The Dalton Library

October 29, 2002

Contents

1 Module Dalton_aux : Auxiliary definitions
  1.1 Kinds .................................................. 1
  1.2 Variances .............................................. 2
  1.3 Contractor arguments .................................. 2
  1.4 Module Dalton_aux.Kind : Basic operations on kinds are provided by the module
  Variance ................................................. 2
  1.5 Module Dalton_aux.Variance : Basic operations on variances are provided by the
  module Variance ........................................ 3

2 Module Dalton_sig : Library module parameters. 3
  2.1 The ground algebra ..................................... 3
  2.2 Pretty-print ........................................... 4
  2.3 Error report ........................................... 4
  2.4 Drawing ................................................ 4
  2.5 Module type Dalton_sig.GROUND : The ground term algebra is specified by an im-
  plementation of the signature GROUND. .................. 4
  2.5.1 Constant bounds .................................... 4
  2.5.2 Row labels .......................................... 4
  2.5.3 Type constructors ................................... 4
  2.5.4 Module Dalton_sig.GROUND.Lb : The module Lb specifies the set of constant
  lower bounds .............................................. 5
  2.5.5 Module Dalton_sig.GROUND.Ub : The module Ub specifies the set of constant
  upper bounds ............................................. 6
  2.5.6 Module Dalton_sig.GROUND.Lub : The module Lub provides functions relating
  lower and upper bounds ................................ 7
  2.5.7 Module Dalton_sig.GROUND.Label : The set of row labels is defined by the
  module Label ............................................. 7
  2.5.8 Module Dalton_sig.GROUND.Type : Type constructors are given by the mod-
  ule Type ................................................ 8
  2.6 Module type Dalton_sig.PRINT : Printing of constraints may be parametrized by an
  implementation of the signature PRINT .................. 10
  2.7 Module type Dalton_sig.ERROR_REPORT : The implementation of the signature ERROR_REPORT
  given to the library allows customizing error messages printed when unification, res-
  olution or comparison fail .............................. 11
  2.7.1 Unification errors .................................. 11
2.7.2 Constraints solving errors ........................................ 12
2.7.3 Schemes comparison ............................................. 12
2.7.4 Minimal instance ................................................ 13
2.8 Module type: Dalton_sig.DRAW: Graphical representation of schemes is controlled
by a module of signature DRAW giving an implementation of drawing primitives. ...... 13

3 Module Dalton_templates: Templates of module parameters. ............... 14
3.1 Pretty-print ....................................................... 14
3.2 Drawing .......................................................... 14
3.3 Error report ....................................................... 14
3.4 Module Dalton_templates.Print: The module Print provides a standard style for
pretty-printing constraints. ............................................ 14
3.5 Module Dalton_templates.DrawGraphics: The module DrawGraphics provides
graphics primitives for the graphics library from the Objective Caml distribution ... 15
3.6 Module Dalton_templates.ErrorReport: The module ErrorReport provides stan-
dard error report messages. ........................................... 16

4 Module Dalton: The core of the library .................................. 17
4.1 Module Dalton.Make: The constraint solver comes as a functor parametrized by
four modules whose respective expected signatures are given in Dalton_sig[??]. .... 17
4.1.1 Constraint sets .................................................. 18
4.1.2 Terms ............................................................ 18
4.1.3 Setting constraints ............................................. 19
4.1.4 Substitutions ................................................... 20
4.1.5 Schemes ........................................................ 20
4.1.6 Module type: Dalton.Make.SCHEME_ROOT: A (type) scheme is made of a con-
straint set and a series of entry nodes, its roots. ................................ 20
4.1.7 Module Dalton.Make.Scheme: The functor scheme allows to build an imple-
mentation of functions dealing which each considered form schemes. .............. 21
4.1.8 Resolution ....................................................... 22
4.1.9 Comparison ..................................................... 22
4.1.10 Minimal instances ............................................ 23

1 Module Dalton_aux: Auxiliary definitions

This module define several auxiliary datatypes that are useful for the description of the ground
algebra. They are also used internally by the solver.

type 'a printer = Format.formatter -> 'a -> unit

Pretty-printing in the library is performed by the Format module. Therefore a printer of
values of type 'a may be viewed as a function of type 'a printer.

type printing = Format.formatter -> unit

Similarly, the printing of some message on a formatter may be abstractly represented by a
function of type printing.

type color = int
For drawing purposes, a color is represented by a simple integer, as in the Graphics module of the Objective Caml standard library.

```
type kind =
  | Katom
  | Ktype
  | Krow of kind

In the term algebra considered by the solver, terms may have one of the following kinds:
  • Katom for atoms,
  • Ktype for type,
  • Krow k for rows whose elements have kind k.
```

```
module Kind : sig end
[1.4]
```

```
type variance =
  | Covariant
  | Contravariant
  | Invariant

A variance is one of the three elements Covariant, Contravariant and Invariant.
```

```
module Variance : sig end
[1.5]
```

```
type constructor_arg = {
  variance : variance ;
  The variance of the argument.

  kind : kind ;
  The kind of the argument.

  ldestr : bool ;
  A boolean setting whether left destructor decomposes on this argument.

  rdestr : bool ;
  A boolean setting whether right destructor decomposes on this argument.
}
```

Signatures of type constructors are specified by giving for each argument a record of type constructor_arg.
Dalton_aux.Kind
Variance

module Kind : sig

val atomic : Dalton_aux.kind -> bool
  atomic k tests whether the kind k is atomic, i.e. is Katom or Krow Katom or Krow (Krow Katom) etc.

val rows : Dalton_aux.kind -> int
  rows k counts the number of Row in the kind k. For instance rows Katom and rows Ktype return 0, while rows (Krow Katom) and rows (Krow (Krow Ktype)) return respectively 1 and 2.

val fprint : Format.formatter -> Dalton_aux.kind -> unit
  fprint ppf k prints the kind k on the formatter ppf.

end

Dalton_aux.Variance
Variance

module Variance : sig

val leq : Dalton_aux.variance -> Dalton_aux.variance -> bool
  leq v1 v2 tests whether the variance v1 is less than or equal to the variance v2 in the usual order on variance (which is the smallest order such that Covariant and Contravariant are less than Invariant).

val combine :
  Dalton_aux.variance -> Dalton_aux.variance -> Dalton_aux.variance
  combine v1 v2 calculates the combination of two variances.

val to_string : Dalton_aux.variance -> string
  to_string v gives a string representation of the variance v.

val fprint : Format.formatter -> Dalton_aux.variance -> unit
  fprint ppf v prints the variance v on the formatter ppf, using one of the three symbols "+", "-" and "=".

val fprint_name : Format.formatter -> Dalton_aux.variance -> unit
fprint name ppf v prints the variance v on the formatter ppf, using its literal name (i.e. "covariant", "contravariant" or "invariant").

end

2 Module Dalton_sig : Library module parameters.

The Dalton solver is parametrized by several modules, which allow defining the term algebra, pretty-print, drawing and errors report. This module gives the expected signatures of this modules.

module type GROUND = sig end [2.5]

module type PRINT = sig end [2.6]

module type ERROR_REPORT = sig end [2.7]

module type DRAW = sig end [2.8]

Dalton_sig.GROUND

module type GROUND = sig
    It must defines datatypes for constant bounds, type constructors and row labels; and simultaneously operations on them. One may distinguish two categories of such operations: 1. Algebraic operations, which allows manipulating the mathematical properties of the provided objects, 2. Computational operations, which relate merely the internal representation of these objects and allow efficient algorithms (e.g. hashing, comparison, pretty-print...)
2.5.1 Constant bounds

The client must provide two sets of atomic constants, one for representing variables lower bounds and another one for upper bounds. These two sets must be equipped with a semi-lattice structure.

```ml
module Lb : sig end [2.5.4]
module Ub : sig end [2.5.5]
module Lub : sig end [2.5.6]
```

2.5.2 Row labels

```ml
module Label : sig end [2.5.7]
```

2.5.3 Type constructors

```ml
module Type : sig end [2.5.8]
end
```

2.5.4 Module Dalton_sig.GROUND.Lb: The module Lb specifies the set of constant lower bounds.

```ml
module Lb : sig

type t

The type of constant lower bound.

val bottom : t

bottom is a distinguished lower bound. It is the bottom element of the semi-lattice.

val is_bottom : t -> bool

is_bottom lb must return true if and only if lb is bottom.

val union : t -> t -> t

union lb1 lb2 gives the union (according to the semi-lattice structure) of the lower bounds lb1 and lb2.

val leq : t -> t -> bool
```
leq env lb1 lb2 tells whether lb1 is less than or equal to lb2 in the semi-lattice of constant lower bounds, i.e.: for all alpha, lb2 < alpha implies lb1 < alpha.

val compare : t -> t -> int
  A comparison function on constant lower bounds has to be provided. It is just used for computation and has no semantical meaning.

val normalize : t -> t
  normalize lb internally normalizes the lower bound lb.

val fprintf : Format.formatter -> t -> unit
  fprintf pff lb pretty-prints the constant lower bound lb on the formatter pff. (This function is used for printing of constants in constraints.)

val fprintf_in_term :
  int -> Format.formatter -> t -> unit
  fprintf_in_term pff lb is used to prints the constant lower bound lb on the formatter pff when it appears in a term, in place of a non-negative variable which has no predecessor. An usual implementation may be: let fprintf_in_term _ pff lb = Format.fprintf pff "> %a"
  fprintf lb

val draw : t -> string list
end

2.5.5 Module Dalton_sig.GROUND.Ub: The module Ub specifies the set of constant upper bounds.

module Ub : sig

type t
  The type of constant upper bound.

val top : t
  top is a distinguished upper bound. It is the top element of the semi-lattice.

val is_top : t -> bool
  is_top lb must return true if and only if lb is top.

val inter :
  t -> t -> t
  inter lb1 lb2 gives the intersection (according to the semi-lattice structure) of the lower bounds lb1 and lb2.
val geq : t -> t -> bool
    geq lb lb2 tells whether lb1 is greater than or equal to lb2 in the semi-lattice of constant upper bounds.

val compare : t -> t -> int
    A comparison function on constant upper bounds has to be provided. It is just used for computation and has no semantical meaning.

val normalize : t -> t
    normalize lb internally normalizes the lower bound lb.

val fprint : Format.formatter -> t -> unit
    fprint pff ub pretty-prints the constant upper bound ub on the formatter pff. (This function is used for printing of constants in constraints.)

val fprint_in_term : int -> Format.formatter -> t -> unit
    fprint_in_term pff ub is used to print the constant lower bound ub on the formatter pff when it appears in a term, in place of a non-negative variable which has no predecessor.
    An usual implementation may be: let fprint_in_term _ ub = Format.fprintf pff "< %a"
    fprint ub

val draw : t -> string list
end

2.5. Module Dalton_sig.GROUND.Lub : The module Lub provides functions relating lower and upper bounds.

module Lub : sig

val leq : Dalton_sig.GROUND.Lb.t -> Dalton_sig.GROUND.Ub.t -> bool
    geq lb ub returns true if and only if lb is less than or equal to ub, i.e. there exists some alpha such that lb < alpha and alpha < ub.

val geq : Dalton_sig.GROUND.Lb.t -> Dalton_sig.GROUND.Ub.t -> bool
    geq lb ub returns true if and only if lb is greater than or equal to ub, i.e. for all alpha and beta, alpha < ub and lb < beta implies alpha < beta

val fprint_in_term :
    int ->
    Format.formatter ->
    Dalton_sig.GROUND.Lb.t -> Dalton_sig.GROUND.Ub.t -> unit

8
fprint_in_term ppf lb ub is used to print a pair of a lower bound and a upper bound in a term.
An usual implementation may be let fprint_in_term _ ppf lb ub = Format.printf ppf "\n%a \lt %a\n" Lb.fprint lb Ub.fprint ub

end

2.5.7 Module Dalton_sig.GROUND.Label : The set of row labels is defined by the module Label.

module Label : sig

type t

The type of row labels.

val compare : t -> t -> int

A function compare defining a total order on row labels must be provided. This order is used for governing label mutations.

val hash : t -> int

hash lb1 returns a hash integer of the label lb1. If compare lb11 lb12 returns 0 then hash lb11 and hash lb12 must return the same integer.

val fprint : Format.formatter -> t -> unit

fprint ppf lb1 pretty-prints the row label lb1 on the formatter ppf.

end

2.5.8 Module Dalton_sig.GROUND>Type : Type constructors are given by the module Type.

module Type : sig

type 'a t

A type constructor (with its arguments) is represented by a value of type 'a t, where 'a is the type of the arguments.

val ldestr_inv : bool

The boolean constant ldestr_inv tells wether there exists a type constructor for which the left destructor propagates on an invariant argument.

val rdestr_inv : bool

The boolean constant ldestr_inv tells wether there exists a type constructor for which the right destructor propagates on an invariant argument.

val ldestr : 'a t -> bool
ldestr t returns false if the type t cannot be an argument of the left destructor.

val rdestr : 'a t -> bool
  rdestr t returns false if the type t cannot be an argument of the right destructor.

val compatible :
  'a t -> 'a t -> bool
  compatible t1 t2 indicates whether the type constructors t1 and t2 are compatible.

val map :
  (Dalton_aux.constructor_arg -> 'a -> 'b) ->
  'a t -> 'b t
  map f t returns the constructor t' obtained by replacing every son x of t by f i x (where
  i is the information of variance, kind and destructor propagation associated to the argument
  x in t).

val iter :
  (Dalton_aux.constructor_arg -> 'a -> unit) ->
  'a t -> unit
  iter f t applies f on every son x of t.

val iter2 :
  (Dalton_aux.constructor_arg -> 'a -> 'b -> unit) ->
  'a t -> 'b t -> unit
  Given two compatible constructors t1 and t2, iter2 f t1 t2 applies f on every pair of
  corresponding sons of t1 and t2. The result is not specified if t1 and t2 do not correspond.

val map2 :
  (Dalton_aux.constructor_arg -> 'a -> 'b -> 'c) ->
  'a t ->
  'b t -> 'c t
  Given two compatible constructors t1 and t2, map2 f t1 t2... The result is not specified if
  t1 and t2 do not correspond.

val forall2 :
  (Dalton_aux.constructor_arg -> 'a -> 'b -> bool) ->
  'a t -> 'b t -> bool
  Given two compatible constructors t1 and t2, forall f t1 t2 tests whether for all pair of
  sons x1 and x2, f i x1 x2 is true.

val hash : int t -> int
hash t returns a hash integer of the type constructor t which carries hashes of its sons.

type position
Values of type position represents a context of pretty-print. Distinguishing different contexts allows fine parenthesizing of imbricated terms.

val parenthesize :
  position -> 'a t -> bool
  parenthesize pos t returns a boolean indication whether the type constructor t must be parenthesized in context pos.

val fprint :
  Format.formatter ->
  ('a -> bool) ->
  (Dalton_aux.constructor_arg ->
   position -> Format.formatter -> 'a -> unit) ->
  'a t -> unit
  fprint ppf skip f t pretty-prints the type constructor t on the formatter ppf. This function may
  • Apply skip on each of t’s sons. If skip x is true then the son x does not carry any relevant information and may be skipped.
  • Apply f pos ppf x on each of t’s sons in order to pretty-print it.

end

Dalton_sig.PRINT

PRINT

module type PRINT = sig
All printing are performed using the module Format of the Objective Caml standard library. A general purpose instance implementation is provided by Dalton_templates.Print[3.4].

val ghost : string
  The string to be printed in place of ghost variables (i.e. unconstrained anonymous variables), e.g. "_"

val left_destructor : 'a Dalton_aux.printer -> Format.formatter -> 'a -> unit
val right_destructor :
  'a Dalton_aux.printer -> Format.formatter -> 'a -> unit
val left_destructor_skel :
  'a Dalton_aux.printer -> Format.formatter -> 'a -> unit
val right_destructor_skel :
  'a Dalton_aux.printer -> Format.formatter -> 'a -> unit
val same_skel : 'a Dalton_aux.printer -> Format.formatter -> 'a list -> unit
same_skel printer ppf list prints a same-skeleton constraint involving elements of the list list. A printer printer is given as argument for printing each element of the list.

val equal : 'a Dalton_aux.printer -> Format.formatter -> 'a list -> unit

same_skel printer ppf list prints an equality involving elements of the list list. A printer printer is given as argument for printing each element of the list.

leq lhs_printer rhs_printer ppf lhs rhs prints the inequality lhs < rhs on the formatter ppf. Two printers lhs_printer and rhs_printer are provided for printing the left-hand and right-hand sides, respectively.

val leq :
  'a Dalton_aux.printer ->
  'b Dalton_aux.printer -> Format.formatter -> 'a -> 'b -> unit

val lhs : 'a Dalton_aux.printer -> Format.formatter -> 'a list -> unit

lhs printer ppf list prints the left-hand side of an inequality, consisting in the elements of list the list. A printer printer is given as argument for printing each element of the list.

val rhs : 'a Dalton_aux.printer -> Format.formatter -> 'a list -> unit

rhs printer ppf list prints the right-hand side of an inequality, consisting in the elements of list the list. A printer printer is given as argument for printing each element of the list.

val cset_begin : Format.formatter -> unit

cset_begin ppf is called before printing a constraint set on the formatter ppf.

val cset_end : Format.formatter -> unit

cset_end ppf is called at the end of the printing a constraint set on the formatter ppf.

val cset_item : 'a Dalton_aux.printer -> Format.formatter -> 'a -> unit

Every constraint c of a constraint set is printed on the formatter ppf by a call of the form cset_item printer ppf c where printer is a suitable printer for the constraint.

end

Dalton_sig.ERROR_REPORT
ERROR_REPORT

module type ERROR_REPORT = sig
  A general purpose instance implementation is provided by Dalton_templates.ErrorReport[3.6].
2.7.1 Unification errors

val unification : Format.formatter ->
term1:Dalton_aux.printing ->
term2:Dalton_aux.printing -> explanation:Dalton_aux.printing -> unit
  unification ppf ~term1 ~term2 ~explanation reports an unification failure of the terms
term1 and term2. explanation gives a short explanation of the reason of the failure,
generated itself by one of the functions cycle or incompatible.

val cycle : Format.formatter ->
variable:Dalton_aux.printing -> term:Dalton_aux.printing -> unit
  cycle ppf variable term prints the explanation of an unification failure due to the
  occur-check.

val incompatible : Format.formatter ->
term1:Dalton_aux.printing -> term2:Dalton_aux.printing -> unit
  cycle ppf ~term1 ~term2 prints the explanation of an unification failure due to
  incompatibles type constructors.

2.7.2 Constraints solving errors

val ldestr : Format.formatter -> term:Dalton_aux.printing -> unit
  ldestr ppf ~term tells that the left-destructor has been applied on the type term term
  which cannot be so.

val rdestr : Format.formatter -> term:Dalton_aux.printing -> unit
  rdestr ppf ~term tells that the right-destructor has been applied on the type term term
  which cannot be so.

val inequality :
  Format.formatter ->
  lb:Dalton_aux.printing -> ub:Dalton_aux.printing -> unit
  inequality ppf ~lb ~ub tells that a constraint is not satisfiable because it implies the
  incorrect inequality lb < ub between constant bounds.
2.7.3 Schemes comparison

val incompatible_schemes :
Format.formatter ->
scheme1:Dalton_aux.printing ->
scheme2:Dalton_aux.printing -> explanation:Dalton_aux.printing -> unit
    incompatible_schemes ppf "scheme1""scheme2""explanation reports that schemes
    scheme1 and scheme2 are not comparable, because of an unification error. explanation
    gives a short explanation of the reason of the failure, generated itself by one of the functions
    cycle or incompatible above.

val missing_desc :
Format.formatter ->
scheme:Dalton_aux.printing ->
variable:Dalton_aux.printing -> term:Dalton_aux.printing -> unit
    missing_desc ppf "scheme""variable""term reports a comparison failure due to a
    variable instantiation.

val missing_constraint :
Format.formatter ->
scheme:Dalton_aux.printing -> constrain:Dalton_aux.printing -> unit
    missing_desc ppf "scheme""variable""term reports a comparison failure due to a
    missing inequality.

val missing_bound :
Format.formatter ->
scheme:Dalton_aux.printing ->
constrain:Dalton_aux.printing ->
explanation:Dalton_aux.printing option -> unit
    missing_desc ppf "scheme""variable""term reports a comparison failure due to a
    missing constant bound.

2.7.4 Minimal instance

val minimal :
Format.formatter ->
scheme:Dalton_aux.printing -> variables:Dalton_aux.printing -> unit
    minimal ppf "scheme""variables" prints a message telling that the type scheme scheme
    has no minimal instance. variables prints a list of the variables of the scheme which do not
    have a minimal solution.

end
module type DRAW = sig
    This allows performing drawing on a variety of device using appropriate external libraries.
    An example implementation using the Graphics library of the Objective Caml system is given
    in Dalton_templates.DrawGraphics[3.5].

type window
val draw_lines : window ->
    color:Dalton_aux.color -> lw:int -> (int * int) list -> unit
val draw_rect : window ->
    color:Dalton_aux.color ->
    lw:int -> x:int -> y:int -> w:int -> h:int -> unit
val draw_ellipse : window ->
    color:Dalton_aux.color ->
    lw:int -> x:int -> y:int -> rx:int -> ry:int -> unit
val fill_rect : window ->
    color:Dalton_aux.color -> x:int -> y:int -> w:int -> h:int -> unit
val fill_ellipse : window ->
    color:Dalton_aux.color -> x:int -> y:int -> rx:int -> ry:int -> unit
val fill_poly : window ->
    color:Dalton_aux.color -> (int * int) list -> unit
val draw_text : window ->
    color:Dalton_aux.color ->
    ?name:string -> size:int -> x:int -> y:int -> string -> unit
val text_size : window -> ?name:string -> size:int -> string -> int * int
val draw_dotted_lines : window ->
    color:Dalton_aux.color -> (int * int) list -> unit
end

3 Module Dalton_templates: Templates of module parameters.

This unit provides templates of modules which may be used as argument for the solver’s functor.
module Print : Dalton_sig.PRINT
  [3.4]

module DrawGraphics : sig end
  [3.5]

module ErrorReport : Dalton_sig.ERROR_REPORT
  [3.6]

  Dalton_templates.Print
       Print

module Print : sig

val ghost : string
val leftDestructor : 
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a -> unit
val rightDestructor : 
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a -> unit
val leftDestructor_skel : 
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a -> unit
val rightDestructor_skel : 
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a -> unit
val print_list : 
  (Format.formatter -> 'a -> unit) -> 
  string -> Format.formatter -> 'a list -> unit
val same_skel : 
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a list -> unit
val equal : 
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a list -> unit
val leq : 
  (Format.formatter -> 'a -> unit) -> 
  (Format.formatter -> 'b -> unit) -> Format.formatter -> 'a -> 'b -> unit
val lhs : 
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a list -> unit
val rhs : 
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a list -> unit
val first : bool Pervasives.ref
val cset_begin : 'a -> unit
val cset_item :
    (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a -> unit
val cset_end : Format.formatter -> unit
end

Dalton_templates.DrawGraphics

module DrawGraphics : sig

  type window = unit

  val draw_lines :
    unit -> color:Graphics.color -> lw:int -> (int * int) list -> unit

  val draw_curves :
    unit ->
    color:Graphics.color ->
    lw:int ->
    int * int -> ((int * int) * (int * int) * (int * int)) list -> unit

  val draw_rect :
    unit ->

  val draw_ellipse :
    unit ->
    color:Graphics.color ->
    lw:int -> x:int -> y:int -> rx:int -> ry:int -> unit

  val fill_rect :
    unit ->
    color:Graphics.color -> x:int -> y:int -> w:int -> h:int -> unit

  val fill_ellipse :
    unit ->
    color:Graphics.color -> x:int -> y:int -> rx:int -> ry:int -> unit

  val fill_poly :
    unit -> color:Graphics.color -> (int * int) list -> unit

  val draw_text :
    unit ->
    color:Graphics.color ->
    ?name:string -> size:int -> x:int -> y:int -> string -> unit

  val text_size :
    unit ->
    ?name:string -> size:int -> string -> int * int

  val draw_dotted_lines :
    'a -> color:Graphics.color -> (int * int) list -> unit
end

Dalton_templates.ErrorReport

module ErrorReport : sig

end

17
val unification : 
  Format.formatter ->
term1:(Format.formatter -> unit) ->
term2:(Format.formatter -> unit) ->
explanation:(Format.formatter -> unit) -> unit
val cycle : 
  Format.formatter ->
  variable:(Format.formatter -> unit) ->
term:(Format.formatter -> unit) -> unit
val incompatible : 
  Format.formatter ->
term1:(Format.formatter -> unit) ->
term2:(Format.formatter -> unit) -> unit
val minimal : 
  Format.formatter ->
scheme:(Format.formatter -> unit) ->
variables:(Format.formatter -> unit) -> unit
val ldestr : Format.formatter -> term:(Format.formatter -> unit) -> unit
val rdestr : Format.formatter -> term:(Format.formatter -> unit) -> unit
val inequality : 
  Format.formatter ->
lb:(Format.formatter -> unit) -> ub:(Format.formatter -> unit) -> unit
val incompatible_schemes : 
  Format.formatter ->
scheme1:(Format.formatter -> unit) ->
scheme2:(Format.formatter -> unit) ->
explanation:(Format.formatter -> unit) -> unit
val missing_desc : 
  Format.formatter ->
scheme:(Format.formatter -> unit) ->
variable:(Format.formatter -> unit) ->
term:(Format.formatter -> unit) -> unit
val missing_constraint : 
  Format.formatter ->
scheme:(Format.formatter -> unit) ->
constraint:(Format.formatter -> unit) -> unit
val missing_bound : 
  Format.formatter ->
scheme:(Format.formatter -> unit) ->
constraint:(Format.formatter -> unit) ->
explanation:(Format.formatter -> unit) option -> unit
end
4 Module Dalton: The core of the library

module Make : functor (Ground : Dalton_sig.GROUND) ->
  functor (Print : Dalton_sig.PRINT) ->
  functor (Draw : Dalton_sig.DRAW) ->
    functor (Report : Dalton_sig.ERROR_REPORT) -> sig end

Dalton.Make

module Make : sig
  parameters:
    • Ground : Dalton_sig.GROUND
    • Print : Dalton_sig.PRINT
    • Draw : Dalton_sig.DRAW
    • Report : Dalton_sig.ERROR_REPORT

4.1.1 Constraint sets

type cset
  Constraint sets are represented by values of type cset.

val cset : unit -> cset
  Each invocation of cset () returns a new fresh empty constraint set.

val merge_cset : cset -> cset -> unit
  merge_cset cs1 cs2 merges the two constraint sets cs1 and cs2. After invoking this
  function, cs1 and cs2 point to the same constraint set cs which corresponds to the
  conjunction of the previous cs1 and cs2.

4.1.2 Terms

type node
  (Multi-equations of) Terms are represented by values of type node.

val variable : cset -> Dalton_aux.kind -> node
  variable cs k returns a fresh variable term of kind k. This variable may be used in the
  constraint set cs.
val variable_in_sk : node -> node
    variable_in_sk nd returns a fresh variable belonging to the same skeleton (and the same
    constraint set) as the node nd.

val typ :
    cset -> node Ground.Type.t -> node
    typ cs t returns a fresh type term described by the type constructor t in the constraint set
    cs. Every son of t must be a node belonging to cs.

val row :
    cset ->
    Ground.Label.t * node * node -> node
    row cs (lb1, nd_lb1, nd’) returns a fresh row node representing the term lb1:
    nd_lb1, nd’ in the constraint set cs. nd_lb1 and nd’ must both belong to cs.

4.1.3 Setting constraints

type skeleton
    Multi-skeletons are represented by values of type skeleton.

type node_or_skeleton =
    | Nd of node
    | Sk of skeleton
    A value of type node_or_skeleton is either a node or a skeleton. Such values are used to
    represent weak inequalities.

type unification_report
    Unification errors are described by a value of type unification_report. The
    implementation of this type is not described. As a result, reports may be used only in order
    to print error messages thanks to the function report_unification.

exception UnificationError of unification_report
    Above functions report unification errors by raising an exception UnificationError with an
    appropriate report as argument.

val report_unification :
    Format.formatter -> unification_report -> unit
    report_unification ppf r pretty prints an error message on the formatter ppf describing
    the unification error reported by r.

val same_skel : node -> node -> unit
same_skel nd1 nd2 sets a constraint nd1 = nd2. nd1 and nd2 must be nodes of the same constraint set and the same kind. If setting this constrain entails an unification error, an exception UnificationError is raised.

val equal : node -> node -> unit
  equal nd1 nd2 sets a constraint nd1 = nd2. nd1 and nd2 must be nodes of the same constraint set and the same kind. If setting this constrain entails an unification error, an exception UnificationError is raised.

val strong_leq : node -> node -> unit
  strong_leq nd1 nd2 sets the strong inequality ns1 < ns2. nd1 and nd2 must be nodes of the same constraint set and the same kind. If setting this constrain entails an unification error, an exception UnificationError is raised.

val weak_leq :
  node_or_skeleton -> node_or_skeleton -> unit
  weak_leq ns1 ns2 sets a weak inequality ns1 < ns2. ns1 and ns2 must be nodes or skeletons of the same constraint set.

val lower_bound : Ground.Lb.t -> node_or_skeleton -> unit
  lower_bound lb ns sets the weak inequality lb < ns.

val upper_bound : node_or_skeleton -> Ground.Ub.t -> unit
  upper_bound ns ub sets the weak inequality ns < ub.

4.1.4 Substitutions

type subst = {
  lb : Ground.Lb.t -> Ground.Lb.t ;
    The substitution applied on lower bounds appearing in the constraint set.
  ub : Ground.Ub.t -> Ground.Ub.t ;
    The substitution applied on upper bounds appearing in the constraint set.
  typ : ?a. ?a Ground.Type.t -> ?a Ground.Type.t ;
    The substitution applied on type constructors.
  label : Ground.Label.t -> Ground.Label.t ;
    The substitution applied on row labels.
}
A substitution may be applied while copying a scheme. It is defined by a record of four functions of type subst.

4.1.5 Schemes

module type SCHEME_ROOT = sig end

module Scheme : functor (Root : SCHEME_ROOT) -> sig end

4.1. Module type Dalton.Make.SCHEME_ROOT : A (type) scheme is made of a constraint set and a series of entry nodes, its roots.

module type SCHEME_ROOT = sig
  The same instance of the library may deal with several form of schemes. Each of them has to be described by an implementation of the signature SCHEME_ROOT.

type t
  The type of schemes.

val cset : t -> Dalton.Make.cset
  cset sh returns the constraint set of the scheme sh.

val copy :
  Dalton.Make.cset ->
  (Dalton.Make.node -> Dalton.Make.node) ->
  t -> t
  copy cset' f sh creates a new scheme sh' as follows:
  • the constraint set of sh' is cset',
  • each root of sh' is obtained by applying f on the corresponding root of sh.

val iter :
  (Dalton_aux.variance -> Dalton.Make.node -> unit) ->
  t -> unit
  iter f sh applies f on every root of sh (with the variance of the root as first argument).

val iter2 :
  (Dalton_aux.variance -> Dalton.Make.node -> Dalton.Make.node -> unit) ->
  t -> t -> unit
  iter2 f sh1 sh2 applies f on every pair of corresponding roots of sh1 and sh2 (with the variance of the roots as first argument).
val fprint :
  Format.formatter ->
  Dalton.Make.cset Dalton_aux.printer ->
  (Dalton_aux.variance -> Format.formatter -> Dalton.Make.node -> unit) ->
  t -> unit
  fprint ppf print_cset print_node sh pretty prints the scheme sh on the formatter ppf.
  Two functions are provided as argument to allow printing of information handled by the
  solver:
  • print_cset ppf cset prints the constraint set cset on the formatter ppf
  • print_node v ppf nd prints the node nd of variance v on the formatter ppf.

end

4.1.7 Module Dalton.Make.Scheme : The functor scheme allows to build an implement-
  ation of functions dealing which each considered form schemes.

module Scheme : sig
  parameters:
  • Root : Dalton.Make.SCHEME_ROOT

val copy : ?subst: Dalton.Make.subst -> Root.t -> Root.t
  copy ?subst sh returns a fresh copy of the type scheme sh. No particular assumption is
  made about the type scheme sh, but, for efficiency, it is more than a good idea to solve it
  previously.

val fprint : Format.formatter -> Root.t -> unit
  fprint ppf sh pretty-prints the scheme sh on the formatter ppf. The scheme sh is
  assumed to be solved.

val draw : Draw.window -> Root.t -> int -> int -> int * int
  draw window sh x y draws the scheme sh on the window window. The bottom left corner of
  the drawing has coordinates x and y and the function returns the coordinates of the upper
  right corner.

4.1.8 Resolution

type solve_report
  A solve report records an explanation of why the resolution of a scheme fails.

val report_solve : Format.formatter -> solve_report -> unit
report_solve ppf r pretty prints an error message on the formatter ppf corresponding to the comparison report r.

val solve : Root.t -> solve_report option

solve sh solves the scheme sh. If this function returns None then the scheme sh has some instances. Moreover, it is stored in a "solved" form which is preserved as long as no term or constraint is added to its constraint set.

4.1.9 Comparison
type comparison_report

A comparison report records an explanation of the failure of the comparison of two schemes.

val report_comparison :
Format.formatter -> comparison_report -> unit

report_comparison ppf r pretty prints an error message on the formatter ppf describing the comparison report r.

val compare : Root.t -> Root.t -> comparison_report option

compare sh1 sh2 test whether sh2 is more general than sh1 (i.e. sh2 is a correct implementation of sh1). It returns None if sh2 is effectively so. Otherwise, it returns Some r when r is a report "explaining" why sh2 is not more general than sh1. The current implementation assumes that sh1 and sh2 are solved.

val equivalent : Root.t -> Root.t -> bool

equivalent sh1 sh2 returns a boolean indicating whether the type schemes sh1 and sh2 are equivalent. The current implementation assumes that sh1 and sh2 are in solved form.

4.1.10 Minimal instances
type minimal_report

A comparison report records an explanation of why a scheme has no minimal instance.

val report_minimal :
Format.formatter -> minimal_report -> unit

report_minimal ppf r pretty prints an error message on the formatter ppf describing the report r.

val has_minimal_instance : Root.t -> minimal_report option
has_minimal_instance $sh$ tests whether the scheme $sh$ has a minimal instance. If so, the function returns None. Otherwise, it returns Some $r$ where $r$ is a value of type minimal_report "explaining" why $sh$ has no minimal instance. The current implementation assumes that $sh$ is in solved form.