Technical Report

Tyrolean Complexity Tool

Martin Avanzini and Georg Moser and Andreas Schnabl

17 November 2008

Abstract

This document provides the module documentation of TcT.

Contents

1 Strategies **1**

1.1 Module `CPstrategies` : Strategy Wrappers for complexity methods . . . 1
1.2 Module `ConditionMonade` . . . 2
1.3 Module `ConditionState` . . . 3
1.4 Module `ConditionWrapper` : This is the module conditionWrapper of TTT2. . . . 3
1.5 Module `StrategyMonade` . . . 3
1.6 Module `StrategyState` . . . 4
1.7 Module `Strategy` : This module, taken from TTT2, is used to execute a strategy given by the `-s` command line argument. . . . 4
1.8 Module `Proof` : This module defines proof objects, and is based on its brother-in-names in TTT2. . . . 6
1.9 Module `TerminationWrapper` : This is the module terminationWrapper of TTT2. . . . 9

2 Conditions **9**
2.1 Module `Nonincreasing` : Checks whether a TRS is not size increasing 9

3 Context Dependent Interpretations and Matrix Interpretations **9**
3.1 Module `Cdinterpretation` : Creation of constraints for abstract polynomial and context dependent interpretations 9
3.2 Module `Cdiparametric` : Creation of abstract polynomial and context dependent interpretations 10
3.3 Module `Cdipoly` : Polynomials for polynomial and context dependent interpretations 11
3.4 Module `Cdisolve` : Generation of SAT Encoding 14
3.5 Module `Mcdiparametric` : Creation of abstract matrix interpretations 15
3.6 Module `Mcdipoly` : Creation of standard and context dependent matrix interpretations 16
3.7 Module `Mcdiprove` : Creation of matrix interpretations 20
3.8 Module `Mcdisolve` : Generation of SAT Encoding 21
3.9 Module `Mcdinterpretation` : Creation of constraints for abstract matrix interpretations 21
3.10 Module `Cdiprove` : Creation of polynomial and context dependent interpretations 22

4 POP* **24**
4.1 Module `Pop` : POP* with optionally semantic labeling and POP* as reduction pair 24
4.2 Module `PopSL` : POP* with and without finite semantic labeling. 24
4.3 Module `PopWidp` : POP* as reduction pair 24

5 Propositional Logic **25**
5.1 Module `PLogx` : Extensions to PLog 25
5.2 Module `LazyPLog` : Lazy processing of propositonal formulas . . 27
5.3 Module `VarCache` : Caching of propositional variables 28
5.4 Module `ArgumentFilteringCache` : Propositional encodings of argument filters 31
5.5 Module `PrecedenceCache` : This module provides a variable cache for the encoding of quasi- and strict precedences. 32
5.6 Module `SafeMappingCache` : Propositional encoding of safe mappings 32
5.7 Module `FiniteFnEncoding` : Propositional encoding of functions with finite carrier 33
5.8 Module `FiniteModel` : Function symbols with abstract labels . . 34
5.9 Module `ProperOrderedSet` : Encoding of proper ordered sets . . 36

6 Dependency Pairs **37**
6.1 Module `Wdp` : Weak Dependency Pairs 37
6.2 Module `Wdg` : Weak Dependency graph approximations 39

6.3 Module `Maxpath` : Maximal Path 40
6.4 Module `Maxpaths` : Maximal Paths 40

7 Utilities **40**
7.1 Module `Listxx` : Extended Extended lists 40
7.2 Module `Setxx` : Extended Setx 41
7.3 Module `Graphx` : Extended Graph 41
7.4 Module `Termxx` : Extended Term 42
7.5 Module `Rulex` : Extended Rule 42
7.6 Module `Trsx` : Extended TRS 43
7.7 Module `VariadicFunctionSymbol` : Function Symbols with multiple arities. 43
7.8 Module `Algebra` : Set of interpretations over a signature. 44
7.9 Module `Assign` : This module provides an implementation for assignments from variables to a specified carrier. 44
7.10 Module `CPutils` : Standard utility functions 47
7.11 Module `Debug` : when `debugmessages = true` debugging messages are emitted 47
7.12 Module `PredicativeRewriting` : Utility functions for predicative systems 47
7.13 Module `SafeMapping` : Implementation of safe mappings 47

1 Strategies

1.1 Module `CPstrategies` : Strategy Wrappers for complexity methods

Author(s): Andreas Schnabl, Martin Avanzini

This module contains uniform wrapper functions for calling various complexity strategies.

```
exception Configuration_failure of string
```

```
val cp_cdiprover :
  string list -> Data.t -> StrategyState.t -> (Data.t * StrategyState.t) list
```

`cp_cdiprover flags _ state` applies a polynomial or context dependent interpretation to the current termination problem in `state` using `flags` as options for the method. If neither the `-sdp` nor the `-nosdp` flag is given, then the methods makes this choice own its own based on whether the current problem is a direct termination problem, or a DP problem.

```
val cp_maxpaths :
  string list -> Data.t -> StrategyState.t -> (Data.t * StrategyState.t) list
```

`cp_popstar flags data state` decomposes the current termination problem into several smaller problems, according to the paths in the current dependency graph. The method is applied in `state` using `flags` as options for the method.

```
val cp_mcdi :
  string list -> Data.t -> StrategyState.t -> (Data.t * StrategyState.t) list
```

`cp_mcdi flags _ state` applies a matrix interpretation to the current termination problem in `state` using `flags` as options for the method. If neither the `-sdp` nor the `-nosdp` flag is given, then the methods makes this choice own its own based on whether the current problem is a direct termination problem, or a DP problem.

```
val cp_popstar :
  string list -> Data.t -> StrategyState.t -> (Data.t * StrategyState.t) list
```

`cp_popstar flags _ state` applies the pop* method to the current termination problem in `state` using `flags` as options for the method.

```
val cp_smc :
  string list -> Data.t -> StrategyState.t -> (Data.t * StrategyState.t) list
```

`cp_smc flags _ state` applies the smc method to the current termination problem in `state` using `flags` as options for the method.

```
val cp_wdg :
  (Trs.t -> Trs.t -> Wdg.proof) ->
  string list -> Data.t -> Data.t StrategyMonade.t
```

`cp_wdg f _ _ state` creates the dependency graph for the current termination problem in `state` using using the method `f` for estimating the graph.

```
val cp_wdp :
  string list -> Data.t -> StrategyState.t -> (Data.t * StrategyState.t) list
```

`cp_wdp flags _ state` applies the direct WDP method to the current termination problem in `state` using `flags` as option for the method.

```
val cp_wdp_rp :
  string list -> Data.t -> StrategyState.t -> (Data.t * StrategyState.t) list
```

`cp_wdp_rp flags _ state` applies the WDP method to the current termination problem in `state` using `flags` as options for the method, turning the given direct problem into a DP problem. Moreover, the method tries to find a suitable SLI for the usable rules.

```
val cp_widp :
  string list -> Data.t -> StrategyState.t -> (Data.t * StrategyState.t) list
```

`cp_widp flags _ state` applies the direct WIDP method to the current termination problem in `state` using `flags` as options for the method.

```
val cp_widp_rp :
  string list -> Data.t -> StrategyState.t -> (Data.t * StrategyState.t) list
```

`cp_widp_rp flags _ state` applies the WIDP method to the current termination problem in `state` using `flags` as options for the method, turning the given direct problem into a DP problem. Moreover, the method tries to find a suitable SLI for the usable rules.

1.2 Module `ConditionMonade`

Author(s): Martin Korp

Since: Wed Jan 2 12:45:12 CET 2008

Description: State monade used to execute conditional methods. AS: Needed as source for compilation of TCT, because we want it to reference the ConditionState module referencing the new proof module, not the one referencing the old proof module of TTT2

```
type 'a t = ConditionState.t -> 'a * ConditionState.t
val conjunction : 'a t -> ('a -> 'a t) -> 'a t
val disjunction : 'a t -> 'a t -> 'a t
val negation : 'a t -> 'a t
val return : 'a -> 'a t
val run : 'a t -> 'a t
```

1.3 Module ConditionState

Author(s): Martin Korp

Since: Wed Jan 2 12:58:24 CET 2008

Description: Defines the state used by the condition monade. AS: Needed as source for compilation of TCT, because we want it to reference the new proof module, not the old one of TTT2

```
type t
val default : t
val get_proof : t -> Proof.proof
val get_system : t -> System.t
val get_value : t -> bool
val set_proof : Proof.proof -> t -> t
val set_system : System.t -> t -> t
val set_value : bool -> t -> t
```

1.4 Module ConditionWrapper : This is the module conditionWrapper of TTT2.

We have shortened the module to only include the methods of TTT2 which we use directly.

```
exception Configuration_failure of string
val c_duplicating : string list -> unit -> unit ConditionMonade.t
val c_constructor : string list -> unit -> unit ConditionMonade.t
val c_left_linear : string list -> unit -> unit ConditionMonade.t
```

1.5 Module StrategyMonade

Author(s): Martin Korp

Since: Wed Jan 2 13:29:40 CET 2008

Description: State/List monade used to execute strategies. AS: Needed as source for compilation of TCT, because we want it to reference the StrategyState module referencing the new proof module, not the one referencing the old proof module of TTT2

```
type 'a t = StrategyState.t -> ('a * StrategyState.t) list
val composition : 'a t -> ('a -> 'a t) -> 'a t
val application : 'a t -> ('a -> 'a t) -> 'a t
val condition : 'a ConditionMonade.t ->
  'b t -> 'b t -> 'b t
```

```
val fail_execution : 'a t -> 'a t
val numbered_iteration : int -> ('a -> 'a t) -> 'a -> 'a t
val option : 'a t -> 'a t -> 'a t
val parallel_option : 'a t -> 'a t list -> 'a t
val return : 'a -> 'a t
val run : 'a t -> 'a t
val smooth_execution : ('a -> 'a t) -> 'a -> 'a t
val smooth_iteration : ('a -> 'a t) -> 'a -> 'a t
val strict_execution : 'a t -> 'a t
val strict_iteration : ('a -> 'a t) -> 'a -> 'a t
val timed_execution : float -> 'a t -> 'a t -> 'a t
val timed_iteration : float -> ('a -> 'a t) -> 'a -> 'a t
```

1.6 Module `StrategyState`

Author(s): Martin Korp

Since: Wed Jan 2 13:49:40 CET 2008

Description: Defines the state used by the strategy monade. AS: Needed as source for compilation of TCT, because we want it to reference the new proof module, not the old one of TTT2

```
type t
val default : t
val get_edg : t -> Rule.t Graph.t option
val get_name_supply : t -> NameSupply.t
val get_proof : t -> Proof.proof
val get_root_labeling_index : t -> FunctionSymbol.t list Index.t
val get_status : t -> Status.t
val get_system : t -> System.t
val set_edg : Rule.t Graph.t -> t -> t
val set_name_supply : NameSupply.t -> t -> t
val set_proof : Proof.proof -> t -> t
val set_root_labeling_index : FunctionSymbol.t list Index.t -> t -> t
val set_status : Status.t -> t -> t
val set_system : System.t -> t -> t
```

1.7 Module `Strategy` : This module, taken from TTT2, is used to execute a strategy given by the `-s` command line argument.

For the grammar of strategies, we refer to the documentation of TTT2. We have changed the list of available strategies in this module in order to suit the needs o complexity proving.

1.7 Module `Strategy` : This module, taken from TTT2, is used to execute a strategy given by the `-s` command line argument.

```
val get_strategies : unit -> string list
```

`get_strategies ()` returns a list of available strategies.

```
val rt_of_channel : Pervasives.in_channel -> Trs.t * Trs.t -> Proof.t
```

`rt_of_channel c` parses a strategy from the channel `c`.

```
val rt_of_lexbuf : Lexing.lexbuf -> Trs.t * Trs.t -> Proof.t
```

`rt_of_lexbuf b` same as `Strategy.st_of_channel`[1.7] with the difference that the strategy is parsed from the lexer-buffer `b`.

```
val rt_of_string : string -> Trs.t * Trs.t -> Proof.t
```

`rt_of_string s` same as `Strategy.st_of_channel`[1.7] with the difference that the strategy is parsed from the string `s`. This function supports also abbreviations for different strategies.

```
val set_abbreviations : (string * string) list -> unit
```

`set_abbreviations abr` sets the used abbreviations.

```
val st_of_channel : Pervasives.in_channel -> Trs.t -> Proof.t
```

`st_of_channel c` parses a strategy from the channel `c`.

```
val st_of_lexbuf : Lexing.lexbuf -> Trs.t -> Proof.t
```

`st_of_lexbuf b` same as `Strategy.st_of_channel`[1.7] with the difference that the strategy is parsed from the lexer-buffer `b`.

```
val st_of_string : string -> Trs.t -> Proof.t
```

`st_of_string s` same as `Strategy.st_of_channel`[1.7] with the difference that the strategy is parsed from the string `s`. This function supports also abbreviations for different strategies.

```
val tt_of_channel : Pervasives.in_channel -> Trs.t -> Proof.t
```

`tt_of_channel c` parses a strategy from the channel `c`.

```
val tt_of_lexbuf : Lexing.lexbuf -> Trs.t -> Proof.t
```

`tt_of_lexbuf b` same as `Strategy.st_of_channel`[1.7] with the difference that the strategy is parsed from the lexer-buffer `b`.

```
val tt_of_string : string -> Trs.t -> Proof.t
```

`tt_of_string s` same as `Strategy.st_of_channel`[1.7] with the difference that the strategy is parsed from the string `s`. This function supports also abbreviations for different strategies.

1.8 Module `Proof` : This module defines proof objects, and is based on its brother-in-names in TTT2.

See the documentation of TTT2 for a description of the functionality of that initial proof module. The types of the proof objects have been changed from the termination proof types used in TTT2 to proof types used for estimating derivational or runtime complexity of TRSs. Moreover, we have added functionality for extracting complexity bounds from proofs of the types specified in this module.

```
type condition =
  | C_con of Constructor.proof
  | C_dup of Duplicating.proof
  | C_llin of LeftLinear.proof
```

This type defines possible condition proof objects.

```
type termination =
  | T_bounds of BoundsStrategy.proof
  | T_rlab of RootLabeling.proof
```

This type defines possible termination proof objects.

```
type complexity =
  | CP_popstar of Pop.proof
  | CP_cdiprover of Cdiprove.proof
  | CP_mcdi of Mcdiprove.proof
  | CP_maxpath of Maxpath.proof
  | CP_maxpaths of Maxpaths.proof
  | CP_wdg of Wdg.proof
  | CP_wdp of Wdp.proof
  | CP_wdprp of Wdp.redpairproof
  | CP_widp of Wdp.proof
  | CP_widprp of Wdp.redpairproof
```

This type defines possible complexity proof objects.

`type proof`

This type defines the structure of termination and non-termination proof objects.

`type t`

This type defines a proof. So far only proof objects for relative termination, standard termination and termination modulo theory are supported.

```
type complexitykind =
  | DC
  | RC
  | IDC
  | IRC
```

1.8 Module `Proof` : This module defines proof objects, and is based on its brother-in-names in TTT2.

This type defines the kinds of complexity proofs we consider. We distinguish proofs of derivational complexity and runtime complexity, both for either standard rewriting, or innermost rewriting.

```
type complexityclass =
  | Specpoly of int
  | Polynomial
  | Exponential
  | Doubleexp
  | Primrec
  | Unknown
```

This type defines the complexity classes we consider.

```
type answer =
  | No
  | Yes
  | Maybe
```

This type defines the possible outcomes of a proof object. `No` stands for non-termination and `Yes` for termination. `Maybe` is used to indicate that a proof has not been finished completely.

```
val answer : t -> answer
```

`answer t` returns `Yes` if the underlying TRS has been proved terminating, `No` if the underlying TRS has been proved non-terminating and `Maybe` if the proof is not finished.

```
val append : proof -> proof -> proof
```

`append p q` replaces `Open` in `q` with `p`. If a `Fail` or `Qed` exists instead of a `Open`, the expetion `Invalid_argument "Proof.append"` is raised.

```
val create_c : condition -> proof -> proof
```

`create_c data p` creates a condition proof.

```
val create_cp : complexity -> proof list -> proof
```

`create_cp data ps` creates a complexity proof.

```
val create_fail : proof
```

`create_fail` creates the proof `Fail`.

`create_n data ps` creates a non-termination proof.

```
val create_open : proof
```

`create_open` creates the proof `Open`.

```
val create_qed : proof
```

`create_qed` creates the proof `Qed`.

`create_r data ps` creates a transformation proof.

```
val create_rt : Trs.t * Trs.t -> proof -> t
```

`create_rt data p` creates a relative termination proof.

```
val create_st : Trs.t -> proof -> t
```

`create_st data p` creates a standard termination proof.

```
val create_t : termination -> proof list -> proof
```

`create_t data ps` creates a termination proof.

```
val create_tt : Trs.t -> proof -> t
```

`create_tt data p` creates a theory termiantion proof.

```
val fprintf : Format.formatter -> t -> unit
```

`fprintf ppf p` pretty prints `p`.

```
val fprintf_answer : Format.formatter -> answer -> unit
```

`fprintf_answer ppf a` pretty prints `a`.

```
val fprintf_complexity : bool -> Format.formatter -> complexityclass -> unit
```

`fprintf_complexity b ppf c` pretty prints `c`. When the boolean flag b is specified, then only polynomial bounds are printed, higher bounds result in a MAYBE output.

```
val get_complexity : t -> complexitykind -> complexityclass
```

`get_complexity p c` returns the derivational, runtime or computational complexity, depending on the value of c, that can be inferred from p.

```
val is_fail : proof -> bool
```

`fail p` returns `true` if at least one leaves of the proof could not be proved completely (indicated by `Fail`). Otherwise `false` is returned.

```
val is_open : proof -> bool
```

`open p` returns `true` if at least one leaves of the proof has not been proved completely (indicated by `Open`). Otherwise `false` is returned.

```
val is_qed : proof -> bool
```

`qed p` returns `true` if all leaves of the proof are closed (indicated by `Qed`). Otherwise `false` is returned.

```
val merge : proof list -> proof
```

`merge ps` merges the proofs of `ps`.

```
val to_string : t -> string
```

`to_string p` transforms the proof `p` into a string.

1.9 Module TerminationWrapper : This is the module terminationWrapper of TTT2.

We have shortened the module to only include the methods of TTT2 which we use directly.

```
exception Configuration_failure of string
val t_bounds : string list -> Data.t -> Data.t StrategyMonade.t
val t_fail : string list -> Data.t -> Data.t StrategyMonade.t
val t_root_lab : string list -> Data.t -> Data.t StrategyMonade.t
val t_success : string list -> Data.t -> Data.t StrategyMonade.t
```

2 Conditions

2.1 Module Nonincreasing : Checks whether a TRS is not size increasing

Author(s): Andreas Schnabl

This is a condition which is true if for all rules, the right hand side is not bigger than the left hand side.

```
type proof = bool * Trs.t
val check : Trs.t -> bool * Trs.t
```

`check trs` returns a pair which contains in the first part a boolean value indicating whether `trs` is nonincreasing, and `trs` in the second part.

```
val fprintf : Format.formatter -> bool * Trs.t -> unit
```

`fprintf ppf data` pretty-prints out `data` (which is expected to be the result of a call to `check`) to the formatter `ppf`.

3 Context Dependent Interpretations and Matrix Interpretations

3.1 Module Cdinterpretation : Creation of constraints for abstract polynomial and context dependent interpretations

Author(s): Sarah Winkler (for polynomial interpretations), Andreas Schnabl (for context dependent interpretations)

Creation of constraints for abstract polynomial and context dependent interpretations

Cdinterpretation is responsible for creating constraints on the possible instantions of polynomial and context dependent interpretations.

```
type t = string * (Cdipoly.poly * (Cdipoly.variable * Cdipoly.poly) list)
```

```
val coefficient_inequalities_of_trs :
  Trs.t ->
  t list ->
  bool ->
  bool ->
  FunctionSymbol.t Setx.t ->
  bool ->
  Cdipoly.diophantinelistconstraint list *
  Cdipoly.diophantinelistconstraint list list
```

`coefficient_inequalities_of_trs trs inter is_ftaui is_quasi fs is_relative` creates a list of inequalities from a TRS `trs` and a polynomial or context dependent interpretation `inter` which ensure the according monotonicity and compatibility properties. The boolean argument `is_ftaui` specifies whether the given interpretation is polynomial or context dependent. Setting the flag `is_quasi` specifies that the rules of `trs` should be weakly oriented. The set `fs` gives the list of function symbols whose interpretation functions have to be strictly monotone - for all function symbols not in that set, only weak monotonicity is enforced. The flag `is_relative` specifies that one of the rules in `trs` has to be weakly oriented (there is still an error somewhere in this mode, so please do not set this flag to true yet). The function returns two lists of diophantine constraints: the first list contains inequalities which absolutely have to be satisfied, while the second part is a list of lists of inequalities, of which only one list has to be satisfied.

```
val of_term : Term.t -> t list -> Cdipoly.poly -> Cdipoly.poly
```

`of_term t inter d` returns the interpretation of the term `t` using the interpretation functions `inter`. `d` denotes the current value of the parameter delta in context dependent interpretations.

3.2 Module `Cdiparametric` : Creation of abstract polynomial and context dependent interpretations

Author(s): Sarah Winkler (for standard polynomial interpretations), Andreas Schnabl (for context dependent interpretations, and complexity relevant polynomial interpretations)

Creation of abstract polynomial and context dependent interpretations

Cdiparametric is responsible for creating abstract polynomial and context dependent interpretations. They can be made concrete by instantiating the existentially quantified variables.

```
type pclass =
  | LIN
  | QSIMP
  | SIMP
  | SM
```

```
  | QUAD
  | SLI
  | LINC
  | QSIMPC
  | SIMPC
  | SMC
  | QUADC
  | QUALIN
  | QUAQSIMP
  | QUASIMP
  | QUASM
  | QUAQUAD
  | DRES
  | DLIN
  | DLINC
val create :
  FunctionSymbol.t Setx.t ->
  pclass -> Trs.t -> Cdinterpretation.t list
```

create fs class trs returns an abstract interpretation of the kind denoted by pclass for each function symbol given in the set fs with respect to the TRS trs.

3.3 Module Cdipoly : Polynomials for polynomial and context dependent interpretations

Author(s): Sarah Winkler (for diophantine constraint solver), Andreas Schnabl (for context dependent interpretations)

Cdipoly provides all necessary algebraic foundations for polynomial and context dependent interpretations.

```
type monovariable =
  | FreshVar of int
  | RestrictedVar of int
type monomonomial =
  | Mmono of int * (monovariable * int) list
type monocoeff = monomonomial list
type variable =
  | Var of string
  | MetaVar of string
type varexp =
  | Varexp of variable * int
  | Deltaquotient of variable * monocoeff * monocoeff
type monomial =
  | Mono of monocoeff * varexp list
type poly = monomial list
```

```
type diophantinelistconstraint =
  | Greaterequal of monocoeff
  | Lessequal of monocoeff
  | Equal of monocoeff
type diophantineconstraint =
  | Greaterequal' of polynomial
  | Lessequal' of polynomial
  | Equal' of polynomial
type polynomial =
  | AbstractVar of var
  | Constant of int
  | Plus of polynomial * polynomial
  | Times of polynomial * polynomial
  | Power of var * int
type var =
  | ParseMetaVar of string
  | ParseVar of string
exception This_should_not_happen of string
exception Duplicate_var of variable
exception Duplicate_monovar of monovariable
val add_coeffs : monocoeff -> monocoeff -> monocoeff
```

`add_coeffs c1 c2` calculates the sum of the two polynomials `c1` and `c2` (which only contain existentially quantified variables). If possible, monomonomials with several equal power products are summed up into a single monomonomial.

```
val coefficient_polynomials : poly -> monocoeff list
```

`coefficient_polynomials p` splits the polynomial `p` into the coefficients of the universally quantified power products. If possible, monomials with several equal power products are summed up into a single monomial.

```
val delta : monomial
```

`delta` returns a monomial consisting only of the variable delta, which is used as special parameter in context dependent interpretations

```
val elim_deltaquotients : poly -> poly
```

`elim_deltaquotients p` eliminates Deltaquotients in `p` by multiplying `p` with the denominator of its Deltaquotients until no more Deltaquotients remain.

```
val elim_simple_delta : poly -> poly
```

`elim_deltaquotients p` eliminates Deltaquotients in `p` by replacing trivial Deltaquotients (of the form (delta/(1+0delta))) by delta.

`val join_vars_eq : diophantinelistconstraint -> diophantinelistconstraint`

`join_vars_eq eq` sums up monomials in `eq` with the same power products into a single monomial.

`val minus : poly -> poly -> poly`

`minus p1 p2` calculates the sum of `p1` and the negation of `p2`. If possible, monomials with several equal power products are summed up into a single monomial.

`val minus_coeffs : monocoeff -> monocoeff -> monocoeff`

`minus_coeffs c1 c2` calculates the sum of `c1` and the negation of `c2` (which only contain existentially quantified variables). If possible, monomonomials with several equal power products are summed up into a single monomonomial.

`val monoone : monomonomial`

`monoone` returns the constant one monomonomial

`val monozero : monomonomial`

`monozero` returns the constant zero monomonomial

`val multiply : poly -> poly -> poly`

`multiply p1 p2` calculates the product of the two polynomials `p1` and `p2`. If possible, monomials with several equal power products are summed up into a single monomial.

`val multiply_coeff : monocoeff -> monocoeff -> monocoeff`

`multiply_coeff c1 c2` calculates the product of the two polynomials `c1` and `c2` (which only contain existentially quantified variables). If possible, monomonomials with several equal power products are summed up into a single monomonomial.

`val one : monomial`

`one` returns the constant one monomial

`val poly_mono_multiply : poly -> monomial -> poly`

`multiply p m` calculates the product of `p` and `m`. If possible, monomials with several equal power products are summed up into a single monomial.

`val power : poly -> int -> poly`

`power p n` calculates the product of multiplying `n` copies of `p` with each other (i.e., the `n`th power of `p`). If possible, monomials with several equal power products are summed up into a single monomial.

`val quotientdelta : monomial`

`quotientdelta` returns the monomial representing the expression (delta/(1+0delta), where delta is the special parameter used in context dependent interpretations

```
val zero : monomial
```

`zero` returns the constant zero monomial

3.4 Module `Cdisolve` : Generation of SAT Encoding

Author(s): Andreas Schnabl

Generation of SAT Encoding
Cdisolve in responsible for encapsulating Diophantine constraints into satisfiabiliy constraints of propositional logic. It makes heavy use of the plogic library of TTT2.

```
exception No_solution
```

```
val last_built_bound : int Pervasives.ref
```

`last_built_bound` returns the variable bound whose SAT encoding was finished last by Cdisolve.

```
val last_intermediate_bound : int Pervasives.ref
```

`last_intermediate_bound` returns an internally tracked variable bound

```
val last_started_bound : int Pervasives.ref
```

`last_started_bound` returns the variable bound which was considered last by Cdisolve.

```
val solve :
  Cdipoly.diophantinelistconstraint list ->
  Cdipoly.diophantinelistconstraint list list ->
  int -> bool -> (Cdipoly.monovariable * int) list
```

`solve eqs oreqs n i` returns an assignment for the variables occurring in `eqs` and `or_eqs` such that the constraints in `eqs` are satisfied, and at least one of the lists of constraints in `or_ineqs` is satisfied. The upper bound for the instantion of the variables is given by `n`. If the flag `i` is set, then the bound is not fuly used from the beginning, but approached incrementally in several runs of the SAT solver. If the constraints are impossible to fulfill for the given bound, the exception `No_solution` is thrown.

```
val variable_count : int Pervasives.ref
```

`variable_count` returns the number of (not necessarily distinct) variable occurrences in the last formula given to the SAT solver.

3.5 Module Mcdiparametric : Creation of abstract matrix interpretations

Author(s): Sarah Winkler (for standard polynomial interpretations), Andreas Schnabl (for context dependent interpretations, complexity relevant polynomial interpretations, and matrix interpretations)

Creation of abstract matrix interpretations

Mcdiparametric is responsible for creating abstract matrix interpretations. They can be made concrete by instantiating the existentially quantified variables.

```
type metavector =
  | Metavec of Mcdipoly.variable list
type freshvector =
  | Freshvec of Mcdipoly.monovariable list
type freshmatrix =
  | Freshmat of freshvector list
type extmonovariable =
  | ZeroVar
  | MVar of Mcdipoly.monovariable
type extfreshvector =
  | EFreshvec of extmonovariable list
type extfreshmatrix =
  | EFreshmat of extfreshvector list
type pclass =
  | LIN
  | TRI
  | LINC
  | CDIEQ
  | DLIN
  | DRES
exception Unsupported_poly_class
val create :
  FunctionSymbol.t Setx.t ->
  pclass ->
  Trs.t ->
  int ->
  bool ->
  FunctionSymbol.t Setx.t ->
  (string *
   ((Mcdipoly.vector * Mcdipoly.matrix list * Mcdipoly.vector) *
    (metavector * Mcdipoly.poly) list))
  list * extfreshmatrix list list *
  (Mcdipoly.monomonomial * Mcdipoly.monomonomial) list list *
  (extfreshmatrix * extfreshmatrix) list list
```

`create fs class trs d is_quasi safe_fs` returns an abstract matrix interpretation of dimension `d` and the kind denoted by `pclass` for each function symbol given in the set `fs` with respect to the TRS `trs`. `safe_fs` denotes a list of function symbols which have to be itnerpreted by strictly monotone functions. Moreover, `create` returns a list of matrices whose upper left entry has to be positive, a list of pairs of monomonomials whose sums have to be positive, and the list of all fresh matrices created for the interpretation.

3.6 Module `Mcdipoly` : Creation of standard and context dependent matrix interpretations

Author(s): Sarah Winkler (for diophantine constraints), Andreas Schnabl (for context dependent and matrix interpretations)

Mcdipoly provides all necessary algebraic foundations for matrix interpretations. This module is based on the module Cdipoly.

```
type monovariable =
  | FreshVar of int
  | RestrictedVar of int
type monomonomial =
  | Mmono of int * (monovariable * int) list
type monocoeff = monomonomial list
type variable =
  | Var of string
  | MetaVar of string
type varexp =
  | Varexp of variable * int
  | Deltaquotient of variable * monocoeff * monocoeff
type monomial =
  | Mono of monocoeff * varexp list
type poly = monomial list
type vector = poly list
type mrow =
  | Row of vector
type mcol =
  | Col of vector
type matrix = mrow list
type colmatrix = mcol list
type diophantinelistconstraint =
  | Greaterequal of monocoeff
  | Lessequal of monocoeff
  | Equal of monocoeff
type diophantineconstraint =
```

```
  | Greaterequal' of polynomial
  | Lessequal' of polynomial
  | Equal' of polynomial
type polynomial =
  | AbstractVar of var
  | Constant of int
  | Plus of polynomial * polynomial
  | Times of polynomial * polynomial
  | Power of var * int
type var =
  | ParseMetaVar of string
  | ParseVar of string
exception This_should_not_happen of string
exception Duplicate_var of variable
exception Duplicate_monovar of monovariable
val add_coeffs : monocoeff -> monocoeff -> monocoeff
```

`add_coeffs c1 c2` calculates the sum of the two polynomials `c1` and `c2` (which only contain existentially quantified variables). If possible, monomonomials with several equal power products are summed up into a single monomonomial.

```
val coefficient_polynomials : poly -> monocoeff list
```

`coefficient_polynomials p` splits the polynomial `p` into the coefficients of the universally quantified power products. If possible, monomials with several equal power products are summed up into a single monomial.

```
val delta : monomial
```

`delta` returns a monomial consisting only of the variable delta, which is used as special parameter in context dependent interpretations

```
val deltadiffrow : int -> vector
```

`deltadiffrow d` returns a vector of dimension `d` containing delta in the first entry, and the zero polynomial in each other entry.

```
val elim_deltaquotients : poly -> poly
```

`elim_deltaquotients p` eliminates Deltaquotients in `p` by multiplying `p` with the denominator of its Deltaquotients until no more Deltaquotients remain.

```
val elim_simple_delta : poly -> poly
```

`elim_deltaquotients p` eliminates Deltaquotients in `p` by replacing trivial Deltaquotients (of the form (delta/(1+0delta))) by delta.

```
val matrix_add : matrix -> matrix -> matrix
```

`matrix_add m n` adds the matrices `m` and `n` in a pointwise fashion. In each addition, monomials with several equal power products are summed up into a single monomial, if possible.

```
val join_vars_eq : diophantinelistconstraint -> diophantinelistconstraint
```

`join_vars_eq eq` sums up monomials in `eq` with the same power products into a single monomial.

```
val matrix_vector_mult : matrix -> vector -> vector
```

`matrix_vector_mult m x` computes the product `mx` of the matrix `m` and the vector `x`. In each addition and multiplication, monomials with several equal power products are summed up into a single monomial, if possible.

```
val minus : poly -> poly -> poly
```

`minus p1 p2` calculates the sum of `p1` and the negation of `p2`. If possible, monomials with several equal power products are summed up into a single monomial.

```
val minus_coeffs : monocoeff -> monocoeff -> monocoeff
```

`minus_coeffs c1 c2` calculates the sum of `c1` and the negation of `c2` (which only contain existentially quantified variables). If possible, monomonomials with several equal power products are summed up into a single monomonomial.

```
val monoone : monomonomial
```

`monoone` returns the constant one monomonomial

```
val monozero : monomonomial
```

`monozero` returns the constant zero monomonomial

```
val multiply : poly -> poly -> poly
```

`multiply p1 p2` calculates the product of the two polynomials `p1` and `p2`. If possible, monomials with several equal power products are summed up into a single monomial.

```
val multiply_coeff : monocoeff -> monocoeff -> monocoeff
```

`multiply_coeff c1 c2` calculates the product of the two polynomials `c1` and `c2` (which only contain existentially quantified variables). If possible, monomonomials with several equal power products are summed up into a single monomonomial.

```
val one : monomial
```

`one` returns the constant one monomial

```
val onediffrow : int -> vector
```

`onediffrow d` returns a vector of dimension `d` containing the one polynomial in the first entry, and the zero polynomial in each other entry.

`val poly_mono_multiply : poly -> monomial -> poly`

`multiply p m` calculates the product of `p` and `m`. If possible, monomials with several equal power products are summed up into a single monomial.

`val power : poly -> int -> poly`

`power p n` calculates the product of multiplying `n` copies of `p` with each other (i.e., the `n`th power of `p`). If possible, monomials with several equal power products are summed up into a single monomial.

`val quotientdelta : monomial`

`quotientdelta` returns the monomial representing the expression (delta/(1+0delta), where delta is the special parameter used in context dependent interpretations

`val scalar_matrix_mult : poly -> matrix -> matrix`

`scalar_matrix_mult p m` multiplies `p` and the matrix `m` in a pointwise fashion. In each multiplication, monomials with several equal power products are summed up into a single monomial, if possible.

`val scalar_vector_mult : poly -> vector -> vector`

`scalar_vector_mult p xs` multiplies `p` and the vector `xs` in a pointwise fashion. In each multiplication, monomials with several equal power products are summed up into a single monomial, if possible.

`val to_columns : matrix -> colmatrix`

`to_columns m` changes the representation of `m` (which is normally done by giving a list of its row vectors) into a list of its column vectors.

`val vector_add : vector -> vector -> vector`

`vector_add xs ys` returns the sum of the two vector `xs` and `ys` by doing component-wise addition. In each addition, monomials with several equal power products are summed up into a single monomial, if possible.

`val zero : monomial`

`zero` returns the constant zero monomial

`val zero_row : int -> vector`

`zero_row d` returns a vector of dimension `d` containing the zero polynomial in each entry.

3.7 Module `Mcdiprove` : Creation of matrix interpretations

Author(s): Sarah Winkler (for polynomial interpretations), Andreas Schnabl (for context dependent and matrix interpretations)

Creation of matrix interpretations

Mcdiprove is the interface class for polynomial and context dependent interpretations.

```
type proof = Mcdinterpretation.t list * (Mcdipoly.monovariable * int) list * string
  int * Trs.t * Trs.t
```

```
type complexitykind =
  | DC
  | RC
  | CP
```

```
type complexityclass =
  | Specpoly of int
  | Polynomial
  | Exponential
  | Doubleexp
  | Primrec
  | Unknown
```

```
exception Unsupported_poly_class
```

```
val complexity_of_class :
  Mcdiparametric.pclass ->
  int -> complexitykind -> complexityclass
```

`complexity_of_class d pclass c` returns an upper bound in the complexity category `c` induced by a proof of kind `pclass` wth dimension `d`.

```
val fprintf_proof : Format.formatter -> proof -> unit
```

`fprintf_proof ppf proof` pretty-prints `proof` to the formatter `ppf`

```
val normalize_class : string -> Mcdiparametric.pclass
```

`normalize_class pclass` turns a string representation of a proof category into its internal representation of type pclass.

```
val terminating_class :
  Trs.t ->
  Trs.t -> int -> string -> bool -> int -> bool -> proof option
```

`terminating_class strict weak n class is_incremental d is_sdp` tries to compute a compatible matrix interpretation of the kind specified by `pclass` for the TRSs `strict` and `weak`. The rules contained in `strict` are strictly oriented, and the ruels contained in `weak` are weakly oriented if a suitable interpretation is found. `n` denotes an upper bound for all occurring coefficient. `d` fixes the dimension of the used matrices and vectors. `is_incremental` decides whether `n` should be fully used in

the first run of the SAT solver (if is_incremental is false), or should be approached incrementally (otherwise).If is_sdp is set, then all non-compound symbols only need to have weakly monotone interpretation functions (instead of strcitly monotone), i.e., a safe reduction pair is generated instead of a direct reduction ordering.

3.8 Module Mcdisolve : Generation of SAT Encoding

Author(s): Andreas Schnabl

Generation of SAT Encoding

Mcdisolve in responsible for encapsulating Diophantine constraints into satisfiabiliy constraints of propositional logic. It makes heavy use of the plogic library of TTT2.

```
exception No_solution
```

```
val last_built_bound : int Pervasives.ref
```

last_built_bound returns the variable bound whose SAT encoding was finished last by Cdisolve.

```
val last_intermediate_bound : int Pervasives.ref
```

last_intermediate_bound returns an internally tracked variable bound

```
val last_started_bound : int Pervasives.ref
```

last_started_bound returns the variable bound which was considered last by Cdisolve.

```
val solve :
  Mcdipoly.diophantinelistconstraint list ->
  int -> bool -> (Mcdipoly.monovariable * int) list
```

solve eqs n i returns an assignment for the variables occurring in eqs and such that all constraints in eqs are satisfied. The upper bound for the instantion of the variables is given by n. If the flag i is set, then the bound is not fuly used from the beginning, but approached incrementally in several runs of the SAT solver. If the constraints are impossible to fulfill for the given bound, the exception No_solution is thrown.

3.9 Module Mcdinterpretation : Creation of constraints for abstract matrix interpretations

Author(s): Sarah Winkler (for polynomial interpretations), Andreas Schnabl (for context dependent and matrix interpretations)

Creation of constraints for abstract matrix interpretations

Mcdinterpretation is responsible for creating constraints on the possible instantions of matrix interpretations.

```
type t = string *
  ((Mcdipoly.vector * Mcdipoly.matrix list * Mcdipoly.vector) *
   (Mcdiparametric.metavector * Mcdipoly.poly) list)
val coefficient_inequalities_of_trs :
  Trs.t ->
  t list ->
  bool ->
  bool ->
  int ->
  Mcdiparametric.extfreshmatrix list ->
  (Mcdipoly.monomonomial * Mcdipoly.monomonomial) list ->
  (Mcdiparametric.extfreshmatrix * Mcdiparametric.extfreshmatrix) list ->
  Mcdipoly.diophantinelistconstraint list
```

`coefficient_inequalities_of_trs trs inter is_ftaui is_quasi dim monotones monotone_twos fms` creates a list of inequalities from a TRS `trs` and a polynomial or context dependent interpretation `inter` which ensure the according monotonicity and compatibility properties. The boolean argument `is_ftaui` specifies whether the given interpretation is polynomial or context dependent. Setting the flag `is_quasi` specifies that the rules of `trs` should be weakly oriented. The set `fs` gives the list of function symbols whose interpretation functions have to be strictly monotone - for all function symbols not in that set, only weak monotonicity is enforced. The list `monotones` denotes a list of matrices whose upper left entries have to be positive. The list `monotone_twos` gives a list of pairs of monomonomials whose sums have to be positive. `fms` gives the collection of all matrices which were used in the interpretation `inter`.

3.10 Module `Cdiprove` : Creation of polynomial and context dependent interpretations

Author(s): Sarah Winkler (for polynomial interpretations), Andreas Schnabl (for context dependent interpretations)

Creation of polynomial and context dependent interpretations

Cdiprove is the interface class for polynomial and context dependent interpretations.

```
type proof = Cdinterpretation.t list * (Cdipoly.monovariable * int) list * string *
  bool * Trs.t * Trs.t * Trs.t
type complexitykind =
  | DC
  | RC
  | CP
type complexityclass =
  | Linear
```

```
  | Quadratic
  | Polynomial
  | Exponential
  | Doubleexp
  | Primrec
  | Unknown
exception Unsupported_poly_class
val complexity_of_class :
  Cdiparametric.pclass -> complexitykind -> complexityclass
```

`complexity_of_class pclass c` returns an upper bound in the complexity category `c` induced by a proof of kind `pclass`.

```
val fprintf_proof : Format.formatter -> proof -> unit
```

`fprintf_proof ppf proof` pretty-prints `proof` to the formatter `ppf`

```
val normalize_class : string -> Cdiparametric.pclass
```

`normalize_class pclass` turns a string representation of a proof category into its internal representation of type pclass.

```
val safe_redpair_terminating_class :
  Trs.t -> Trs.t -> int -> string -> bool -> bool -> proof option
```

`safe_redpair_terminating_class strict weak n class is_incremental is_relative` does the same as `terminating_class strict weak n class is_incremental is_relative`, but creates a reduction pair instead of a single reduction order. This means that all interpretation functions only have to be weakly monotone (instead of strictly monotone), except for the compound symbols of the weak dependency pair method.

```
val terminating_class :
  Trs.t -> Trs.t -> int -> string -> bool -> bool -> proof option
```

`terminating_class strict weak n class is_incremental is_relative` tries to compute a compatible polynomial or context dependent interpretation of the kind specified by `pclass` for the TRSs `strict` and `weak`. The rules contained in `strict` are strictly oriented, and the ruels contained in `weak` are weakly oriented if a suitable interpretation is found. `n` denotes an upper bound for all occurring coefficient. `is_incremental` decides whether `n` should be fully used in the first run of the SAT solver (if `is_incremental` is false), or should be approached incrementally (otherwise).If `is_relative` is set, then the rules in `strict` are only weakly oriented - only at least one rule in `strict` has to be strictly oriented.

4 POP*

4.1 Module Pop : POP* with optionally semantic labeling and POP* as reduction pair

```
exception Invalid_arguments of string
```

```
type proof
```

An aggregation of the proof objects of the modules popSL and popWidp.

```
val solve : bool -> bool -> bool -> int -> Trs.t -> Trs.t -> proof option
```

solve is_dp_problem is_strict is_incremental bound strict_TRS weak_TRS applies either POP* as reduction pair to strict_TRS and weak_TRS (when is_dp_problem = true) or orients strict_TRS strongly using POP* (when is_dp_problem = false). The arguments is_strict, is_incremental and bound are currently only supported when is_dp_problem = false. Otherwise the exception Invalid_arguments is raised. The argument is_strict tells wheter a strict precedence should be employed. When bound <= 1 then semantic labeling is not applied, else bound gives the cardinality of the carrier for finite semantic labeling with model. When is_incremental = true, then bound is incrementally increased upon failure.

```
val fprintf_proof : Format.formatter -> proof -> unit
```

Pretty printing function of proof objects

4.2 Module PopSL : POP* with and without finite semantic labeling.

```
type proof
```

```
val solve : int -> bool -> bool -> Trs.t -> proof option
```

For documentation, c.f. Pop.solve.

```
val fprintf_proof : Format.formatter -> proof -> unit
```

For documentation, c.f. Pop.solve.

4.3 Module PopWidp : POP* as reduction pair

```
type proof
```

```
val solve : Trs.t -> Trs.t -> proof option
```

For documentation, c.f. Pop.solve.

```
val fprintf_proof : Format.formatter -> proof -> unit
```

For documentation, c.f. Pop.solve.

5 Propositional Logic

5.1 Module PLogx : Extensions to PLog

PLog library

```
val big_and : Formula.t list -> Formula.t
```

These functions below directly correspond to the counterparts in module PLog.

```
val big_or : Formula.t list -> Formula.t
val bot : Formula.t
val fresh : unit -> Formula.var
val is_atom : Formula.t -> bool
val is_var : Formula.t -> bool
val is_sat : Formula.t -> Assignment.t option
val no_cycles : ((int * int) * Formula.t) Setx.t -> Formula.t
val s_and : Formula.t -> Formula.t -> Formula.t
val s_or : Formula.t -> Formula.t -> Formula.t
val s_not : Formula.t -> Formula.t
val s_iff : Formula.t -> Formula.t -> Formula.t
val s_implies : Formula.t -> Formula.t -> Formula.t
val xor : Formula.t -> Formula.t -> Formula.t
val test : unit -> unit
val top : Formula.t
val val_of : Formula.t -> Assignment.t -> bool
```

Extensions to the PLog library

Below convenience functions for constructing propositional formulas are given. All these functions try to reduce the formula size by applying rules of propositional logic.

```
val ite : Formula.t -> Formula.t -> Formula.t -> Formula.t
```

`ite guard then else` constructs a formula corresponding to an if-expression. When `guard` reduces to `top` or `bot` then `then` or `else` is returned.

```
val cases : (Formula.t * Formula.t) list -> Formula.t
```

`cases cs` constructs a formula from the assoc-list `cs` consisting of tuples `(precondition, conclusion)`.

```
val zero : Formula.t list -> Formula.t
```

`zero fs` expresses that none of the formulas `fs` are true.

```
val one : Formula.t list -> Formula.t
```

`one fs` expresses that exactly one of the formulas `fs` is true.

```
val exists :
  ?filter:('a -> bool) -> 'a list -> ('a -> Formula.t) -> Formula.t
```

`exists ~filter:filter ls fn` produces a formula corresponding to the disjunction of `fn` applied to every element in the set $\{l \mid \neg(\mathsf{filter}\ l),\ l \in ls\}$.

```
val exists_set :
  ?filter:('a -> bool) -> 'a Setx.t -> ('a -> Formula.t) -> Formula.t
```

Same as `exists ~filter:filter ls fn` but takes a set of elements

```
val exists2 : 'a list -> 'b list -> ('a -> 'b -> Formula.t) -> Formula.t
```

`exists2 xs ys fn` produces a formula corresponding to the disjunction of `fn` applied pairwise on elements from `xs` and `ys`.

Raises `Invalid_argument` when `xs` any `ys` are of different length.

```
val forall :
  ?filter:('a -> bool) -> 'a list -> ('a -> Formula.t) -> Formula.t
```

`forall ~filter:filter ls fn` produces a formula corresponding to the conjuction of `fn` applied to every element in the set $\{l \mid \neg(\mathsf{filter}\ l),\ l \in ls\}$.

```
val forall_set :
  ?filter:('a -> bool) -> 'a Setx.t -> ('a -> Formula.t) -> Formula.t
```

Same as `forall ~filter:filter ls fn` but takes a set of elements

```
val forall2 : 'a list -> 'b list -> ('a -> 'b -> Formula.t) -> Formula.t
```

`forall2 xs ys fn` produces a formula corresponding to the conjunction of `fn` applied pairwise on elements from `xs` and `ys`.

Raises `Invalid_argument` when `xs` any `ys` are of different length.

```
val fm : bool -> Formula.t
```

Lifts a boolean value to a formula.

```
val fprintf_formula : Format.formatter -> Formula.t -> unit
val formula_size : Formula.t -> int
module Infix :
  sig
```

```
    val (&&) : Formula.t -> Formula.t -> Formula.t
```

Infix conjuction

```
    val (||) : Formula.t -> Formula.t -> Formula.t
```

Infix disjunction

```
val (-->) : Formula.t -> Formula.t -> Formula.t
```

Infix implication

```
val (<->) : Formula.t -> Formula.t -> Formula.t
```

Infix iff

```
val neg : Formula.t -> Formula.t
```

Negation

```
end
```

5.2 Module LazyPLog : Lazy processing of propositonal formulas

This module provides the functionality of module Plog but lifted to lazy expressions. For a documentation, please revice module documentation for PLog

```
type t = Formula.t Lazy.t
val top : Formula.t lazy_t
val bot : Formula.t lazy_t
val s_or : t -> t -> t
val s_or' : Formula.t -> t -> Formula.t
val s_and : t -> t -> t
val s_and' : Formula.t -> t -> Formula.t
val s_implies : t -> t -> t
val s_not : t -> t
val big_or : t list -> t
val big_or' : t list -> Formula.t
val big_and : t list -> t
val big_and' : t list -> Formula.t
val forall : 'a list -> ('a -> Formula.t) -> t
val forall' : 'a list -> ('a -> Formula.t) -> Formula.t
val forall_set : 'a Setx.t -> ('a -> Formula.t) -> t
val forall_set' : 'a Setx.t -> ('a -> Formula.t) -> Formula.t
val exists : 'a list -> ('a -> Formula.t) -> t
val exists_set : 'a Setx.t -> ('a -> Formula.t) -> t
val exists' : 'a list -> ('a -> Formula.t) -> Formula.t
val exists_set' : 'a Setx.t -> ('a -> Formula.t) -> Formula.t
val ite' : Formula.t -> t -> t -> Formula.t
```

ite' guard then else constructs a formula corresponding to an if-expression. When guard reduces to top or bot then then or else is returned.

```
val fm : bool -> t
```

Lift boolean values to lazy formulas

```
module Infix :
  sig
```

```
    val (&&.) : Formula.t -> LazyPLog.t -> Formula.t
```

infix conjunction.

```
    val (||.) : Formula.t -> LazyPLog.t -> Formula.t
```

infix disjunction.

```
    val (-->.) : Formula.t -> LazyPLog.t -> Formula.t
```

infix implication.

```
    val (&&*) : LazyPLog.t -> LazyPLog.t -> LazyPLog.t
```

infix conjunction.

```
    val (||*) : LazyPLog.t -> LazyPLog.t -> LazyPLog.t
```

infix disjunction.

```
    val (-->*) : LazyPLog.t -> LazyPLog.t -> LazyPLog.t
```

infix implication.

```
  end
```

5.3 Module `VarCache` : Caching of propositional variables

This module provides a general interface for associating values with fresh, unique propositional variables. This module uses imperative features. A variable cache relates values of type `KEY.t` to elements of type `VALUE.t`. Elements of type `VALUE.t` are encoded as elements of type `VALUE.enc`. The propositional encoding of `VALUE.t` is goverened by the particular implementation of `VALUE`. For instance, the module `NatEncoding` implements the encoding of naturals using the `BinNat` module. Below, various modules implementing `VALUE` for different types of `VALUE.t` are given. A concrete example of a variable cache is `MakeVarCache(struct type t = (int,int) end)(NatEncoding)`. This module associates tuples of integers with the encoding of natural numbers. For each tuple `(a,b)` unique variables are generated and cached.

```
module type TYPABLE =
  sig
```

```
    type t
```

```
  end
module type KEY =
  TYPABLE
module type VALUE =
  sig
```

```
    type t
```

Values of variable caches
The type of a concrete instance.

```
    type opt
```

Options for generating fresh encodings of values.

```
    type enc
```

The type of the encoding.

```
    val fresh : opt -> enc
```

Generate a fresh encoding under the given options.

```
    val from_assignment : enc -> Assignment.t -> t
```

Construct from an encoding and assignment a concrete instance of the encoded value.

```
  end
module type VARCACHE =
  sig
```

```
    type t
```

The type of a variable cache.

```
    module Key :
    VarCache.KEY
```

The module implementing the key.

```
    module Value :
    VarCache.VALUE
    val cache : Value.opt option -> Key.t -> Value.enc
```

`cache option key` returns the encoding of the value for the given key `key`. If no value was previously cached, a new encoding is constructed by means of `Value.fresh option` and stored.
Raises `Not_initialized` when global cache is not initialized

```
val fold : (Key.t -> Value.enc -> 'a -> 'a) -> 'a -> 'a
```

Standard fold over the value

```
val initialize : ?debug_name:string -> Value.opt -> int -> unit
```

This function initializes the variable cache. The second argument defines the default options, the third the initial size of the cache.

```
val size : unit -> int
```

Returns the current size of the cache.

```
val fprintf_status : Format.formatter -> unit -> unit
```

Pretty prints a small status message.

```
val from_assignment : Assignment.t -> Key.t -> Value.t -> Value.t
```

`from_assignment assign key default` constructs from the assignment `assign` the concrete value of the encoding associated with `key`. If no encoding is associated with `key` then `default` is returned.

```
val with_fresh_cache :
  ?size:int -> Value.opt -> ((Key.t -> Value.enc) -> 'a) -> 'a
```

`with_fresh_cache ~size:n option cache` provides a non-imperative interface to the `varCache` module. The final argument take the role of the `cache` function described above.

```
  end

module MakeVarCache :
  functor (K : KEY) -> functor (V : VALUE) -> VARCACHE  with module
Key = K and module Value = V
module BoolEncoding :
  sig
```

Encoding of boolean values

```
type t = bool
type enc = Formula.var
type opt =
  | NoOpt
```

No Options

```
val fresh : opt -> enc
val fprintf : Assignment.t option -> Format.formatter -> enc -> unit
val from_assignment : enc -> Assignment.t -> t
```

```
  end
module NatEncoding :
  sig
    type t = int
```

Encoding of naturals via the BinNat module

```
    type enc = BinNat.t
    type opt =
      | Max of int
```

The maximum value encoded

```
    val fresh : opt -> enc
    val from_assignment : enc -> Assignment.t -> int
    val fprintf : 'a -> Format.formatter -> Formula.t list -> unit
    val gt : enc -> enc -> Formula.t
    val eq : enc -> enc -> Formula.t
  end
```

5.4 Module ArgumentFilteringCache : Propositional encodings of argument filters

This module provides a variable cache for the encoding of argument filterings. Non-imperative interface.

```
type is_collapsing = VariadicFunctionSymbol.t -> Formula.t
type is_unfiltered = VariadicFunctionSymbol.t -> int -> Formula.t
type is_usable = VariadicFunctionSymbol.t -> Formula.t
type predicates = is_collapsing * is_unfiltered
type variadicSignature = VariadicFunctionSymbol.t Setx.t
type argumentFiltering = ArgumentFiltering.elt ArgumentFiltering.t
val with_af : Trs.t -> (predicates -> Formula.t) -> Formula.t
```

`with_af trs f` produces the conjunction of the propositional formula `f predicates` together with constraints asserting a valid encoding of argument filterings. The function `f` is supplied with additional predicates for the encoded filtering

```
val from_assignment :
  variadicSignature ->
  Assignment.t -> argumentFiltering
```

`from_assignment signature assign` constructs an argument filtering for the signature `signature` based on the evaluation of the employed variables given from the assignment `assign`.

5.5 Module `PrecedenceCache` : This module provides a variable cache for the encoding of quasi- and strict precedences.

Non-imperative interface.

```
type gt = FunctionSymbol.t -> FunctionSymbol.t -> Formula.t
type eq = FunctionSymbol.t -> FunctionSymbol.t -> Formula.t
type predicates = gt * eq
type signature = FunctionSymbol.t Setx.t
val with_precedence : signature ->
  (predicates -> Formula.t) -> Formula.t
```

`with_precedence signature f` produces the conjunction of the propositional formula `f predicates` together with constraints asserting a valid encoding of the precedence. It asserts that the resulting strict part `>` is indeed a proper order, likewise it is asserted that the resulting equivalence relation `~` is indeed an equivalence relation. The function `f` is supplied with additional predicates for the encoded precedences.

```
val to_precedence : Assignment.t -> Precedence.t
```

`to_assignment assign` constructs a precedence based on the evaluation of the employed variables given from the assignment `assign`.

5.6 Module `SafeMappingCache` : Propositional encoding of safe mappings

This module provides a variable cache for the encoding of safe mappings. Non-imperative interface.

```
type is_normal = VariadicFunctionSymbol.t -> int -> Formula.t
type is_safe = VariadicFunctionSymbol.t -> int -> Formula.t
type predicates = is_normal * is_safe
type signature = VariadicFunctionSymbol.t Setx.t
val with_safe_mapping :
  signature ->
  signature ->
  (predicates -> Formula.t) -> Formula.t
```

`with_safe_mapping signature constructors f` produces the conjunction of the propositional formula `f predicates` together with constraints asserting a valid encoding of a safe mapping filterings. In particular, it is asserted that all positions of symbols in `constructors` are safe. The function `f` is supplied with additional predicates for the encoded safe mapping.

```
val from_assignment : Assignment.t -> SafeMapping.t
```

`from_assignment assign` constructs a safe mapping based on the evaluation of the employed variables given from the assignment `assign`.

5.7 Module FiniteFnEncoding : Propositional encoding of functions with finite carrier

```
type arity = int
```

```
type domain = int Setx.t
```

```
type opt = (arity * domain) option
```

This module uses internally the module `VarCache`. The type `opt` corresponds to the initialization option of `VarCache`. For an encoding of functions with finite carrier, this is a tuple of the arity and domain of the encoded function f: dom$^n \to$ dom .

```
type enc
```

Abstract type of an encoding.

```
type t
```

Abstract type of a function with finite carrier.

```
val fresh : opt -> enc
```

`fresh option` produces a fresh encoding.

Raises `Invalid_argument` when `opt=None`.

```
val fprintf_enc : Format.formatter -> enc -> unit
```

Pretty printing of encodings.

```
val fprintf : Format.formatter -> t -> unit
```

Pretty printing of functions over finite domain.

```
val from_assignment : enc -> Assignment.t -> t
```

`from_assignment enc assign` constructs from an encoding and satisfying assignment a concrete instance of the encoded function.

```
val define : enc -> BinNat.t list -> BinNat.t -> Formula.t
```

`define enc arguments r` produces a propositional formula that asserts for arguments $= [a_1, ..., a_n]$ that $f(a_1, ..., a_n) = r$ for a satisfying assignment α .

```
val fold_enc :
  (Assign.NatAssign.t -> BinNat.t -> 'a -> 'a) ->
  enc -> 'a -> 'a
```

Standard `fold` over abstract encodings.

```
val fold :
  (Assign.NatAssign.t -> Assign.Nat.t -> 'a -> 'a) ->
  t -> 'a -> 'a
```

Standard `fold` over concrete instances.

```
val weak_monotonicity :
  (Assign.Nat.t -> Assign.Nat.t -> Formula.t) ->
  enc -> Formula.t
```

`weak_monotinicity ord enc` produces a propositional formula that asserts weak monotonicity of the encoded function `enc` with respect to the ordering relation given by `ord`.

```
val value : t -> Assign.Nat.t list -> Assign.Nat.t
```

`value f args` returns the result of the concrete instance `f` when given arguments `args`.

5.8 Module `FiniteModel` : Function symbols with abstract labels

```
module LabelledVariadicFunctionSymbol :
  sig
```

Function symbols with abstract labels

```
type t
val root_symbol : (Term.t -> VarCache.NatEncoding.enc) ->
  Term.t -> t
```

given a labeling function `l` and term `t`, `root_symbol l t` returns the abstract labelled root symbol of `t`.

```
val unlabeled_root_symbol : Term.t -> t
```

Same as `root_symbol`, but without label.

```
val label : t -> VarCache.NatEncoding.enc
```

returns the label of a symbol.

```
val variadic_symbol : t -> VariadicFunctionSymbol.t
```

returns the given symbol as variadic function symbol.

```
val symbol : t -> FunctionSymbol.t
```

returns the given symbol as non-variadic function symbol.

```
val lift : (VariadicFunctionSymbol.t -> 'a) ->
  t -> 'a
```

Lift a function of type VariadicFunctionSymbol.t -> 'a to type t -> 'a

```
val arity : t -> int
```

returns the arity of the given symbol

```
val argument_positions : t -> int list
```

returns a list of all argument positons of the given symbol.

```
val equal : t ->
  t -> Formula.t
```

Propositional encoding of equality between two abstract labeled symbols.

```
end
```

```
type interpretation = Term.t -> VarCache.NatEncoding.enc
```

An (abstract) interpretation is a mapping from terms to the encoding of naturals .

```
type labeling = Term.t -> VarCache.NatEncoding.enc
```

An (abstract) labeling is a mapping from terms to the encoding of naturals.

```
val initialize : int -> unit
```

This module relies on imperative features. The function initialize, when given n, initializes the domain 0,...,n for use in the encodings of the labeling and the algebra.

```
val forall_labelled_rules :
  Trs.t ->
  (interpretation ->
   labeling -> Assign.NatAssign.t -> Rule.t -> Formula.t) ->
  Formula.t
```

forall_labelled_rules trs f returns a formula assessing f interpretation labeling assignment rule holds for all and assignments and rules in the given trs. Here interpretation and labeling correspond to the interpretation- and labeling functions under the current assignment assignment.

```
val model_condition : Trs.t -> Formula.t
```

model_condition trs produces a propositional formula asserting that the internally employed encodings of labeling functions $\mathcal{A}$ satisfy the model condition, i.e $[\alpha]_{\mathcal{A}}(l) = [\alpha]_{\mathcal{A}}(r)$ for all rules $l \to r \in \mathcal{R}$ under the given assignment α .

```
val labelled_signature : ('a -> int -> 'b) -> 'a Setx.t -> 'b Setx.t
```

`labelled_signature f signature` computes from the signature `signature` the labeled signature with the help of the labeling function `f`. The function `f` takes as first argument an unlabeled symbol and as second argument the label, denoted by an integer.

```
val with_label :
  LabelledVariadicFunctionSymbol.t ->
  (VariadicFunctionSymbol.t -> Formula.t) -> Formula.t
```

`with_label lsym f` constructs a formula defined by case analysis on the abstract label of `lsym`, where the label ranges over all elements from the domain. Depending on this label, it asserts that the formula obtained from `f sym` holds. The function symbol `sym` is obtained by instantiating `lsym`.

```
val algebra_from_assignment : Assignment.t -> Algebra.t
```

Given a satisfying assignment, it constructs an algebra from the internal mutable variables.

```
val labelling_from_assignment : Assignment.t -> Algebra.t
```

Given a satisfying assignment, it constructs labeling from the internal mutable variables.

```
val labelled_trs_from_assignment : Assignment.t -> Trs.t -> Trs.t
```

Given a satisfying assignment and a TRS, it constructs the labeled TRS from the internal mutable variables.

5.9 Module `ProperOrderedSet` : Encoding of proper ordered sets

```
module Make :
  functor (K : VarCache.KEY) ->   sig
```

Encoding of proper ordered sets

This module relies on imperative features, that is the module `VarCache`. It provides an implementation for the encoding of proper odered sets over the type `K.t`.

```
val initialize : K.t list -> unit
```

Initialize the variable cache with domain as defined in the first argument

```
val carrier : unit -> K.t Setx.t
```

Returns the elements from the poset

```
val gt : K.t -> K.t -> Formula.t
```

`gt a b` constructs a formula asserting $a > b$ with respect the the encoded order ¿ . Notice that ¿ is not per default a proper order. This needs to be enforced by `assert_transitive` and `assert_irreflexive` given below.

```
val assert_transitive : unit -> Formula.t
```

Asserts that all variables employed in the encoding form a transitive relation

```
val assert_irreflexive : unit -> Formula.t
```

Asserts that all variables employed in the encoding form a irreflexive relation

```
end
```

6 Dependency Pairs

6.1 Module `Wdp` : Weak Dependency Pairs

Author(s): Andreas Schnabl

Weak Dependency Pairs
This is an implementation of the WDP method by Hirokawa and Moser.

```
type dptype =
  | WDP
  | WIDP
  | DP
type proof = Trs.t * Trs.t * dptype
type redpairproof = Trs.t * Trs.t * dptype * Cdiprove.proof
val fprintf : Format.formatter -> proof -> unit
```

`fprintf ppf proof` pretty-prints `proof`, which is expected to be a WDP/WIDP/DP transformation, to `ppf`.

```
val fprintf_redpair : Format.formatter -> redpairproof -> unit
```

`fprintf_redpair ppf proof` pretty-prints `proof`, which is expected to be a WDP/WIDP/DP transformation combined with an SLI for the usable rules, to `ppf`.

```
val usable_rules : Trs.t -> Trs.t -> Trs.t
```

`usable_rules strict weak` returns the usable rules of `weak` with respect to `strict`.

```
val wdp_inn : bool -> bool -> Trs.t -> proof option
```

`wdp_inn forcedp forcewdp trs` transforms `trs` into the union of its weak innermost dependency pairs or dependency pairs, and the respective usable rules. If `trs` has a rule whose right hand side contains a variable not contained in its left hand side, then None is returned. If both `forcedp` and `forcewdp` are false, then dependency pairs are chosen if all compound symbos are nullary, and weak innermost dependency pairs are chosen otherwise. If the flag `forcedp` is true, then None is returned whenever weak innermost dependency pairs would be chosen. If `forcewdp` is true, then weak innermost dependency pairs are always chosen.

```
val wdp_nosli_inn : bool -> bool -> Trs.t -> proof option
```

`wdp_nosli_inn forcedp forcewdp trs` transforms `trs` into a DP problem, where the weak innermost dependency pairs or dependency pairs are the strict part, and the usable rules are the weak part of the problem. If `trs` has a rule whose right hand side contains a variable not contained in its left hand side, or if the usable rules cannot be oriented by an SLI, then None is returned. If both `forcedp` and `forcewdp` are false, then dependency pairs are chosen if all compound symbos are nullary, and weak innermost dependency pairs are chosen otherwise. If the flag `forcedp` is true, then None is returned whenever weak innermost dependency pairs would be chosen. If `forcewdp` is true, then weak innermost dependency pairs are always chosen.

```
val wdp_nosli_std : Trs.t -> proof option
```

`wdp_nosli_std trs n i` transforms `trs` into a DP problem, where the weak dependency pairs are the strict part, and its usable rules are the weak part of the problem. If `trs` has a rule whose right hand side contains a variable not contained in its left hand side, or if the usable rules cannot be oriented by an SLI, then None is returned.

```
val wdp_redpair_inn :
  bool -> bool -> Trs.t -> int -> bool -> redpairproof option
```

`wdp_redpair_inn forcedp forcewdp trs n i` transforms `trs` into a DP problem, where the weak innermost dependency pairs or dependency pairs are the strict part, and the usable rules are the weak part of the problem. If `trs` has a rule whose right hand side contains a variable not contained in its left hand side, or if the usable rules cannot be oriented by an SLI, then None is returned. If both `forcedp` and `forcewdp` are false, then dependency pairs are chosen if all compound symbos are nullary, and weak innermost dependency pairs are chosen otherwise. If the flag `forcedp` is true, then None is returned whenever weak innermost dependency pairs would be chosen. If `forcewdp` is true, then weak innermost dependency pairs are always chosen. `n` and `i` are two parameters for searching the SLI: `n` fixes the upper bound on the coefficients of the SLI, and `i` specifies whether `n` should be fully used immediately (if `i` is false), or approached incrementally (otherwise).

`val wdp_redpair_std : Trs.t -> int -> bool -> redpairproof option`

`wdp_redpair_std trs n i` transforms `trs` into a DP problem, where the weak dependency pairs are the strict part, and its usable rules are the weak part of the problem. If `trs` has a rule whose right hand side contains a variable not contained in its left hand side, or if the usable rules cannot be oriented by an SLI, then None is returned. `n` and `i` are two parameters for searching the SLI: `n` fixes the upper bound on the coefficients of the SLI, and `i` specifies whether `n` should be fully used immediately (if `i` is false), or approached incrementally (otherwise).

`val wdp_std : Trs.t -> proof option`

`wdp_std trs` transforms `trs` into the union of its weak dependency pairs and the usable rules of its weak dependency pairs. If `trs` has a rule whose right hand side contains a variable not contained in its left hand side, then None is returned.

6.2 Module Wdg : Weak Dependency graph approximations

Author(s): Andreas Schnabl

Description This module is a wrapper for the module EDG of TTT2, which does some pre- and postprocessing in order to ensure that the specialties of WDP (several DP-symbols in parallel positions RHSs in DP, RHSs in DP without any DP-symbols) do no harm in the EDG functions

`type proof = Rule.t Graph.t`

`val edg : Trs.t -> Trs.t -> proof`

`edg trs dp` returns the EDG of `dp`.

`val eidg : Trs.t -> Trs.t -> proof`

`eidg trs dp` returns the EIDG of `dp`.

`val edg_star : Trs.t -> Trs.t -> proof`

`edg_star trs dp` returns the EDG* of `dp` using `trs`.

`val eidg_star : Trs.t -> Trs.t -> proof`

`eidg_star trs dp` returns the EIDG* of `dp`.

`val edg_star_star : Trs.t -> Trs.t -> proof`

`edg_star_star trs dp` return the EDG** of `dp` using `trs`.

`val eidg_star_star : Trs.t -> Trs.t -> proof`

`eidg_star_star trs dp` returns the EIDG** of `dp`.

`val edg_star_star_star : Trs.t -> Trs.t -> proof`

`edg_star_star_star trs dp` returns the EDG*** of `dp` using `trs`.

`val eidg_star_star_star : Trs.t -> Trs.t -> proof`

`eidg_star_star_star trs dp` returns the EIDG*** of `dp` using `trs`.

`val fprintf : Format.formatter -> proof -> unit`

`fprintf ppf p` pretty-prints `p` to `ppf`.

6.3 Module `Maxpath` : Maximal Path

Author(s): Andreas Schnabl

Proof type for a single WDP problem emerging from maximal path analysis

```
type proof = Trs.t * Trs.t * Cdiprove.proof
val fprintf : Format.formatter -> proof -> unit
```

`fprintf ppf p` pretty prints the proof `p` to the formatter `ppf`.

6.4 Module `Maxpaths` : Maximal Paths

Author(s): Andreas Schnabl

This method uses dependency graphs and maximal paths in that graph to split a WDP problem into several smaller WDP problems.

```
type proof = (Trs.t * Trs.t * Cdiprove.proof) list
exception This_should_not_happen of string
val fprintf : Format.formatter -> proof -> unit
```

`fprintf ppf p` pretty prints the proof `p` to the formatter `ppf`.

`val solve : Rule.t Graph.t -> Trs.t -> int -> bool -> proof option`

`solve wdg weak bound iter` decomposes the given weak dependency graph into a number of smaller WDP problems. For each path in the graph starting from a source node, a subproblem is created. Moreover, for each maximal path in the graph, a compatible SLI for all usable rules for that path has to be found. The options `bound` and `iter` determine the constrains for searching the SLIs: `bound` denotes the maximal coefficient, and `iter` determines whether the given bound shall be approached iteratively. If one of the SLIs can not be found, then None is returned.

7 Utilities

7.1 Module `Listxx` : Extended Extended lists

Author(s): Andreas Schnabl

This is an extension of the listx module from ocamlutils, which is an extension of the list module of the OCaml standard library.

```
val get_ith_element : int -> 'a list -> 'a
```

get_ith_argument i xs returns the ith element of xs. If the length of xs is less than i, Invalid_argument is raised.

```
val iter3 : ('a -> 'b -> 'c -> unit) -> 'a list -> 'b list -> 'c list -> unit
```

iter3 f xs ys zs executes in turn f x1 y1 z1, f x2 y2 z2, and so on. Raises Invalid_argument if xs, ys and zs do not have the same length.

```
val map3 : ('a -> 'b -> 'c -> 'd) -> 'a list -> 'b list -> 'c list -> 'd list
```

map3 f xs ys zs builds a list with elements f x1 y1 z1, f x2 y2 z2, and so on. Raises Invalid_argument if xs, ys and zs do not have the same length.

```
val split3 : ('a * 'b * 'c) list -> 'a list * 'b list * 'c list
```

split3 xs, where xs is a list of triples, returns three lists consisting of the first, second or third element of the triples in xs, respectively.

```
val split4 :
  ('a * 'b * 'c * 'd) list -> 'a list * 'b list * 'c list * 'd list
```

split4 xs, where xs is a list of 4-tuples, returns four lists consisting of the first, second, third or fourth element of the tuples in xs, respectively.

7.2 Module Setxx : Extended Setx

Author(s): Andreas Schnabl

This is an extension of the Setx module from ocamlutils, which is an extension of the Set module from the standard library.

```
val of_list : ?c:('a -> 'a -> int) -> 'a list -> 'a Setx.t
```

of_list l transforms the list l into a set, and allows an optional argument specifying the comparison function on the created set.

7.3 Module Graphx : Extended Graph

Author(s): Andreas Schnabl

This is an extension of the graph module from ocamlutils.

```
module MyNode :
  sig
```

```
    type t = Rule.t Setx.t
    val compare : t -> t -> int
    val equal : t -> t -> bool
    val hash : t -> int
  end
module SetGraph :
  Graph.GRAPH  with type elt = Rule.t Setx.t
val maximal_paths : SetGraph.t -> Rule.t Setx.t list list
```

`maximal_paths g` returns a list of maximal paths of `g`, where each path is represented by the list of its node, starting with the last node of the path, and ending with the first node.

```
val noncompound_rhs_union_graph : Rule.t Graph.t -> Trs.t -> Rule.t Graph.t
```

`noncompund_rhs_union_graph g trs` computes a graph from `g` where for each rule in `trs`, the nodes corresponding to the noncompound subterms of the ruls's rhs are collapsed into a single node.

```
val scc_equiv_graph : Rule.t Graph.t -> SetGraph.t
```

`scc_equiv_graph g` computes a graph from `g` where each scc is collapsed into a single node.

7.4 Module `Termxx` : Extended Term

Author(s): Andreas Schnabl

This is an extension of the term module from trslib.

```
val noncompound_subterms : Trs.t -> Term.t -> Term.t list
```

`noncompound_subterms trs t` returns a list of subterms of `t` such no root symbol of any term in that list is a compound symbol, and the original term can be recovered by combining the terms in the list only by compound symbols.

7.5 Module `Rulex` : Extended Rule

Author(s): Andreas Schnabl

This is an extension of the Rule module from trslib.

```
val is_nonincreasing : Rule.t -> bool
```

`is_nonincreasing rule` returns true iff `rule` is nonincreasing, i.e. its lhs is greater or equal to its rhs with respect to term size

```
val is_dp_rule : Rule.t -> bool
```

`is_dp_rule rule` returns true iff `rule` is a dependency pair. This is checked by the criterion that the root its lhs is a DP symbol.

7.6 Module Trsx : Extended TRS

Author(s): Andreas Schnabl

This is an extension of the trs module from trslib.

`val big_union : Trs.t Setx.t -> Trs.t`

`big_union trss` returns the union of the TRSs given in `trss`

`val get_compound_symbols : Trs.t -> FunctionSymbol.t Setx.t`

`get_compound_symbols trs` returns all compound function symbols of `trs`, i.e. all function symbols occurring above dependency pair symbols on the right hand side of a weak dependency pair

`val is_compound_symbol : FunctionSymbol.t -> Trs.t -> bool`

`is_compound_symbol f trs` returns true iff `f` is a compound function symbol in `trs`, i.e. if it occurs above a dependeny pair symbol on the right hand side of a weak dependency pair

`val is_constructor_symbol : FunctionSymbol.t -> Trs.t -> bool`

`is_constructor_symbol f trs` returns true iff `f` is a constructor symbol in `trs`, ie. if it not a defined function symbol

`val is_defined_symbol : FunctionSymbol.t -> Trs.t -> bool`

`is_defined_symbol f trs` returns true iff `f` is a defined function symbol in `trs`, or if the corresponding dependency pair symbol occurs in `trs`

`val is_nonincreasing : Trs.t -> bool`

`is_nonincreasing trs` returns true iff `trs` contains only nonincreasing rules, i.e. rules whose lhs is greater or equal to its rhs with respect to term size

`val is_dp_trs : Trs.t -> bool`

`is_dp_trs trs` returns true iff `trs` is a set of dependency pairs. This is checked by the criterion that the root of the lhs of some selected rule is a DP symbol.

7.7 Module VariadicFunctionSymbol : Function Symbols with multiple arities.

`type t`

deprecated, will be deleted asap.

`val compare : t -> t -> int`

`val fprintf : Format.formatter -> t -> unit`

```
val make : FunctionSymbol.t -> int -> t
val root_symbol : Term.t -> t
val term_function_symbols : Term.t -> t Setx.t
val trs_function_symbols : Trs.t -> t Setx.t
val is_defined_symbol : Trs.t -> t -> bool
val is_constructor_symbol : Trs.t -> t -> bool
val arity : t -> int
val symbol : t -> FunctionSymbol.t
val label_with : t -> int -> t
val label : t -> int
val argument_positions : t -> int list
```

7.8 Module `Algebra` : Set of interpretations over a signature.

```
type t
val fprintf : Format.formatter -> t -> unit
```

Pretty printing functionality

```
val add : VariadicFunctionSymbol.t -> FiniteFnEncoding.t -> t -> t
```

Associate a variadic function symbol with an interpretation function. Observe that `FiniteFnEncoding.t` is the type of a concrete instance of an interpretation function, and not its encoding

```
val fold :
  (VariadicFunctionSymbol.t -> FiniteFnEncoding.t -> 'a -> 'a) ->
  t -> 'a -> 'a
```

Standard `fold` for algebras

```
val empty : t
```

The algebra for the empty signature

```
val lookup : t -> VariadicFunctionSymbol.t -> FiniteFnEncoding.t
```

`lookup algebra symbol` returns the interpretation associated with `symbol`

7.9 Module `Assign` : This module provides an implementation for assignments from variables to a specified carrier.

```
module type Codomain =
  sig
```

7.9 Module `Assign` : This module provides an implementation for assignments from variables to a specified carrier.

This module defines the codomain C of an assignment $\alpha\colon \mathcal{V} \to C$. In order to decide whether $\alpha = \beta$ for two assignments α, β the codomain needs to be comparable.

`type t`

The datatype of the codomain

`val compare : t -> t -> int`

C.f. `Pervasives.compare`.

`val fprintf : Format.formatter -> t -> unit`

Pretty printer for elements of type `t`. Employed for pretty printing assignments.

`end`

`module Make :`
`functor (C : Codomain) -> sig`

Given a module of type `Codomain`, this functor constructs an assignment module.

`type t`

The type of an assignment

`val empty : t`

The empty assignment

`val extend : t -> Variable.t -> C.t -> t`

`extend alpha v e` extends the assignment alpha by mapping the variable `v` to the value `e`.

`val domain : t -> Variable.t Setx.t`

`domain alpha` returns the domain of the assignment `alpha`.

`val codomain : t -> C.t Setx.t`

`codomain alpha` returns the codomain of the assignment `alpha`.

`val lookup : t -> Variable.t -> C.t`

`lookup alpha v` returns `alpha(v)`.
Raises `Not_found` when `alpha(v)` is undefined

`val enumeration : Variable.t Setx.t -> C.t Setx.t -> t list`

`enumerations variables codomain` returns an enumeration of all assignments ranging over `variables` with codomain `codomain`.

```
    val fprintf : Format.formatter -> t -> unit
```

Pretty printing function for assignments.

```
    val compare : t -> t -> int
```

C.f. `Pervasives.compare`

```
    val of_list : (Variable.t * C.t) list -> t
```

`of_list l` translates the assoc-list `l` to an assignment.

```
    val fold : (Variable.t -> C.t -> 'a -> 'a) -> t -> 'a -> 'a
```

Standard `fold` for assignments.

```
  end
module Nat :
  sig
    type t = int
```

Natural numbers as codomain.

```
    val compare : t -> t -> int
    val fprintf : Format.formatter -> t -> unit
  end
module NatAssign :
  sig
    type t
```

Assignments from variables to naturals. For documentation, c.f. module `Assign.Make`.

```
    val empty : t
    val extend : t -> Variable.t -> int -> t
    val domain : t -> Variable.t Setx.t
    val codomain : t -> int Setx.t
    val lookup : t -> Variable.t -> int
    val enumeration : Variable.t Setx.t -> int Setx.t -> t list
    val fprintf : Format.formatter -> t -> unit
    val compare : t -> t -> int
    val of_list : (Variable.t * int) list -> t
    val fold : (Variable.t -> int -> 'a -> 'a) -> t -> 'a -> 'a
  end
```

7.10 Module CPutils : Standard utility functions

```
val uncurry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
val curry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
val liftPair : ('a -> 'b) -> 'a * 'a -> 'b * 'b
val pair : 'a -> 'b -> 'a * 'b
val flip : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
val zipWith : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
val zip : 'a list -> 'b list -> ('a * 'b) list
val indexed : ?start:int -> 'a list -> (int * 'a) list
val upto : int -> int -> int list
val id : 'a -> 'a
```

7.11 Module Debug : when debugmessages = true debugging messages are emitted

```
val debugmessages : bool
```

when `debugmessages = true` debugging messages are emitted

```
val asserts : bool
```

when `asserts = true` asserting statements are executed.

```
val asserting : bool Lazy.t -> unit
```

`asserting f` takes a lazyily evaluated predicate which is checked provided `asserts = true`.

7.12 Module PredicativeRewriting : Utility functions for predicative systems

```
val fprintf : SafeMapping.t -> Format.formatter -> Trs.t -> unit
```

This function is a pritty printer for term rewrite systems in predicative style: For a safe mapping safe , a term $\mathsf{f}(t_1, \ldots, t_n)$ is printed as $\mathsf{f}(t_{i_1}, \ldots, t_{i_p}; t_{j_1}, \ldots, t_{j_q})$ where $\mathsf{safe}(\mathsf{f}) = [i_1, \ldots, i_p]$. Observe that argument positions may be permuted.

7.13 Module SafeMapping : Implementation of safe mappings

```
type t
val fprintf : Format.formatter -> t -> unit
```

Pretty printer

```
val safe_positions : VariadicFunctionSymbol.t -> t -> int Setx.t
```

`safe_positions f safe` returns the set of safe positions of the symbol `f`.

```
val normal_positions : VariadicFunctionSymbol.t -> t -> int Setx.t
```

`normal_positions f safe` returns the set of safe positions of the symbol `f`.

```
val is_safe : VariadicFunctionSymbol.t -> int -> t -> bool
```

`is_safe f positions safe` returns true iff the position `position` is a safe position of `f`.

```
val is_normal : VariadicFunctionSymbol.t -> int -> t -> bool
```

`is_normal f positions safe` returns true iff the position `position` is a normal position of `f`.

```
val empty : t
```

The empty safe mapping. That is, all positions are normal for each function symbol.

```
val set_safe : VariadicFunctionSymbol.t -> int -> t -> t
```

`set_safe f position safe` returns a new safe mapping based on `safe` where additionally the argument position `position` of the symbol `f` is safe.

```
val set_normal : VariadicFunctionSymbol.t -> int -> t -> t
```

`set_normal f position safe` returns a new safe mapping based on `safe` where additionally the argument position `position` of the symbol `f` is normal.

```
val of_list : (VariadicFunctionSymbol.t * int) list -> t
```

`of_list assoc` translates the assoc list `assoc` to a safe mapping. For an element `(f,pos)` the position `pos` is regarded as a safe position of `f`.

```
val apply_argument_filtering :
  ArgumentFiltering.elt ArgumentFiltering.t -> t -> t
```

`apply_argument_filtering af safe` returns a new safe mapping filtered according to `af`. That is, only those argument positions are considered safe that are unfiltered according to the argument filtering `af`.