Verifying models with Dolmen

Guillaume Bury[1] **Francois Bobot**[2]

July 6th, 2023 – SMT Workshop

[1]OCamlPro SAS, Paris

[2]CEA-List, Université Paris-Saclay

Introducing Dolmen

What is Dolmen ?

- Parser + Typechecker for :
  SMT-LIB, TPTP, Dimacs, Alt-Ergo and
  Zipperposition's format
- Usable as:
  - an LSP server (i.e. text editor plugin)
  - a CLI binary
    -> used to check new benchmarks for the SMT-LIB
  - an OCaml library
    -> used in the frontend of Alt-Ergo and Colibri2
- Now also verifies SMT-LIB models !

Validating models

Model validation for ground formulas:

- Term/formula evaluation
- Not ground breaking research[1]
- Often already done internally in provers
- Not very rewarding

[1] pun intended

Validating models

Model validation for ground formulas:

- Term/formula evaluation
- Not ground breaking research[1]
- Often already done internally in provers
- Not very rewarding
- But useful, e.g. for SMT-COMP
- pySMT can do something similar, but does not support all theories

[1] pun intended

Verifying models: Example

example.smt2

```
(set-logic ALL)
(define-fun a () Int)
(define-fun b () Int)
(define-fun c () Int)
(assert (= (+ a b) c))
(check-sat)
```

example.rsmt2

```
sat
(
  (define-fun a () Int 1)
  (define-fun b () Int 2)
  (define-fun c () Int 3)
)
```

```
# dolmen --check-model=true -r example.rsmt2 example.smt2
```

Evaluating expressions

Typed expressions

- Variables (function parameter, let-bound variable)
- Constants (interpreted and non-interpreted symbols and functions)
- Function application (Dolmen supports higher-order)
- Binders (let-bindings, lambdas)
- A pattern match, that is a scrutinee and a list of patterns and arms:

```
match scrutinee with
| pattern1 -> arm1
| ...
```

Extensible values

Values are extensible: each theory can define its own kind of values

- Simple examples: bools, arbitrary size integers, ...
- Datatypes: head constructors + list of values for arguments
- Functions: arity + contents
  - OCaml code (for builtins)
  - Parameters + body
  - Ad-hoc instances for polymorphic functions

Evaluator structure

Three parts:

- Environment:
  - represented (partial) models
  - Maps variables and constants to values
- Core evaluator:
  - Takes a (typed) expression and an environment, returns a value
  - Handles the expression structure
- Theory-specific evaluation:
  - Handles all builtin symbols (e.g. addition)
  - Function from symbols to values
  - Returned values can represent functions

Core evaluator

- Variables: find it in the env

Core evaluator

- Variables: find it in the env
- Constants:
  - interpreted: ask the theory functions
  - non-interpreted: find it in the env

Core evaluator

- Variables: find it in the env
- Constants:
  - interpreted: ask the theory functions
  - non-interpreted: find it in the env
- Function application:
  - Evalute the callee and arguments
  - Function values have an arity
  - Accumulate partial applications until reduction can be done

Core evaluator

- Variables: find it in the env
- Constants:
  - interpreted: ask the theory functions
  - non-interpreted: find it in the env
- Function application:
  - Evalute the callee and arguments
  - Function values have an arity
  - Accumulate partial applications until reduction can be done
- Binders: evaluate defining expr, bind it to the variable in the env, then evaluate body

```
let x = defining_expr in body
```

Core evaluator

- Variables: find it in the env
- Constants:
  - interpreted: ask the theory functions
  - non-interpreted: find it in the env
- Function application:
  - Evalute the callee and arguments
  - Function values have an arity
  - Accumulate partial applications until reduction can be done
- Binders: evaluate defining expr, bind it to the variable in the env, then evaluate body

```
let x = defining_expr in body
```

- A pattern match: evaluate scrutinee, then match against each pattern, and evaluate the correct arm

```
match scrutinee with
| pattern1 -> arm1
| ...
```

Builtin example: Algebraic datatypes

```
let mk head args =
  Value.mk ~ops { head; args; }

let eval_tester cstr value =
  let { head; args = _ } = Value.extract_exn ~ops value in
  if C.equal cstr head then Bool.mk true else Bool.mk false

let eval_dstr ~eval env dstr cstr field tys arg =
  let { head; args; } = Value.extract_exn ~ops arg in
  if C.equal cstr head then List.nth args field
  else Fun.corner_case ~eval env dstr tys [arg]

let builtins ~eval env (cst : C.t) =
  match cst.builtin with
  | B.Constructor _ -> Some (Fun.fun_n ~cst (mk cst))
  | B.Tester { cstr; _ } ->
    Some (Fun.mk_clos @@ Fun.fun_1 ~cst (eval_tester cstr))
  | B.Destructor { cstr; field; _ } ->
    Some (Fun.mk_clos @@ Fun.poly ~arity:1 ~cst (fun tys args ->
        match args with
        | [arg] -> eval_dstr ~eval env cst cstr field tys arg
        | _ -> raise (Fun.Bad_arity (cst, 1, args))))
  | _ -> None
```

Evaluator library

Available as an OCaml library

```
type env (** evaluation environments *)
type builtins =
  eval:(env -> expr -> value) ->
  env -> symbol -> value option
(** Evaluation of builtin symbols *)

val builtins : builtins list -> builtins
(** Combine theory builtins functions *)

val mk_env : model -> builtins -> env
(** Environment creation *)

val eval : Env.t -> expr -> value
(** Evaluation function *)
```

Challenges

Partially specified builtins

Some builtin symbols in SMT-LIB are only partially specified

- Real division by zero
- Integer division and modulo by zero
- In floating point logics:
  - `fp.min` / `fp.max`
  - `fp.to_sbv`, `fp.to_ubv`
- Datatype selectors/destructors

```
(declare-datatypes ((list 1)) ((par (alpha) (
  (nil)
  (cons (head alpha) (tail (list alpha)))
))))
(assert (= 0 (head (tail (as nil (list Int))))))
```

Example: division by zero

```
(set-logic ALL)
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun c () Int)
(declare-fun d () Int)
(declare-fun z () Int)
(assert (= z 0))
(assert (= c (div a z)))
(assert (= d (div b z)))
(assert (not (= c d)))
(check-sat)
```

$$\begin{cases} z \mapsto 0 \\ a \mapsto 1 \quad c \mapsto 13 \quad \frac{1}{0} \mapsto 13 \\ b \mapsto 2 \quad d \mapsto 42 \quad \frac{2}{0} \mapsto 42 \end{cases}$$

Example: division by zero

$$\begin{cases} z \mapsto 0 \\ a \mapsto 1 \quad c \mapsto 13 \quad \frac{1}{0} \mapsto 13 \\ b \mapsto 2 \quad d \mapsto 42 \quad \frac{2}{0} \mapsto 42 \end{cases}$$

```
sat (
  (define-fun z () Int 0)
  (define-fun a () Int 1)
  (define-fun b () Int 2)
  (define-fun c () Int 13)
  (define-fun d () Int 42)
  (define-fun div ((x Int) (y Int)) Int
    (ite (and (= x 1) (= y 0)) 13
     (ite (and (= x 2) (= y 0)) 42
      (div x y)))))
```

Completing partially defined symbols

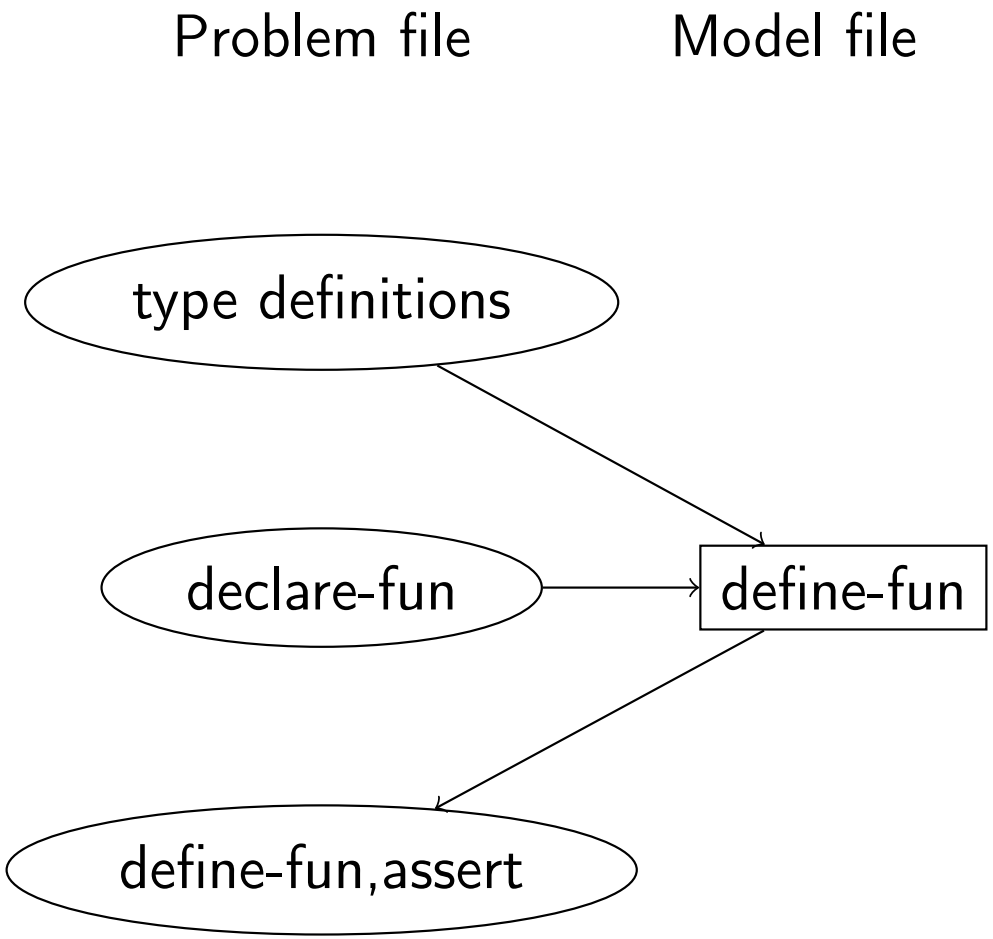
Solution:

- Models can define partially specified builtin symbols
- In theory evaluation functions, when a builtin is applied in an unspecified case, find the completed interpretation in the env, and evaluate it
- During that evaluation, the re-definition is removed from the environment/model (to avoid infinite recursion)

Example: ADT selector

```
(set-logic ALL)
(declare-datatypes ((list 1)) ((par (alpha) (
  (nil)
  (cons (head alpha) (tail (list alpha)))))))
(assert (= 0 (head (tail (as nil (list Int))))))
(check-sat)
; Model
sat (
  (define-fun head ((l (list Int))) Int
    (match l (
      (nil 0)
      ((cons hd tl) hd))))
  (define-fun tail ((l (list Int))) (list Int)
    (match l (
      ((cons hd tl) tl)
      (nil (as nil (list Int)))))))
```

Statement evaluation dependencies



The problem: concrete example

```
(set-logic ALL)
(declare-fun b () Int)
(define-fun c () Int (+ b
    1))
(declare-fun a () Int)
(assert (= a c))
(check-sat)
```

```
sat
(
  (define-fun a () Int 2)
  (define-fun b () Int 1)
)
```

- We need to store the typed expression for c until we have a in the model
- On bigger problems, we may need to store an arbitrary number of expressions until they can be evaluated
- Typed expressions take a lot of space

The problem: concrete example

```
(set-logic ALL)
(declare-fun b () Int)
(define-fun c () Int (+ b
    1))
(declare-fun a () Int)
(assert (= a c))
(check-sat)
```

```
sat
(
  (define-fun a () Int 2)
  (define-fun b () Int 1)
)
```

- We need to store the typed expression for c until we have a in the model
- On bigger problems, we may need to store an arbitrary number of expressions until they can be evaluated
- Typed expressions take a lot of space
- Alternatively, could store parsed model terms in memory, **assuming** all expressions are values with no dependencies (i.e. no sharing of sub-expressions beetween values)

Problem and solutions

- Problem: need to store arbitrary formulas to evaluate leater
- This can take a lot of memory !
- Potential solutions:
  - Require that model definitions are in the same order as declarations in the input problem
  - Require that problem are ordered:
    1. type definitions
    2. constant declarations
    3. term definitions and assertions

Conclusion

Conclusion

- Dolmen v0.9 can now verify ground models for all theories/logics (except Strings)[2]

[2] Algebraic numbers are implemented but not released yet, due to problems with some dependencies

[3] OCaml package manager

Conclusion

- Dolmen v0.9 can now verify ground models for all theories/logics (except Strings)[2]
- Dolmen will be used to check models at SMT-COMP

[2] Algebraic numbers are implemented but not released yet, due to problems with some dependencies

[3] OCaml package manager

Conclusion

- Dolmen v0.9 can now verify ground models for all theories/logics (except Strings)[2]
- Dolmen will be used to check models at SMT-COMP
- Dolmen is available on Opam[3], and at :
  `https://github.com/Gbury/dolmen`
  (binary releases available)

[2] Algebraic numbers are implemented but not released yet, due to problems with some dependencies

[3] OCaml package manager

Conclusion

- Dolmen v0.9 can now verify ground models for all theories/logics (except Strings)[2]
- Dolmen will be used to check models at SMT-COMP
- Dolmen is available on Opam[3], and at :
  `https://github.com/Gbury/dolmen`
  (binary releases available)
- Try and verify your solver's (ground) models with Dolmen

[2] Algebraic numbers are implemented but not released yet, due to problems with some dependencies

[3] OCaml package manager

Conclusion

- Dolmen v0.9 can now verify ground models for all theories/logics (except Strings)[2]
- Dolmen will be used to check models at SMT-COMP
- Dolmen is available on Opam[3], and at :
  `https://github.com/Gbury/dolmen`
  (binary releases available)
- Try and verify your solver's (ground) models with Dolmen
- Do not hesitate to submit issues !
  (bugs, questions, extension proposals, . . .)

[2] Algebraic numbers are implemented but not released yet, due to problems with some dependencies

[3] OCaml package manager

End

Questions ?

Abstract values

Abstract values (e.g. for arrays) :

```
sat (
  (define-fun a () (Array Int Int)
    (let ((arr (as @a0 (Array Int Int))))
    (store arr 1 42))))
```

Which interpretation ?

- Implicitly declared constant of the annotated type, therefore forbidding any use of the same name with a different type
- Implicit polymorphic constant of type `\forall a. a -> a`, constrained to the annotated type

```
    ... (as @0 t1) ....
    (as @0 t2) ...
```

Function representation

```
type value_function =
  | Lambda of {
      ty_params : E.Ty.Var.t list;
      term_params : E.Term.Var.t list;
      body : E.Term.t; }
  | Lazy of ... (* used for evaluating if-then-else *)
  | Poly of {
      arity : int;
      cst : E.Term.Const.t;
      eval_p : E.Ty.t list -> Value.t list -> Value.t; }
  | Ad_hoc of {
      arity : int;
      ty_arity : int;
      cst : E.Term.Const.t;
      eval_l : (E.Ty.t list * (E.Ty.subst -> value_function)) list; }
and t =
  | Closure of {
      func : value_function;
      tys : E.ty list; (* type args *)
      args : E.term list; (* partial applications arguments *)}
```

Typed expressions

```
type 'ty id = {
  id_ty : 'ty;
  index : index;
  path : Path.t;
  builtin : builtin;
  mutable tags : Tag.map; }
and type_ = Type
and ty_var = type_ id
and ty_cst = type_fun id
and ty_descr =
  | TyVar of ty_var
  | TyApp of ty_cst * ty list
  | Arrow of ty list * ty
  | Pi of ty_var list * ty
and ty = {
  mutable ty_hash : hash;
  mutable ty_tags : Tag.map;
  mutable ty_descr : ty_descr;
  mutable ty_head : ty; }
```

```
type term_var = ty id
and term_cst = ty id
and term_descr =
| Var of term_var
| Cst of term_cst
| App of term * ty list * term list
| Binder of binder * term
| Match of term * (pattern * term) li
and binder =
| Let_seq  of (term_var * term) list
| Let_par of (term_var * term) list
| Lambda of ty_var list * term_var li
| Exists of ty_var list * term_var li
| Forall of ty_var list * term_var li
and term = {
  term_ty : ty;
  term_descr : term_descr;
  mutable term_hash : hash;
  mutable term_tags : Tag.map; }
```