
| Decision j -> k j
(* compile_stm :
  stm -> (int -> Ints.t env -> unit) -> Ints.t env -> unit *)
```

As shown by its type, continuation `k` takes two arguments, a decision and an environment. Hence, by using a continuation that prints the current environment and the decision made, one can write the analog of the `enum` function of section 3.1. For instance on the simple following decision program:

```
(IF (AND (EQUALS x 0) (EQUALS y 1)) (DECISION 0) (DECISION 1))
```

We get:

```
x:{0} y:{1}  -> 0
x:{0} y:~{1} -> 1
x:~{0}       -> 1
```

That is, the program reaches decision 0 for $x \in \{0\} \wedge y \in \{1\}$, while it reaches decision 1 for $x \in \{0\} \wedge y \in \mathbb{Z} \setminus \{1\}$ or $x \in \mathbb{Z} \setminus \{0\}$.

Our program equivalence tester is almost written, it suffices to find the appropriate continuations.

```
let equivalent_stm s1 s2 =
  let c2 r1 env1 =
    compile_stm s2 (fun r2 env -> if r1 <> r2 then raise Exit) env1 in
  let c1 = compile_stm s1 c2 in
  try c1 initial ; true with Exit -> false
```

Enumeration on `s1` starts in some `initial` environment that binds all variables to `universe`. Then, when a first decision `r1` is reached for some environment `env1`, a second enumeration on `s2` is started in environment `env1`. That way, all decisions that `s2` can reach by extending `env1` are compared to `r1`. For instance we can check that the previous decision program is equivalent to this other program:

```
(IF (AND (EQUALS y 1) (EQUALS x 0))
    (IF (EQUALS x 1) (DECISION 2) (DECISION 0))
    (DECISION 1))
```

The equivalence of the two programs results from the commutativity of `AND` and from the fact that `(EQUALS x 1)` occurs in a context where x value must be 0. By slighty modifying `equivalent_stm` so that it prints the environment when both decisions are reached and running this verbose version, we get:

```
x:{0}  y:{1}  -> 0, 0
x:{0}  y:~{1} -> 1, 1
x:~{0} y:{1}  -> 1, 1
x:~{0} y:~{1} -> 1, 1
```

Observe the the case $x \in \mathbb{Z} \setminus \{0\}$ now produces two lines. This stems from the opposite order of `AND` arguments in the two programs. However, our method still saves some tests, since a naive enumeration method would consider the additional condition $x \in \{1\}$.

The equivalence checker can be made more efficient in practice by a simple improvement. The key idea is compiling statement `s2` once to a Caml function. As mentioned at the end of section 4.1 in the case of conditions, the continuation `k` in `compile_stm s2 k` must be a fixed function. We achieve this with a reference cell.

```
let equivalent_stm s1 s2 =
  let r = ref 0 in (* any integer fits *)
  let c2 = compile_stm s2 (fun r2 env2 -> if !r<>r2 then raise Exit) in
  let c1 = compile_stm s1 (fun r1 env1 -> r := r1 ; c2 env1) in
  try c1 initial ; true with Exit -> false
```

The use of a reference cell to convey the successive values of `r1` is not that elegant. However, it is easy to convince oneself that the reference `r` is set to the proper value before `c2` is called. And hence, `r1` and `r2` are indeed compared.

The implemented equivalence checker is an optimized version of the one presented above. Optimizations consist in resolving references to variables at compile time (one variable is associated with one reference cell), avoiding enumeration when a condition can be found true or false by a simple scan (which is performed by another kind of evaluating functions, compiled from conditions), and reordering the arguments of connectors in order to present the most frequent variables first. In the case the previous example, this technique indeed saves some final decision comparison, since the arguments of the `AND` connector are scanned in some normalized order.

The equivalence checker has been tested on the contest inputs, by comparing one input program with the output of one available optimizer. With optimizations enabled, the equivalence checker runs in no more than a few seconds on any of the inputs. Note that contest inputs include one program with more than one thousand variables and one other program almost three megabytes large. Without optimizations, runtime is prohibitive on the largest inputs. More information on this benchmark is available at `http://pauillac.inria.fr/~maranget/enum/speed.html`.

5 Conclusion

As demonstrated by the program equivalence checker, our decision procedure is usable in practice. Of course, such a simple decision procedure cannot compete with more elaborated ones. In particular, in the case of the propositional calculus, Binary Decision Diagrams (Bryant, 1986) outperform it. However, the presented procedure remains efficient enough to provide a serious reference implementation.

Functional programming is crucial to the method presented in this paper both as a conceptual and implementation tool. First, the decision procedures directly derive from continuation-based semantics of the calculi. Hence, they remain simple and are likely to be programmed correctly. Second, performance partly relies on the compilation of terms of the calculi into closures.

Imperative constructs such as exceptions or reference cells prove useful for exploiting our enumeration technique for various purposes. However, we believe that this aspect is not important and that our technique can also be implemented in a lazy functional language such as Haskell.

References

Bryant, R. E. (1986) Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Conputers* **C-35**(8):677–691.

Mairson, H. G. (2004) Linear lambda calculus and P-TIME-completness. *Journal of Functional Programing*. In this issue.

Ocaml. (2003) *The Objective Caml Language (Version 3.07)*. `http://caml.inria.fr`.

Ramsey, N. and Scott, K. (2000) The 1999 ICFP Programming Contest. *SIGPLAN Notices* **35**(3):73–83. See also `http://www.cs.virginia.edu/~jks6b/icfp/`.