Contents

1 Module Ast_util: util over untyped AST (for comprehensions) 3
2 Module Backend: generate code for various backends 4
3 Module Backend_common: Functions used by multiple backends 5
4 Module Coq_decidable_equality 8
5 Module Def_trans: Infrastructure form definition macros 8
  5.1 Infrastructure form definition macros ........................................... 8
  5.2 Dictionary passing ........................................................................ 9
  5.3 Open / Include / Import .................................................................. 9
  5.4 Misc .......................................................................................... 10
6 Module Finite_map: finite map library 10
7 Module Ident: source-file long identifiers 13
8 Module Initial_env: The initial environment. 14
9 Module Macro_expander 15
10 Module Main 16
11 Module Module_dependencies: module dependency resolution 16
12 Module Name: source-file and internal short identifiers 16
  12.1 plain names ..................................................................................... 16
  12.1.1 basic functions on plain names ...................................................... 16
  12.1.2 modifying names ......................................................................... 17
  12.1.3 generating fresh names ................................................................ 17
  12.2 names with whitespace an type ....................................................... 18
12.3 output functions ............................................................................. 19
13 Module Nvar 19
14 Module Output: Intermediate output format before going to strings 19
  14.1 constructing output ...................................................................... 20
  14.2 Pretty Printing .............................................................................. 21
  14.2.1 Blocks ...................................................................................... 21
  14.2.2 Spacing .................................................................................... 22
  14.3 Output to Rope ............................................................................. 22
  14.4 Latex Output .............................................................................. 23
15 Module Path: internal canonical long identifiers 23
31.4 Pattern Macros .................................................. 48
31.5 Type Class Macros .................................................. 48
31.6 Misc .............................................................. 49
31.7 Macros I don’t understand ...................................... 49

32 Module Typecheck : check_defs backend_targets mod_name filename mod_in_output
env ast typescheck the parsed module ast from file filename in environment env. 49

33 Module Typecheck_ctxt : The distinction between cur_env, new_defs and export_env
is interesting. 50

34 Module Typed_ast : Sets of Names 52

35 Module Typed_ast_syntax : syntax functions for typed_ast 75
  35.1 Types .......................................................... 75
  35.2 Navigating Environments ...................................... 75
  35.3 target-representations ........................................ 79
  35.4 Constructing, checking and destructing expressions ............. 80
  35.5 Constructing, checking and destructing definitions ............. 85
  35.6 Collecting information about uses constants, types, modules ....... 85
  35.7 Miscellaneous .................................................. 86

36 Module Types : Structural comparison of types, without expanding type abbreviations. 88

37 Module Tyvar : type of internal(?) type variables 97

38 Module Util : Mixed useful things 98
  38.1 Option Functions .................................................. 98
  38.2 List Functions .................................................... 99
  38.3 Files ........................................................... 101
  38.4 Strings .......................................................... 102
  38.5 Useful Sets ....................................................... 103

1 Module Ast_util : util over untyped AST (for comprehensions)

val setcomp_bindings : (Name.t -> bool) -> Ast.exp -> Set.Make(Name).t
  Infer the comprehension variables for a set comprehension without explicitly listed
  comprehension variables. The first argument should return true for variables that are
  currently bound in the enclosing environment (such variables cannot become comprehension
  variables)

val get_imported_modules : Ast.defs * Ast.lex_skips -> (Path.t * Ast.l) list
get_imported_modules ast returns a list of the modules imported by the given definitions. These are modules that are explicitly imported via an import statement. The resulting list may contain duplicates and is not sorted in any way.

2 Module Backend: generate code for various backends

val gen_extra_level : int Pervasives.ref
The level of extra information to generate

module Make :
functor (C : sig
  val avoid : Typed_ast.var_avoid_f
  val env : Typed_ast.env
  val dir : string
      the directory the output will be stored. This is important for setting relative paths to import other modules
end ) -> sig

val ident_defs : Typed_ast.def list * Ast.lex_skips -> Ulib.Text.t
val lem_defs : Typed_ast.def list * Ast.lex_skips -> Ulib.Text.t
val hol_defs :
    Typed_ast.def list * Ast.lex_skips -> Ulib.Text.t * Ulib.Text.t option
val ocaml_defs :
    Typed_ast.def list * Ast.lex_skips -> Ulib.Text.t * Ulib.Text.t option
val isa_defs :
    Typed_ast.def list * Ast.lex_skips -> Ulib.Text.t * Ulib.Text.t option
val isa_header_defs : Typed_ast.def list * Ast.lex_skips -> Ulib.Text.t
val coq_defs :
    Typed_ast.def list * Ast.lex_skips -> Ulib.Text.t * Ulib.Text.t
val tex_defs : Typed_ast.def list * Ast.lex_skips -> Ulib.Text.t
val tex_inc_defs :
    Typed_ast.def list * Ast.lex_skips -> Ulib.Text.t
val html_defs : Typed_ast.def list * Ast.lex_skips -> Ulib.Text.t
val ident_exp : Typed_ast.exp -> Ulib.Text.t
val ident_pat : Typed_ast.pat -> Ulib.Text.t
val ident_src_t : Types.src_t -> Ulib.Text.t
val ident_typ : Types.t -> Ulib.Text.t
val ident_def : Typed_ast.def -> Ulib.Text.t
The various backends that generate text from typed asts

3 Module Backend_common : Functions used by multiple backends

val def_add_location_comment_flag : bool Pervasives.ref
    def_add_location_comment_flag controls whether def_add_location_comment.

val def_add_location_comment : Typed_ast.def -> Output.t * Typed_ast.def_aux
    If def_add_location_comment_flag is set, def_add_location_comment d adds a comment
    with location information before definition d. This may require changing the initial
    whitespace before the definition. Therefore, the def_aux of d with changed whitespace as
    well as the output that should be added before d is returned.

val inline_exp_macro :
    Target.non_ident_target ->
    Typed_ast.env ->
    Macro_expander.macro_context -> Typed_ast.exp -> Typed_ast.exp option
    inline_exp_macro target env does the inlining of target specific constant definitions

val inline_pat_macro :
    Target.non_ident_target ->
    Typed_ast.env -> 'a -> 'b -> Typed_ast.pat -> Typed_ast.pat option
    inline_pat_macro target env does the inlining of target specific constant definitions

val component_to_output : Ast.component -> Output.t
    component_to_output c formats component c as an output

val get_module_name :
    Typed_ast.env -> Target.target -> Name.t list -> Name.t -> Name.t
    get_module_name env targ mod_path mod_name looks up the name of module
    mod_path.mod_name in environment env for target targ.

val get_module_open_string :
    Typed_ast.env -> Target.target -> string -> Path.t -> string
    get_module_open_string l env targ dir mod_path looks up how to represent this
    module in import / open statements.

val isa_add_full_library_path_flag : bool Pervasives.ref
isa_add_full_library_path_flag controls whether the full path to directory isabelle-lib is added to Lem library modules imported by files outside the lib. This is useful to process the files easily from Isabelle, without providing the path these library modules can be found explicitly to Isabelle. The resulting files are however not directly portable. Therefore, this flag is turned off by default and it is recommended to add the isabelle-library directory instead to your .isabelle/VERSION/ROOTS file.

val get_imported_target_modules :
Typed_ast.def list * Ast.lex_skips -> Typed_ast.imported_modules list
get_imported_target_modules env targ defs extracts a list of module that should be imported. The exact names of these modules depend on the environment and the target. Therefore, they get extracted in an abstract from and converted (after possible changes to the environment) by imported_modules_to_strings.

val imported_modules_to_strings :
Typed_ast.env ->
Target.target -> string -> Typed_ast.imported_modules list -> string list
imported_modules_to_strings env targ dir imported_mods is used together with get_imported_target_modules. Please see there.

module Make :
functor (A : sig
val env : Typed_ast.env
val target : Target.target
val dir : string
val id_format_args :
  (bool -> Output.id_annot -> Ulib.Text.t -> Output.t) * Ulib.Text.t
end ) -> sig

val open_to_open_target :
Path.t Types.id list -> (Typed_ast.1skips * string) list * Typed_ast.1skips
val function_application_to_output :
Ast.1 ->
(Typed_ast.exp -> Output.t) ->
bool ->
Typed_ast.exp ->
Types.const_descr_ref Types.id -> Typed_ast.exp list -> bool -> Output.t list

function_application_to_output l exp inf full_exp c_id args tries to format a function application as output. It gets an expression full_ex of the from \( c \ arg_1 \ldots arg_n \). The id c_id corresponds to constant \( c \). The arguments \( arg_1, \ldots arg_n \) are handed over as args. The description corresponding to \( c \) is looked up in A.env. Depending on this description and the backend-specific formats therein, the function and its arguments are formated as output. In the simplest case the representation is an
identifier (Ident.t), which is formatted using A.id_format_args and the information, whether it the whole expression is an infix one inf. In more complicated cases, formatting of expressions is needed, which is done via the callback exp. In particular if some arguments are not needed by the formatting of the function application, the function exp is called on these remaining arguments. The original expression full_exp is needed, if not enough parameters are present to format the definition correctly. In this case, eta-expansion is applied and the resulting expression formatting via exp. ascii_alternative denotes whether an ascii alternative representation for this function name is required.

val pattern_application_to_output : Ast.l -> (Typed_ast.pat -> Output.t) -> Types.constdescr_ref Types.id -> TypedList.pat list -> bool -> Output.t list

pattern_application_to_output l pat c_id args tries to format a function application in a pattern as output. It does otherwise the same as function_application_to_output. However, since there are no infix patterns, the parameter inf is always set to false.

val const_id_to_ident : Types.constdescr_ref Types.id -> bool -> Ident.t

const_id_to_ident c_id use_ascii tries to format a constant, constructor or field c_id as an identifier for target A.target using the rules stored in environment A.env. If the flag use_ascii is set, the ascii representation of the constant should be used, if there is one. Depending on the formatting rules for the constant, const_id_to_ident might raise an exception.

val const_ref_to_name : Name.lskips_t -> bool -> Types.constdescr_ref -> Name.lskips_t

const_ref_to_name n use_ascii c tries to format a constant c for target A.target using the rules stored in environment A.env. If use_ascii is set, the ascii-representation is returned. const_ref_to_name always returns a name n'. If special formatting rules are installed, this name might not be the one used by function_application_to_output, though. The argument n is the name used in the original input. It’s whitespace is used to format n'.

val type_path_to_name : Name.lskips_t -> Path.t -> Name.lskips_t

type_path_to_name n p tries to format a type-path p for target A.target using the rules stored in environment A.env. It always returns a name n'. If special formatting rules are installed, this name might not be the one used by function_application_to_output, though. The argument n is the name used in the original input. It’s whitespace is used to format n'.

val type_id_to_ident : Path.t Types.id -> Ident.t
type_id_to_ident ty_id tries to format a type ty_id as an identifier for target A.target using the rules stored in environment A.env.

val type_id_to_output : Path.t Types.id -> Output.t

type_id_to_output ty_id tries to format a type ty_id as an identifier for target A.target using the rules stored in environment A.env.

val type_id_to_ident_no_modify : Path.t Types.id -> Ident.t

type_id_to_ident_no_modify ty_id formats ty_id as an identifier. In contrast to type_id_to_ident neither the target A.target nor the rules stored in environment A.env are used. Instead the type is translated without any modifications. This method is intended to be used for backend types, which are already formatted.

val type_app_to_output :
  (Types.src_t -> Output.t) ->
  Path.t Types.id -> Types.src_t list -> Types.src_t list * Output.t

val module_id_to_ident : Path.t Types.id -> Ident.t

module_id_to_ident m_id tries to format a module m_id as an identifier for target A.target using the rules stored in environment A.env.

end

4 Module Def_trans : Infrastructure form definition macros

4.1 Infrastructure form definition macros

type def_macro = Name.t list ->
  Typed_ast.env -> Typed_ast.def -> (Typed_ast.env * Typed_ast.def list) option

  def_macro is the type of definition macros. A definition macro def_mac gets the arguments rev_path, env and d. The argument d is the definition the macro should process. rev_path represents the path of the module of definition d as a list of names in reverse order. env is the local environment for the module of d. This means that also the definitions in the same module that follow d are present. If the macro does not modify the definition, it should return None. Otherwise, it should return a pair Some (env', ds), where env' is a updated environment and ds a list of definitions that replace d.

val list_to_mac : def_macro list -> def_macro

  list_to_mac macro_list collapses a list of def_macros into a single one. It looks for the first macro in the list that succeeds, i.e. returns not None and returns the result of this macro.
val process_defs : Name.t list -> def_macro -> Name.t -> Typed_ast.env -> Typed_ast.def list -> Typed_ast.env * Typed_ast.def list

process_defs rev_path def_mac mod_name env ds is intended to run the macro def_mac over all definitions in module mod_name. The argument rev_path is the path to module mod_name in reversed order. env is the environment containing module mod_name and ds is the list of definitions in this module. If def_mac modifies a definition d to a list ds, it is then run on all definitions in ds. If one of the is a module-definition, which is not modified by ds, then def_macro is run on all definitions inside this module. For this recursive call the path, module name and environment are adapted.

The result of process_defs is an updated environment and a new list of definitions.

4.2 Dictionary passing

val class_to_record : Target.target -> def_macro

Type classes are not supported by all backends. The def_macro class_to_record takes a definition of a type class and turns it into a definition of a record type. The methods of the class become field of the record. This record can then be used as the dictionary type for the dictionary passing.

val comment_out_inline_instances_and_classes : Target.target -> def_macro

Removes inline instances for backends that employ typeclasses.

val instance_to_dict : bool -> Target.target -> def_macro

instance_to_dict do_inline targ turns instance declarations into a definition of a dictionary record. If do_inline is set, this definition will be inlined (for this the target argument is needed).

val class_constraint_to_parameter : Target.target -> def_macro

4.3 Open / Include / Import

val remove_opens : def_macro
remove_opens removes all open / include and import statements

val remove_import_include : def_macro
remove_import_include removes all import and include statements. Imports are deleted and includes turned into open statements.

val remove_import : def_macro
remove_import removes all import statements.

val remove_module_renames : def_macro
remove_module_renames removes all module rename statements.
4.4 Misc

val remove_types_with_target_rep : Target.target -> def_macro

If a target representation for a type is given, the original type definition is commented out.
Notice that target-specific renamings are not target representations.

val defs_with_target_rep_to_lemma :
Typed_ast.env -> Target.target -> def_macro

If a target representation for a constant is given, the original definition is not needed.
However, turn this definition into a lemma to ensure that the target representation is
sensible.

val remove_vals : def_macro
val remove_indrelns : def_macro
val remove_indrelns_true_lhs : def_macro
val remove_classes : def_macro
val type_annotate_definitions : def_macro
val nvar_to_parameter : def_macro
val prune_target_bindings :
Target.non_ident_target -> Typed_ast.def list -> Typed_ast.def list

5 Module Finite_map : finite map library

module type Fmap =

sig

  type k
  module S :
    Set.S with type elt = k
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val from_list : (k * 'a) list -> 'a t
  val from_list2 : k list -> 'a list -> 'a t
  val insert : 'a t -> k * 'a -> 'a t
  val union : 'a t -> 'a t -> 'a t
  val big_union : 'a t list -> 'a t
  val merge :
    (k -> 'a option -> 'b option -> 'c option) ->
    'a t -> 'b t -> 'c t
  val apply : 'a t -> k -> 'a option
val in_dom : k -> 'a t -> bool
val map : (k -> 'a -> 'b) -> 'a t -> 'b t
val domains_overlap : 'a t -> 'b t -> k option
val domains_disjoint : 'a t list -> bool
val iter : (k -> 'a -> unit) -> 'a t -> unit
val fold : ('b -> k -> 'a -> 'b) -> 'b -> 'a t -> 'b
val filter : (k -> 'a -> bool) -> 'a t -> 'a t
val remove : 'a t -> k -> 'a t
val pp_map :
  (Format.formatter -> k -> unit) ->
  (Format.formatter -> 'a -> unit) ->
  Format.formatter -> 'a t -> unit
val domain : 'a t -> S.t
end

module Fmap_map :
  functor (Key : Set.OrderedType) -> sig
    type k = Key.t
  module S :
    Set.Make(Key)
  type 'a t = 'a M.t
  val empty : 'a M.t
  val is_empty : 'a M.t -> bool
  val from_list : (M.key * 'a) list -> 'a M.t
  val from_list2 : M.key list -> 'a list -> 'a M.t
  val insert : 'a M.t ->
    M.key * 'a -> 'a M.t
  val union : 'a M.t ->
    'a M.t -> 'a M.t
  val merge :
    (M.key -> 'a option -> 'b option -> 'c option) ->
    'a M.t ->
    'b M.t -> 'c M.t
  val apply : 'a M.t -> M.key -> 'a option
  val in_dom : M.key -> 'a M.t -> bool
  val map : (M.key -> 'a -> 'b) ->
    'a M.t -> 'b M.t
  val domains_overlap : 'a M.t ->
module type Dmap =

  sig
    type k
    type 'a t
    val empty : 'a t
    val set_default : 'a t -> 'a option -> 'a t
    val insert : 'a t -> k * 'a -> 'a t
    val insert_opt : 'a t -> k option * 'a -> 'a t
    val apply : 'a t -> k -> 'a option
    val apply_opt : 'a t -> k option -> 'a option
    val remove : 'a t -> k -> 'a t
    val in_dom : k -> 'a t -> bool
  end

module Dmap_map :
  functor (Key : Set.OrderedType) -> sig
    type k = Key.t
    type 'a t = 'a M.t * S.t * 'a option
    val empty : 'a M.t * S.t * 'b option
    val set_default : 'a * 'b * 'c option * 'd -> 'a * 'b * 'd
    val apply : 'a M.t * S.t * 'a option ->
                 M.key -> 'a option
  end
val apply_opt : 'a M.t * S.t * 'a option -> M.key option -> 'a option
val in_dom : M.key -> 'a M.t * S.t * 'b option -> bool
val insert : 'a M.t * S.t * 'b -> M.key * 'a -> 'a M.t * S.t * 'b
val insert_opt : 'a M.t * S.t * 'a option -> M.key option * 'a -> 'a M.t * S.t * 'a option
val remove : 'a M.t * S.t * 'b -> M.key -> 'a M.t * S.t * 'b
end

6 Module Ident: source-file long identifiers

type t
  t is the type of dot separated lists of names (with preceding lexical spacing), e.g. (*Foo*) M . x
val pp : Format.formatter -> t -> unit
  Pretty print
val to_string : t -> string
  to_string i formats i using pp.
val from_id : Ast.id -> t
val from_name : Name.lskips_t -> t
val get_name : t -> Name.lskips_t
  Return the last name in the ident, e.g., M.Y.x gives x
val mk_ident : Ast.lex_skips -> Name.t list -> Name.t -> t
  mk_ident sk ms n creates an identifier n with module prefix ms and leading whitespace sk.
val mk_ident_ast : (Name.lskips_t * Ast.lex_skips) list -> Name.lskips_t -> Ast.l -> t
  mk_ident_ast nsl ns l generates a new identifiers during type-checking. Whitespace is prohibited in all Name.lskips_t except the very first one and all Ast.lex_skips has to be empty. Otherwise, this operation my fails and uses the location l for the error message.
val mk_ident_strings : string list -> string -> t

*mk_ident_strings* is a version of *mk_ident* that uses strings as input and uses empty whitespace.

val to_output_format :
  (Output.id_annot -> Ulib.Text.t -> Output.t) ->
  Output.id_annot -> Ulib.Text.t -> t -> Output.t

val to_output : Output.id_annot -> Ulib.Text.t -> t -> Output.t

val get_lskip : t -> Ast.lex_skips

val replace_lskip : t -> Ast.lex_skips -> t

val to_name_list : t -> Name.t list * Name.t

val has_empty_path_prefix : t -> bool

  *has_empty_path_prefix* i check whether the identifier *i* consists of just a single name without any prefix describing its module path

val strip_path : Name.t -> t -> t

  Remove the name from the identifier if it occurs at the first

val rename : t -> Name.t -> t

  rename *i* *n* renames the last name component of identifier *i* to *n*.

val drop_path : t -> t

  *drop_path* *i* drops the path of an identifier. This means an identifier of the form *M1.M2...Mn.name* is converted to *name*. White-space is preserved.

7 Module Initial env: The initial environment.

It is empty except bindings for predefined things like *bool*

val initial_env : Typed_ast.env

val read_target_constants : string -> Target.target -> Typed_ast.NameSet.t

  *read_target_constants* *lib_path* *target* reads the list of constants that should be avoided for target *target*. These constants are read from a file *lib_path/\{target\}_constants*. If this file does not exists, the empty set is returned.

8 Module Macro expander

    type level =
      | Top_level
      | Nested
type pat_pos =
  | Bind
  | Param

type macro_context =
  | Ctxt_theorem
  | Ctxt_other

type pat_position = level * pat_pos

module Expander :
  functor (C : Typed_ast.Exp_context) -> sig

  val expand_defs :
    Typed_ast.def list ->
    (Macro_expander.macro_context -> Typed_ast.exp -> Typed_ast.exp option) *
    (Types.t -> Types.t) * (Types.src_t -> Types.src_t) *
    (Macro_expander.pat_position ->
      Macro_expander.macro_context -> Typed_ast.pat -> Typed_ast.pat option) ->
    Typed_ast.def list

  val expand_pat :
    Macro_expander.macro_context ->
    Macro_expander.pat_position ->
    Typed_ast.pat ->
    (Types.t -> Types.t) * (Types.src_t -> Types.src_t) *
    (Macro_expander.pat_position ->
      Macro_expander.macro_context -> Typed_ast.pat -> Typed_ast.pat option) ->
    Typed_ast.pat

  val expand_exp :
    Macro_expander.macro_context ->
    (Macro_expander.macro_context -> Typed_ast.exp -> Typed_ast.exp option) *
    (Types.t -> Types.t) * (Types.src_t -> Types.src_t) *
    (Macro_expander.pat_position ->
      Macro_expander.macro_context -> Typed_ast.pat -> Typed_ast.pat option) ->
    Typed_ast.exp -> Typed_ast.exp

end

val list_to_mac :
  (macro_context -> 'b -> 'c option) list ->
  macro_context -> 'b -> 'c option

val list_to_bool_mac :
  (pat_position ->
    macro_context -> 'b -> 'c option)
  list ->
  pat_position ->
  macro_context -> 'b -> 'c option
9 Module Main

10 Module Module_dependencies : module dependency resolution

val process_files :
  bool ->
  string list ->
  (string * bool) list ->
  (string * string * (Ast.defs * Ast.lexer_skips) * bool) list

process_files allow_reorder lib_dirs files parses the files in list files. It checks for
import statements and tries to automatically load the needed files for those as well.
Therefore, files are searched in the directories lib_dirs. If allow_reorder is set, it may
also reorder the order of file in files to satisfy dependencies.
The result is a list of tuples (module_name, filename, ast, needs_output). The flag
needs_output states, whether an output file should be produced. It is set to false for all
automatically imported modules. Since one might to also want to add library modules
manually, the input files is a list of file names and need-output flags as well.

11 Module Name : source-file and internal short identifiers

11.1 plain names

type t
  t is the type of plain names, names are essentially strings

val compare : t -> t -> int

11.1.1 basic functions on plain names

val pp : Format.formatter -> t -> unit
val from_string : string -> t
val to_string : t -> string
val from_rope : Ulib.Text.t -> t
val to_rope : t -> Ulib.Text.t

11.1.2 modifying names

val rename : (Ulib.Text.t -> Ulib.Text.t) -> t -> t
rename r_fun n renames n using the function r_fun. It looks at the text representation n_text of n and returns then the name corresponding to r_fun n_text.

val starts_with_upper_letter : t -> bool
    start_with_upper_letter n checks, whether the name n starts with a character in the range A-Z.

val uncapitalize : t -> t option
    uncapitalize n tries to uncapitalize the first letter of n. If n does not start with a uppercase character, None is returned, otherwise the modified name.

val starts_with_lower_letter : t -> bool
    start_with_lower_letter n checks, whether the name n starts with a character in the range a-z.

val capitalize : t -> t option
    capitalize n tries to capitalize the first letter of n. If n does not start with a lowercase character, None is returned, otherwise the modified name.

val starts_with_underscore : t -> bool
    start_with_underscore n checks, whether the name n starts with an underscore character.

val remove_underscore : t -> t option
    remove_underscore n tries to remove a leading underscores from name n. If n does not start with an underscore character, None is returned, otherwise the modified name.

val ends_with_underscore : t -> bool
    ends_with_underscore n checks, whether the name n ends with an underscore character.

val remove_underscore_suffix : t -> t option
    remove_underscore_suffix n tries to remove a suffix underscores from name n. If n does not end with an underscore character, None is returned, otherwise the modified name.

11.1.3 generating fresh names

val fresh : Ulib.Text.t -> (t -> bool) -> t
    fresh n OK generates a name m, such that OK m holds. m is of the form n followed by an integer postfix. First n without postfix is tried. Then counting up from 0 starts, till OK is satisfied.

val fresh_num_list : int -> Ulib.Text.t -> (t -> bool) -> t list
    fresh_num_list i n OK generates a list of i fresh names. If no conflicts occur it returns a list of the form [ni, n(i-1), ..., n1]. Internally, fresh n OK is used n times. However, OK is updated to ensure, that the elements of the resulting list not only satisfy OK, but are also distinct from each other.
val fresh_list : (t -> bool) -> t list -> t list
    fresh_list OK ns builds variants of the names in list ns such that all elements of the
resulting list ns' satisfy OK and are distinct to each other.

11.2 names with whitespace an type

type lskips_t
    lskips_t is the type of names with immediately preceding skips, i.e. whitespace or
comments
val lskip_pp : Format.formatter -> lskips_t -> unit
val from_x : Ast.x_l -> lskips_t
    creates a name from Ast.x_l, used during typechecking
val from_ix : Ast.ix_l -> lskips_t
    creates a name from Ast.ix_l, used during typechecking
val add_lskip : t -> lskips_t
    add_lskip converts a name into a name with skips by adding empty whitespace
val strip_lskip : lskips_t -> t
    strip_lskip converts a name with whitespace into a name by dropping the preceeding
whitespace
val get_lskip : lskips_t -> Ast.lex_skips
    get_lskip n gets the preceeding whitespace of n
val add_pre_lskip : Ast.lex_skips -> lskips_t -> lskips_t
    add_pre_lskip sk n adds additional whitespace in front of n
val replace_lskip : lskips_t -> Ast.lex_skips -> lskips_t
    replace_lskip sk n replaces the whitespace in front of n with sk. The old whitespace is
thrown away.
val lskip_rename : (Ulib.Text.t -> Ulib.Text.t) -> lskips_t -> lskips_t
    lskip_rename r_fun n is a version of rename that can handle lskips. It renames n using the
function r_fun and preserves the original whitespace.
11.3 output functions

val to_output_format : (Output.id_annot -> Ulib.Text.t -> Output.t) -> Output.id_annot -> lskips_t -> Output.t
  to_output_format format_fun id_annot n formats the name n as output. A name with output consists of preceding whitespace, the name as a text and a name-type. The space is formatted using ws, the other components together with id_annot are formatted with format_fun.

val to_output : Output.id_annot -> lskips_t -> Output.t
  to_output is the same as to_output_format Output.id

val to_output_quoted : string -> string -> Output.id_annot -> lskips_t -> Output.t
  to_output_quoted qs_begin qs_end id_annot n formats n with the quoting strings qs_begin and qs_end added before and after respectively.

val to_rope_tex : Output.id_annot -> t -> Ulib.Text.t
  to_rope_tex a n formats n as a for the tex-backend as a string. The preceding whitespace is ignored.

12 Module Nvar

type t
val compare : t -> t -> int
val pp : Format.formatter -> t -> unit
val nth : int -> t
val from_rope : Ulib.Text.t -> t
val to_rope : t -> Ulib.Text.t

13 Module Output : Intermediate output format before going to strings

type t

type t' =
  | Kwd' of string
  | Ident' of Ulib.Text.t
  | Num' of int

type id_annot =
  | Term_const of bool * bool
13.1 constructing output

val emp : t
  Empty output

val kwd : string -> t
  kwd s constructs the output for keyword s

val num : int -> t
  num i constructs the output for number i

val str : Ulib.Text.t -> t
  str s constructs the output for string constant s

val ws : Ast.lex_skips -> t
  Whitespace

val err : string -> t
  err message is an error output. An exception is thrown with the given message if this output is created. Used for marking problems.

val meta : string -> t
  meta s creates a string directly as output such that the formatting can’t interfere with string s any more

val comment : string -> t
  A comment
val comment_block : int option -> string list -> t
  comment a whole list of lines in a block.

val new_line : t
  a new line

val space : t
  a single space

val texspace : t
  ??? Unsure what it is. Some kind of tex specific space, similar to space, but treated slightly
differently by the Latex backend. It seems to be for example removed at beginnings and
ends of lines and multiple ones are collapsed into a single space.

val id : id_annot -> Ulib.Text.t -> t
  An identifier

val (^) : t -> t -> t
  o1 ^ o2 appends to outputs to each other

val flat : t list -> t
  flat [o0; ...; on] appends all the outputs in the list, i.e. it does o0 ^ ... ^ on.

val concat : t -> t list -> t
  concat sep [o0; ...; on] appends all the outputs in the list using the separator sep, i.e.
it does o0 ^ sep ^ o1 ^ ... ^ sep ^ tn.

val prefix_if_not_emp : t -> t -> t
  prefix_if_not_emp o1 o2 returns o1 ^ o2 if o2 is not empty and emp otherwise

13.2 Pretty Printing

13.2.1 Blocks

Blocks are used for pretty printing if the original whitespace should not be used. This is usually
the case, if the source was generated by some macro, such that either no original spacing is present
or it is likely to be broken. If the first argument of a block is true this block and all it’s content is
printed using OCaml’s Format library. The other arguments of blocks correspond to blocks in the
Format library. They describe indentation, the type of block and the content.

val block : bool -> int -> t -> t
val block_h : bool -> int -> t -> t
val block_v : bool -> int -> t -> t
val block_hv : bool -> int -> t -> t
val block_hov : bool -> int -> t -> t
val core : t -> t
core out is a marker for marking the most important part of some output. It marks for example the rhs of a definition. Together with extract_core this is used to sometimes only print the most essential part of some output.

val remove_core : t -> t
remove_core o removes all occurences of core form t by replacing core o' with just o'.

val extract_core : t -> t list
extract_core o extracts all top-level cores from output o.

13.2.2 Spacing

val remove_initial_ws : t -> t
removes intial whitespace (including comments) from output

val break_hint : bool -> int -> t
break_hint add_space ind is a general hint for a line-break. If add_space is set a space is added in case no line-break is needed. Otherwise a line-break with the given indentation ind is applied.

val break_hint_cut : t
break_hint_cut is short for break_hint false 0. It allows a newline at this position without indentation. If no newline is needed don't add any space.

val break_hint_space : int -> t
break_hint_space ind is short for break_hint true ind. It adds a space or a newline. If a newline is needed use the given indentation.

val ensure_newline : t
Make sure there is a newline starting here. This inserts a newline if necessary.

13.3 Output to Rope

val to_rope : Ulib.Text.t -> (Ast.lex_skip -> Ulib.Text.t) -> (t' -> t' -> bool) -> t -> Ulib.Text.t
to_rope quote_char lex_skips_to_rope need_space t formats the output t as an unicode text. The quote_char argument is used around strings. The function lex_skips_to_rope is used to format whitespace. Finally the function need_space is used to determine, whether an extra space is needed between simplified outputs.

val ml_comment_to_rope : Ast.ml_comment -> Ulib.Text.t
ml_comment_to_rope com formats an ML-comment as a text by putting (* and *) around it.
13.4 Latex Output

val to_rope_tex : t -> Ulib.Text.t  
  to_rope_tex t corresponds to to_rope for the Latex backend. Since it is used for only one 
  backend, the backend parameters of to_rope can be hard-coded.

val to_rope_option_tex : t -> Ulib.Text.t option  
  to_rope_option_tex t is similar to to_rope_tex t. However, it checks whether the result 
  is an empty text and returns None is in this case.

val tex_escape : Ulib.Text.t -> Ulib.Text.t  
val tex_escape_string : string -> string  
val tex_command_escape : Ulib.Text.t -> Ulib.Text.t  
val tex_command_label : Ulib.Text.t -> Ulib.Text.t  
val tex_command_name : Ulib.Text.t -> Ulib.Text.t

14 Module Path: internal canonical long identifiers

type t  
val compare : t -> t -> int  
val pp : Format.formatter -> t -> unit  
val from_id : Ident.t -> t  
val mk_path : Name.t list -> Name.t -> t  
val mk_path_list : Name.t list -> t  
  mk_path_list names splits names into ns @ [n] and calls mk_path ns n. It fails, if names 
  is empty.

val get_module_path : t -> t option  
  get_module_path p returns the module path of path p. If if is a path of an identifier m0. 
  ... . mn . f, then get_module returns the module path m0. ... . mn. If the path 
  does not have a module prefix, i.e. if it is a single name f, None is returned.

val natpath : t  
val listpath : t  
val vectorpath : t  
val boolpath : t  
val bitpath : t  
val setpath : t  
val stringpath : t  
val unitpath : t
val charpath : t
val numeralpath : t
val get_name : t -> Name.t
val get_toplevel_name : t -> Name.t

get_toplevel_name p gets the outmost name of a path. This is important when checking prefixes. For example, the result for path module.submodule.name is module and for name it is name.

val check_prefix : Name.t -> t -> bool
val to_ident : Ast.lex_skips -> t -> Ident.t
val to_name : t -> Name.t
val to_name_list : t -> Name.t list * Name.t
val to_string : t -> string

15 Module Pattern_syntax : general functions about patterns

general functions about patterns

15.1 Destructors and selector functions

val is_var_wild_pat : Typed_ast.pat -> bool
is_var_wild_pat p checks whether the pattern p is a wildcard or a variable pattern. Before checking type-annotations, parenthesis, etc. are removed.

val is_var_pat : Typed_ast.pat -> bool
is_var_pat p checks whether the pattern p is a variable pattern.

val is_ext_var_pat : Typed_ast.pat -> bool
is_ext_var_pat p checks whether the pattern p is a variable pattern in the broadest sense. In contrast to is_var_pat p also variables with type-annotations and parenthesis are accepted. is_var_wild_pat p additionally accepts wildcard patterns.

val is_var_tup_pat : Typed_ast.pat -> bool
is_var_tup_pat p checks whether the pattern p consists only of variable and tuple patterns.

val is_var_wild_tup_pat : Typed_ast.pat -> bool
is_var_wild_tup_pat p checks whether the pattern p consists only of variable, wildcard and tuple patterns.

val dest_var_pat : Typed_ast.pat -> Name.t option
dest_var_pat p destructs variable patterns and returs their name. If p is not a variable pattern, None is returned.
val dest_ext_var_pat : Typed_ast.pat -> Name.t option

dest_ext_var_pat p is an extended version of det_var_pat p. In addition to det_var_pat p it can handle variable patterns with type annotations and is able to strip parenthesis.

val pat_to_ext_name : Typed_ast.pat -> Typed_ast.name_lskips_annot option

pat_to_ext_name p is very similar to dest_ext_var_pat p. However, instead of returning just a name, pat_to_ext_name returns additionally the whitespace and the type in form of a name_lskips_annot.

val is_wild_pat : Typed_ast.pat -> bool

is_wild_pat p checks whether the pattern p is a wildcard pattern.

val dest_tup_pat : int option -> Typed_ast.pat -> Typed_ast.pat list option

dest_tup_pat lo p destructs a tuple pattern. If p is no tuple pattern, None is returned. Otherwise, it destructs the tuple pattern into a list of patterns pL. If lo is not None, it checks whether the length of this list matches the length given by lo. If this is the case Some pL is returned, otherwise None.

val mk_tup_pat : Typed_ast.pat list -> Typed_ast.pat

mk_tup_pat [p1, ..., pn] creates the pattern (p1, ..., pn).

val is_tup_pat : int option -> Typed_ast.pat -> bool

is_tup_pat lo p checks whether p is a tuple pattern of the given length. see dest_tup_pat

val dest_tf_pat : Typed_ast.pat -> bool option

dest_tf_pat p destructs boolean literal patterns, i.e. true and false patterns.

val is_tf_pat : Typed_ast.pat -> bool

if_tf_pat p checks whether p is the true or false pattern.

val is_t_pat : Typed_ast.pat -> bool

if_t_pat p checks whether p is the true pattern.

val is_f_pat : Typed_ast.pat -> bool

if_f_pat p checks whether p is the false pattern.

val mk_tf_pat : bool -> Typed_ast.pat

mk_tf_pat b creates true or false pattern.

val mk_paren_pat : Typed_ast.pat -> Typed_ast.pat

adds parenthesis around a pattern

val mk_opt_paren_pat : Typed_ast.pat -> Typed_ast.pat

adds parenthesis around a pattern, when needed
val dest_num_pat : Typed_ast.pat -> int option
  dest_num_pat p destructs number literal patterns

val is_num_pat : Typed_ast.pat -> bool
  is_num_pat p checks whether p is a number pattern.

val mk_num_pat : Types.t -> int -> Typed_ast.pat
  mk_num_pat num_ty i makes a number pattern.

val dest_num_add_pat : Typed_ast.pat -> (Name.t * int) option
  dest_num_add_pat p destructs number addition literal patterns

val mk_num_add_pat : Types.t -> Name.t -> int -> Typed_ast.pat
  mk_num_add_pat num_ty i makes a number addition pattern.

val is_num_add_pat : Typed_ast.pat -> bool
  is_num_add_pat p checks whether p is a number addition pattern.

val num_ty_pat_cases :
  (Name.t -> 'a) ->
  (int -> 'a) ->
  (Name.t -> int -> 'a) -> 'a -> (Typed_ast.pat -> 'a) -> Typed_ast.pat -> 'a
  num_ty_pat_cases f_v f_i f_a f_w f_else p performs case analysis for patterns of type num. Depending of which form the pattern p has, different argument functions are called:
  • v → f_v v
  • c (num constant) → f_i i
  • v + 0 → f_v v
  • v + i (for i > 0) → f_a v i
  • _ → f_w
  • p (everything else) → f_else p

val dest_string_pat : Typed_ast.pat -> string option
  dest_string_pat p destructs number literal patterns

val is_string_pat : Typed_ast.pat -> bool
  is_string_pat p checks whether p is a number pattern.

val dest_cons_pat : Typed_ast.pat -> (Typed_ast.pat * Typed_ast.pat) option
  dest_cons_pat p destructs list-cons patterns.

val is_cons_pat : Typed_ast.pat -> bool
val dest_list_pat : int option -> Typed_ast.pat -> Typed_ast.pat list option
dest_list_pat p destructs list patterns.

val is_list_pat : int option -> Typed_ast.pat -> bool
val dest_const_pat :
Typed_ast.pat ->
(Typed_ast.const_descr_ref Types.id * Typed_ast.pat list) option
dest_constr_pat p destructs constructor patterns.

val is_const_pat : Typed_ast.pat -> bool
val dest_record_pat :
Typed_ast.pat ->
(Typed_ast.const_descr_ref Types.id * Typed_ast.pat) list option
dest_record_pat p destructs record patterns.

val is_record_pat : Typed_ast.pat -> bool

15.2 Classification of Patterns

val is_constructor :
Ast.l -> Typed_ast.env -> Target.target -> Typed_ast.const_descr_ref -> bool
is_constructor l env targ c checks whether c is a constructor for target targ in
environment env. If you want to know whether it is for any target, use the identity target.
Internally, it checks whether type_defs_get_constr_families returns a non-empty list.

val is_buildin_constructor :
Ast.l -> Typed_ast.env -> Target.target -> Typed_ast.const_descr_ref -> bool
is_buildin_constructor l env targ c checks whether c is a build-in constructor for
target targ in environment env. Build-in constructors are constructors, which the target
pattern compilation can handle.

val is_not_buildin_constructor :
Ast.l -> Typed_ast.env -> Target.target -> Typed_ast.const_descr_ref -> bool
is_not_buildin_constructor l env targ c checks whether c is a constructor for target
targ in environment env, but not a build-in one. Not build-in constructors get compiled
away during pattern compilation.

val direct_subpats : Typed_ast.pat -> Typed_ast.pat list
direct_subpats p returns a list of all the direct subpatterns of p.

val subpats : Typed_ast.pat -> Typed_ast.pat list
subpats p returns a list of all the subpatterns of p. In contrast to direct_subpats p really
all subpatterns are returned, not only direct ones. This means that the result of
direct_subpats p is a subset of subpats p.

val exists_subpat : (Typed_ast.pat -> bool) -> Typed_ast.pat -> bool
exists\_pat \ cf \ p \text{ checks whether } p \text{ has a subpattern } p' \text{ such that } cf \ p' \text{ holds.}

val \text{ for\_all\_subpat} : (\text{Typed\_ast\_pat} -> \text{bool}) -> \text{Typed\_ast\_pat} -> \text{bool}
\text{for\_all\_subpat} \ cf \ p \text{ checks whether all subpatterns } p' \text{ of } p \text{ satisfy } cf \ p'.

val \text{ single\_pat\_exhaustive} : \text{Typed\_ast\_pat} -> \text{bool}
\text{single\_pat\_exhaustive} \ p \text{ checks whether the pattern } p \text{ is exhaustive.}

val \text{ pat\_vars\_src} : \text{Typed\_ast\_pat} -> (\text{Name.lskips\_t, unit}) \text{ Types.annot list}
\text{pat\_vars\_src} \ p \text{ returns a list of all the variable names occurring in the pattern. The names are annotated with the type and the whitespace information.}

15.3 miscellaneous

val \text{ pat\_extract\_lskips} : \text{Typed\_ast\_pat} -> \text{Ast.lex\_skips}
\text{pat\_extract\_lskips} \ p \text{ extracts all whitespace from a pattern}

val \text{ split\_var\_annot\_pat} : \text{Typed\_ast\_pat} -> \text{Typed\_ast\_pat}
\text{split\_var\_annot\_pat} \ p \text{ splits annotated variable patterns in variable patterns + type annotation. All other patterns are returned unchanged.}

exception \text{ Pat\_to\_exp\_unsupported} of \text{Ast.l * string}
val \text{ pat\_to\_exp} : \text{Typed\_ast\_env} -> \text{Typed\_ast\_pat} -> \text{Typed\_ast\_exp}
\text{pat\_to\_exp} \ \text{env} \ p \text{ tries to convert } p \text{ into a corresponding expression. This might fail, e.g. if } p \text{ contains wildcard patterns. If it fails a } \text{pat\_to\_exp\_unsupported} \text{ exception is raised.}

16 Module Patterns: pattern compilation

pattern compilation

16.1 Pattern Compilation

type \text{match\_props} = {
  is\_exhaustive : \text{bool} ;
  missing\_pats : \text{Typed\_ast\_pat list} \text{ list} ;
  redundant\_pats : (\text{int} * \text{Typed\_ast\_pat}) \text{ list} ;
  overlapping\_pats : ((\text{int} * \text{Typed\_ast\_pat}) * (\text{int} * \text{Typed\_ast\_pat})) \text{ list} ;
} ;

val \text{ check\_match\_exp} : \text{Typed\_ast\_env} -> \text{Typed\_ast\_exp} -> \text{match\_props} \text{ option}
\text{check\_match\_exp} \ \text{env} \ e \text{ checks the pattern match expression } e \text{ in environment } \text{env}. \text{ If } e \text{ is not a pattern match, } \text{None} \text{ is returned. Otherwise, a record of type } \text{match\_props} \text{ is returned that contains information on whether the match is exhaustive, contains redundant parts etc.}

28
val check_pat_list : 
Typed_ast.env -> Typed_ast.pat list -> match_props option

check_pat_list env pl checks the pattern list pl in environment env. If pl is empty or the compilation fails, None is returned. Otherwise, a record of type match_props is returned that contains information on whether the match is exhaustive, contains redundant parts etc.

val check_match_exp_warn : Typed_ast.env -> Typed_ast.exp -> unit

check_match_exp_warn env e internally calls check_match_exp env e. Instead of returning the properties of the match expression, it prints appropriate warning messages, though.

val check_match_def : 
Typed_ast.env -> Typed_ast.def -> (Name.t * match_props) list

check_match_def env d checks a definition using pattern matching d in environment env. Definitions of mutually recursive functions can contain multiple top-level pattern matches. Therefore, a list is returned. This lists consists of pairs of the name of the defined function and it’s properties. If the definition does not have a top-level pattern match, i.e. if it is not a function definition, the empty list is returned.

val check_match_def_warn : Typed_ast.env -> Typed_ast.def -> unit

check_match_def_warn env d checks a definition and prints appropriate warning messages.

type match_check_arg

val cleanup_match_exp :
Typed_ast.env -> bool -> Typed_ast.exp -> Typed_ast.exp option

cleanup_match_exp env add_missing e tries to cleanup the match-expression e by removing redundant rows. Moreover, missing patterns are added at the end, if the argument add_missing is set.

val compile_match_exp :
Target.target ->
match_check_arg ->
Typed_ast.env -> Typed_ast.exp -> Typed_ast.exp option

compile_match_exp target_opt pat_OK env e compiles match-expressions. In contrast to check_match_exp only case-expressions are checked. Other types of pattern matches have to be brought into this form first.

If the case-expression e contains a pattern p such that pat_OK p does not hold, the whole case-expression is processed and transformed into an expression with the same semantics that contains only supported patterns. During this compilation, warning messages might be issued. This warning uses target_opt. Otherwise, it is not used.

val compile_exp :
Target.target ->
match_check_arg ->
Typed_ast.env ->

29
Macro_expander.macro_context -> Typed_ast.exp -> Typed_ast.exp option

val compile_def : Target.target -> match_check_arg -> Typed_ast.env -> Def_trans.def_macro

val is_isabelle_pattern_match : match_check_arg
val is_hol_pattern_match : match_check_arg
val is_coq_pattern_match : match_check_arg
val is_ocaml_pattern_match : match_check_arg
val is_pattern_match_const : bool -> match_check_arg

16.2 Other pattern functions

val check_number_patterns : Typed_ast.env -> Typed_ast.pat -> unit
    checked_number_patterns env p checks that all number patterns which are part of p are of type nat or natural.

val remove_function :
    Typed_ast.env ->
    (Typed_ast.exp -> Typed_ast.exp) -> Typed_ast.exp -> Typed_ast.exp option
    remove_function env case_f e replaces the function expression e with with fun x ->
    match x with .... The function case_f is then applied to the new match-expression.

val remove_fun :
    Typed_ast.env ->
    (Typed_ast.exp -> Typed_ast.exp) -> Typed_ast.exp -> Typed_ast.exp option
    remove_fun env case_f e replaces the fun-expression e. If e is of the form
    fun p0 ... pn -> e' such that not all patterns pi are variable patterns, it is replaced with
    fun x0 ... xn -> match (x0, ..., xn) with (p0, ..., pn) -> e'. The function case_f is then
    applied to the new match-expression.

val remove_toplevel_match :
    Target.target ->
    match_check_arg -> Typed_ast.env -> Def_trans.def_macro
    remove_toplevel_match tries to introduce matching directly in the function definition by
    eliminating match-expressions in the body.

val collapse_nested_matches :
    match_check_arg ->
    Typed_ast.env -> Typed_ast.exp -> Typed_ast.exp option
    collapse_nested_matches tries to eliminate nested matches by collapsing them. It is used
    internally by pattern compilation.
type 'a parser
val return : 'a -> 'a parser
val (>>=) : 'a parser ->
  ('a -> 'b parser) -> 'b parser
val fail : 'a parser
val eof : unit parser
val predicate : (char -> bool) -> char parser
val (+++) : 'a parser -> 'a parser -> 'a parser
val many : 'a parser -> 'a list parser
val many1 : 'a parser -> 'a list parser
val repeat : int -> 'a parser -> 'a list parser
val sep_by : 'a parser ->
  'b parser -> 'a list parser
val sep_by1 : 'a parser ->
  'b parser -> 'a list parser
val one_of : char list -> char parser
val char_exact : char -> char parser
val string_exact : string -> string parser
val int_exact : int -> string parser
val bool_exact : bool -> string parser
val digit : int parser
val digits : int parser
val whitespace : string parser
val whitespace1 : string parser

val parse : string -> 'a parser ->
  'a parse_result
val parse_and_print : string -> 'a parser ->
  ('a -> string) -> unit

val pp_str : Format.formatter -> string -> unit
val lst :
  ('a, Format.formatter, unit) Pervasives.format ->
val opt : 
(Format.formatter -> 'a -> unit) -> Format.formatter -> 'a option -> unit
val pp_to_string : (Format.formatter -> 'a) -> string

19  Module Precedence: a prefix operation

type t = 
| P_prefix 
  a prefix operation

| P_infix of int
  a non-associative infix operation of the given precedence, higher precedence bind stronger

| P_infix_left of int
  a left-associative infix operation

| P_infix_right of int
  a right-associative infix operation

| P_special
  an operation with special syntax (e.g. if-then-else)

type context =
| Field
| App_right
| App_left
| Infix_left of t
| Infix_right of t
| Delimited

type exp_kind =
| App
| Infix of t
| Let
| Atomic

type pat_context =
| Plist
| Pas_left
| Pcons_left
| Pcons_right
| Pdelimited

type pat_kind =
| Papp
| Pas
val is_infix : t -> bool
val needs_parens : context -> exp_kind -> bool
val pat_needs_parens : pat_context -> pat_kind -> bool

val get_prec :
  Target.target -> Typed_ast.env -> Typed_ast.const_descr_ref -> t
get_prec target env c looks up the precedence of constant c in environment env for the
target target. Thereby, it follows target-representations of this constant.

val get_prec_exp : Target.target -> Typed_ast.env -> Typed_ast.exp -> t
get_prec target env e looks up the precedence of expression e in environment env for the
target target. If the expression is essentially a constant (i.e. a constant with perhaps
parenthesis or types added), the precedence of this constant is returned using get_prec.
Otherwise P_prefix is returned.

20 Module Process_file : The full environment built after all type-
checking, and transforming

val parse_file : string -> Ast.defs * Ast.lex_skips

val output :
  Typed_ast.env ->
  Typed_ast.var_avoid_f ->
  Target.target -> string option -> Typed_ast.checked_module list -> unit

val output_alltexdoc :
 .Typed_ast.env ->
  TypedAst.var_avoid_f ->
  string -> string -> Typed_ast.checked_module list -> unit
output_alltexdoc produces the latex output for all modules in a single file

val always_replace_files : bool Pervasives.ref
always_replace_files determines whether Lem only updates modified files. If it is set to
true, all output files are written, regardless of whether the files existed before. If it is set to
false and an output file already exists, the output file is only updated, if its content really
changes. For some backends like OCaml, HOL, Isabelle, this is beneficial, since it prevents
them from reprocessing these unchanged files.

val only_auxiliary : bool Pervasives.ref
only_auxiliary determines whether Lem generates only auxiliary files

val output_sig : Format.formatter -> Typed_ast.env -> unit
Module Rename_top_level: renaming and module flattening for some targets

val flatten_modules : Path.t -> Typed_ast.env -> Typed_ast.def list -> Typed_ast.def list

val rename_defs_target : Target.target -> Typed_ast_syntax.used_entities -> Typed_ast.NameSet.t -> Typed_ast.env -> Typed_ast.env

rename_target topt ue consts e processes the entities (constants, constructors, types, modules...) stored in ue and renames them for target topt. This renaming is target specific. It avoids the names in set consts and modifies the descriptions of constants, types, etc. in environment e. The modified environment is returned.

Module Reporting: reporting errors and warnings

22.1 Warnings

type warn_source =
| Warn_source_exp of Typed_ast.exp
| Warn_source_def of Typed_ast.def
| Warn_source_unknown

Warnings can be caused by definitions or expressions. The type warn_source allows to pass the origin easily to warnings.

val warn_source_to_locn : warn_source -> Ast.l

type warning =
| Warn_general of bool * Ast.l * string
| Warn_rename of Ast.l * string * (string * Ast.l) option * string * Target.target
| Warn_pattern_compilation_failed of Ast.l * Typed_ast.pat list * warn_source
| Warn_pattern_not_exhaustive of Ast.l * Typed_ast.pat list list
| Warn_def_not_exhaustive of Ast.l * string * Typed_ast.pat list list

Warning about renaming an identifier. The arguments are the old name, an optional intermediate one, the new name and the target.

pattern compilation failed

pattern match is not exhaustive

a function is defined using non-exhaustive pattern-matching
val warnings_active : bool Pervasives.ref
  if the flag warnings_active is set, warning messages are printed, otherwise they are thrown away.

val report_warning : TypedList_ast.env -> warning -> unit
  report_warning env w reports a warning. Depending on the settings for the warning type this might mean, do nothing, print a warning message or print an error message and exit Lem.

val report_warning_no_env : warning -> unit

Warnings are problems that Lem can deal with. Depending on user settings, they can be completely ignored, reported to the user or even be treated as an error.
report_warning_no_env w reports a warning, when no-environment is available. In contrast to report_warning the warning messages might be more basic, since no information can be extracted from the environment.

22.2 Auxiliary Functions

val warn_opts : (string * Arg.spec * string) list
  Command line options for warnings
val ignore_pat_compile_warnings : unit -> unit
  Turn off pattern compilation warnings, used by main

22.3 Debuging

val print_debug_exp : Typed_ast.env -> string -> Typed_ast.exp list -> unit
val print_debug_def : Typed_ast.env -> string -> Typed_ast.def list -> unit
val print_debug_pat : Typed_ast.env -> string -> Typed_ast.pat list -> unit
val print_debug_typ : Typed_ast.env -> string -> Types.t list -> unit
val print_debug_src_t : Typed_ast.env -> string -> Types.src_t list -> unit

23 Module Reporting_basic : Basic error reporting

Reporting_basic contains functions to report errors and warnings. It contains functions to print locations (Ast.l) and lexing positions. Despite Ast it should not depend on any other Lem-file. This guarantees that it can be used throughout the whole development.

The main functionality is reporting errors. This is done by raising a Fatal_error exception. This is caught inside Lem and reported via report_error. There are several predefined types of errors which all cause different error messages. If none of these fit, Err_general can be used.

Reporting functions that need access to parts of the Lem development like Typed_ast are collected in Reporting.

23.1 Auxiliary Functions

val loc_to_string : bool -> Ast.l -> string
  loc_to_string short l formats l as a string. If short is set, only the most originating location is formatted, not what methods transformed l.
val print_err : bool -> bool -> bool -> Ast.l -> string -> string -> unit
  print_err fatal print_loc_source print_only_first_loc l head mes prints an error / warning message to std-err. It starts with printing location information stored in l. If print_loc_source is set, the original input described by l is retrieved and shown. It then prints "head: mes". If fatal is set, the program exists with error-code 1 afterwards.
23.2 Debugging

val debug_flag : bool Pervasives.ref
  Should debug be printed

val print_debug : string -> unit
  print_debug s prints the string s with some debug prefix to the standard error output.

23.3 Errors

type error =
  | Err_general of bool * Ast.l * string
    General errors, used for multi purpose. If you are unsure, use this one.
  | Err_unreachable of Ast.l * string
    Unreachable errors should never be thrown. It means that some code was executed
    that the programmer thought of as unreachable
  | Err_todo of bool * Ast.l * string
    Err_todo indicates that some feature is unimplemented. Normally, it should be build
    using err_todo in order simplify searching for occorences in the source code.
  | Err_trans of Ast.l * string
  | Err_trans_header of Ast.l * string
  | Err_syntax of Lexing.position * string
  | Err_syntax_locn of Ast.l * string
  | Err_lex of Lexing.position * char
  | Err_type of Ast.l * string
    A typechecking error
  | Err_internal of Ast.l * string
  | Err_rename of Ast.l * string
  | Err_cyclic_build of string
    resolving module dependencies detected a cyclic dependency of the given module
  | Err_cyclic_inline of Ast.l * string * string
    Err_cyclic_inline l target const means that the inline of some constant const is
    cyclic for target target
  | Err_resolve_dependency of Ast.l * string list * string
    could not find a Module that should be imported in given list of directories
  | Err_reorder_dependency of Ast.l * string
    Err_reorder_dependency (l, m) module m is needed at location l, but not allowed
    to be imported, because this would require reording the user input
  | Err_fancy_pattern_constant of Ast.l * string
a constant occurring in a pattern has a fancy target-representation, that cannot be
dealt with for patterns

In contrast to warnings, errors always kill the current run of Lem. They can’t be recovered
from. **Err_todo** should not be used directly, but only through **err_todo** in order to make
search easier.

Errors usually have location information and a message attached. Some also carry a boolean
flag indicating, the original source corresponding to the location information should be
looked up and printed.

```ml
exception Fatal_error of error
```

Since errors are always fatal, they are reported by raising an **Fatal_error** exception instead
of calling a report-function.

```ml
val err_todo : bool -> Ast.l -> string -> exn
  err_todo b l m is an abbreviation for Fatal_error (Err_todo (b, l, m))
```

```ml
val err_general : bool -> Ast.l -> string -> exn
  err_general b l m is an abbreviation for Fatal_error (Err_general (b, l, m))
```

```ml
val err_unreachable : Ast.l -> string -> exn
  err_unreachable l m is an abbreviation for Fatal_error (Err_unreachable (l, m))
```

```ml
val err_type : Ast.l -> string -> exn
  err_type l msg is an abbreviation for Fatal_error (Err_type (l, m), i.e. for a general
type-checking error at location l with error message msg.
```

```ml
val err_type_pp : Ast.l -> string -> (Format.formatter -> 'a -> unit) -> 'a -> exn
  err_type l msg pp n is similar to err_type. However it uses the formatter pp to format n,
resulting in a string label. The error message then has the form label : msg.
```

```ml
val report_error : error -> 'a
  Report error should only be used by main to print the error in the end. Everywhere else,
raising a Fatal_error exception is recommended.
```

24 Module Seplist: general thing of lists with optional separators

```ml
type ('a, 's) t
val empty : ('a, 's) t
val cons_sep : 's -> ('a, 's) t -> ('a, 's) t
val cons_sep_alt : 's -> ('a, 's) t -> ('a, 's) t
```
cons_sep_alt doesn’t add the separator if the list is empty

val cons_entry : 'a -> ('a, 's) t -> ('a, 's) t
val is_empty : ('a, 's) t -> bool
val sing : 'a -> ('a, 's) t
    sing a constructs a seplist with entry a. It does the same as cons_entry a empty.
val hd : ('a, 's) t -> 'a
    gets the first entry, if there is one
val hd_sep : ('a, 's) t -> 's
    gets the first separator, if there is one
val tl : ('a, 's) t -> ('a, 's) t
    Removes the first entry, fails is there is none, or if a separator is first
val tl_alt : ('a, 's) t -> ('a, 's) t
    Removes the first entry, fails is there is none, removes any separator that precedes the first entry
val tl_sep : ('a, 's) t -> ('a, 's) t
    Removes the first separator, fails is there is none, or if an entry is first
val append : 's -> ('a, 's) t -> ('a, 's) t -> ('a, 's) t
    append d sl1 sl2 appends the seplists sl1 and sl2. If sl1 ends with a value and sl2 starts with a value, a default separator s is added. If sl1 ends with a separator and sl2 starts with a separator, the separator of sl2 is dropped.
val flatten : 's -> ('a, 's) t list -> ('a, 's) t
    flatten d sll flattens a list of seplists by applying append repeatedly
val to_list : ('a, 's) t -> 'a list
    Makes a normal list, ignoring separators
val to_pair_list : 's -> ('a, 's) t -> 's option * ('a * 's) list
    Makes a normal list of pairs. The first separator is returned separately, an default one added for the last entry, if the lists ends with a value
val from_pair_list : 's option -> ('a * 's) list -> 'a option -> ('a, 's) t
    constructs a seplist from a list of pairs. In contrast to from_list, the last separator is kept
val from_pair_list_sym :
    'a option -> ('s * 'a) list -> 's option -> ('a, 's) t
from_pair_list_sym first_val_opt sep_val_list last_sep_opt constructs a seplist from a list of pairs sep_val_list. In contrast to from_pair_list, the separator is the first component of these pairs. This also means that we now need an optional first value before the list and an optional last separator after the list, whereas from_pair_list has an optional first separator and last value.

val drop_first_sep : ('a, 's) t -> 's option * ('a, 's) t

If s1 starts with a separator, it is dropped and returned, otherwise nothing happens.

val to_sep_list : ('a -> 'b) -> ('s -> 'b) -> ('a, 's) t -> 'b list

Flattens into a normal list with separators and elements intermixed

type ('s, 'a) optsep =
| Optional
| Require of 's
| Forbid of ('s -> 'a)

val to_sep_list_first : ('s, 'b) optsep ->
('a -> 'b) -> ('s -> 'b) -> ('a, 's) t -> 'b list

Flattens into a normal list with separators and elements intermixed, with special control over the first separator. Optional indicates no special treatment (works as to_sep_list), Require adds the given initial separator if there is none, and Forbid removes the initial separator if there is one. In the latter case, the initial separator is processed by the function argument to Forbid

val to_sep_list_last : ('s, 'b) optsep ->
('a -> 'b) -> ('s -> 'b) -> ('a, 's) t -> 'b list

As to_sep_list_first, but for the last separator

val to_list_map : ('a -> 'b) -> ('a, 's) t -> 'b list

val iter : ('a -> unit) -> ('a, 's) t -> unit

val from_list : ('a * 's) list -> ('a, 's) t

The from list functions ignore the last separator in the input list

val from_list_prefix : 's -> bool -> ('a * 's) list -> ('a, 's) t
val from_list_suffix : ('a * 's) list -> 's -> bool -> ('a, 's) t
val from_list_default : 's -> 'a list -> ('a, 's) t

from_list_default d l constructs a seplist form a list of entries l using the separator d as default separator between all entries.

val length : ('a, 's) t -> int

val map : ('a -> 'b) -> ('a, 's) t -> ('b, 's) t
val map_changed : ('a -> 'a option) -> ('a, 's) t -> ('a, 's) t option
Returns None if the function returns None on all of the elements, otherwise returns a list that uses the original element where the function returns None.

```ml
val map_acc_right :
  ('a -> 'c -> 'b * 'c) -> 'c -> ('a, 's) t -> ('b, 's) t * 'c
Maps with an accumulating parameter. The _right version builds the accumulator
right-to-left, and the _left version builds it left-to-right.

val map_acc_left :
  ('a -> 'c -> 'b * 'c) -> 'c -> ('a, 's) t -> ('b, 's) t * 'c
val fold_right : ('a -> 'c -> 'c) -> 'c -> ('a, 's) t -> 'c
  fold right implemented via map_acc_right

val fold_left : ('a -> 'c -> 'c) -> 'c -> ('a, 's) t -> 'c
  fold left implemented via map_acc_left

val for_all : ('a -> bool) -> ('a, 's) t -> bool

val exists : ('a -> bool) -> ('a, 's) t -> bool
val find : 's -> ('a -> bool) -> ('a, 's) t -> 'a * 's

val pp :
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  Format.formatter -> ('a, 'b) t -> unit
val replace_all_seps : ('s -> 's) -> ('a, 's) t -> ('a, 's) t
```

### 25 Module Syntactic_tests

```ml
val check_positivity_condition_def : Typed_ast.def -> unit
val check_decidable_equality_def : Typed_ast.env -> Typed_ast.def -> unit
val check_id_restrict_e : Typed_ast.env -> Typed_ast.exp -> unit
val check_id_restrict_p : Typed_ast.env -> Typed_ast.pat -> unit
```

### 26 Module Target : Datatype and Function for Targets

```ml
type non_ident_target =
  | Target_hol
  | Target_ocaml
  | Target_isa
  | Target_coq
  | Target_tex
  | Target_html
  | Target_lem
```

41
A datatype for Targets. In contrast to the one in `ast.ml` this one does not carry white-space information.

type target =
  | Target_no_ident of non_ident_target
  | Target_ident

  target for the typechecked ast is either a real target as in the AST or the identity target

val ast_target_to_target : Ast.target -> non_ident_target

  ast_target_to_target t converts an ast-target to a target. This essentially means dropping the white-space information.

val target_to_ast_target : non_ident_target -> Ast.target

  target_to_ast_target t converts a target t to an ast_target. This essentially means adding empty white-space information.

val ast_target_compare : Ast.target -> Ast.target -> int

  ast_target_compare is a comparison function for ast-targets.

val target_compare : non_ident_target -> non_ident_target -> int

  target_compare is a comparison function for targets.

module Targetmap :

  sig

    include Finite_map.Fmap

    val apply_target : 'a t -> Target.target -> 'a option

      apply_target m targ looks up the targ in map m. Target-maps only store information for real targets, not the identity one. If therefore targ is Target_ident, i.e. represents the identity backend, None is returned.

    val insert_target : 'a t -> Target.target * 'a -> 'a t

      insert_target m (targ, v) inserts value v for targ in map m. Target-maps only store information for real targets, not the identity one. If therefore targ is Target_ident, i.e. represents the identity backend, the map is not(!) updated.

  end

  target keyed finite maps

module Targetset :

  Set.S with type elt = non_ident_target

  target sets

val all_targets_list : non_ident_target list
A list of all the targets.

val all_targets : Targetset.t
The set of all the targets.

val all_targets_non_explicit : Targetset.t
The set of targets used when negating or no mentioning explicit targets. Targets like Lem
are excluded by default.

val all_targets_only_exec : Targetset.t
The set of targets that can handle only executable definitions. Currently only Ocaml, but
this might change.

val all_targets_only_exec_list : non_ident_target list

val non_ident_target_to_string : non_ident_target -> string
    non_ident_target_to_string t returns a string description of a target t.

val target_to_string : target -> string
target_to_string t_opt returns a string description of a target. If some target is given, it
does the same as target_to_string. Otherwise, it returns a string description of the
identity backend.

val non_ident_target_to_mname : non_ident_target -> Name.t
    non_ident_target_to_mname t returns a name for a target. It is similar to
    non_ident_target_to_string t. However, it returns capitalised versions.

val target_to_output : Ast.target -> Output.t
target_to_output t returns output for a target t.

val is_human_target : target -> bool
    is_human_target targ checks whether targ is a target intended to be read by humans and
therefore needs preserving the original structure very closely. Examples for such targets are
the tex-, html- and identity-targets.

val dest_human_target : target -> non_ident_target option
    dest_human_target targ destructs targ to get the non-identity target. If it s a
human-target, None is returned, otherwise the non-identity target.

27 Module Target_binding : resolve_module_path l env sk m tries
to find the module-path m in environment env.

It returns a shortest suffix m' of m that resolves to the same module in env, and adds the lskips sk
to the returned ident.

val resolve_module_path :
    Ast.l -> Typed_ast.env -> Types.ident_option -> Path.t -> Ident.t
resolve_module_path l env sk m tries to find the module-path m in environment env. It returns a shortest suffix m' of m that resolves to the same module in env, and adds the lskips sk to the returned ident.

val resolve_type_path : Ast.l -> Typed_ast.env -> Types.ident_option -> Path.t -> Ident.t
    resolve_type_path l env sk p tries to find the type of (absolute) path p in environment env. It returns a shortest suffix p' of p that resolves to the same type in env, and adds the lskips sk to the returned ident.

val resolve_const_ref : Ast.l -> Typed_ast.env -> Target.target -> Types.ident_option -> Typed_ast.const_descr_ref -> Ident.t
    resolve_const_ref l env target io c_ref tries to find the constant c_ref in environment env. Let p be the absolute path for c_ref. If tries io as default, if given. If that fails, it returns a shortest suffix p' of p that resolves to the same constant in env, and adds the lskips from io to the returned ident.

28 Module Target_syntax

val fix_infix_and_parens : Typed_ast.env -> Target.target -> Typed_ast.def list -> Typed_ast.def list

29 Module Target_trans: get_transformation targ returns the (pre-backend) transformation function for target targ

val get_transformation : Target.target -> Typed_ast.env -> Typed_ast.checked_module -> Typed_ast.checked_module
    get_transformation targ returns the (pre-backend) transformation function for target targ

val get_avoid_f : Target.target -> Typed_ast.NameSet.t -> Typed_ast.var_avoid_f
    get_avoid_f targ returns the target specific variable avoid function. Before this function can be used, it needs to get the set of constants to avoid.

val add_used_entities_to_avoid_names : Typed_ast.env ->

44
Target.target ->
Typed_ast_syntax.used_entities -> Typed_ast.NameSet.t -> Typed_ast.NameSet.t
add_used_entities_to_avoid_names env targ ue ns adds the used entities in ue to the
name-set ns. This nameset is intended to contain the names to avoid when using
get_avoid_f or rename_def_params. Since for each target different names need to be
avoided, the intended target is required as well. Finally, the environment is needed to
look-up target representations.

val rename_def_params :
Target.target ->
Typed_ast.NameSet.t ->
Typed_ast.checked_module list -> Typed_ast.checked_module list
Rename the arguments to definitions, if they clash with constants in a given set of constants.
This was previously part of the transformation returned by get_transformation. It got
moved out in order to see all the renamings of definitions before changing their arguments.

val ident_force_pattern_compile : bool Pervasives.ref
This flag enables pattern compilation for the identity backend. Used for debugging.

val ident_force_dictionary_passing : bool Pervasives.ref
This flag enables dictionary passing transformations for the identity backend. Used for
debugging.

val hol_remove_matches : bool Pervasives.ref
This flag enables removing top-level matches from definitions for the HOL4 backend.

30 Module Trans : macros for target_trans

exception Trans_error of Ast.l * string
type 'a macro = Macro_expander.macro_context -> 'a -> 'a option
type pat_macro = Macro_expander.pat_position -> Typed_ast.pat macro
module Macros :
   functor (E : sig
      val env : Typed_ast.env
   end ) -> sig

30.1 Record Macros

val remove_singleton_record_updates : Typed_ast.exp Trans.macro
   remove_singleton_record_updates replaces updates of records that have only one
   field with the construction of a completely new record.
remove_multiple_record_updates replaces record updates simultaneously updating multiple fields with a nested record update, each affecting only one field, that achieves the same effect.

sort_record_fields sorts the fields of a record expression into the same order as in the definition of the record type. If they do not need resorting, everything is fine, otherwise a warning is produced.

30.2 Set and List Comprehension Macros

remove_list_comprehension removes list comprehensions by turning them into fold and insert operations. A Trans_error exception is thrown, if not only bounded quantification is used.

remove_set_comprehension removes set comprehensions by turning them into fold and insert operations. A Trans_error exception is thrown, if not only bounded quantification is used.

remove_set_comprehension_image_filter removes set comprehensions by turning them into set-image, set-filter and set-product operations. For example \( \{ f (x,y,z) \mid (x,y) \in A \land (z \in B) \land P (x, y, z) \} \) is turned into \( \text{Set.image} f (\text{Set.filter} P (\text{Set.cross} A B)) \). If allow_sigma is set and the quantifiers depend on each other, set_sigma is used instead. So, for example \( \{ f (x,y,z) \mid (x,y) \in A \land (z \in B x) \land P (x, y, z) \} \) is turned into \( \text{Set.image} f (\text{Set.filter} P (\text{Set.set_sigma} A (\text{fun} (x, y) \to B x))) \).

In contrast to remove_set_comprehension no exception is thrown, if the translation fails. This is because it is intended to be used with theorem prover backends, which can handle unbounded quantification differently.

remove_setcomp removes set comprehensions with implicit bound variable to ones with explicitly bound ones. For example \( \{ (x, y) \mid x > y \} \) might, depending on context be turned in \( \{ (x, y) \mid \forall x \mid x > y \} \), \( \{ (x, y) \mid \forall x \ y \mid x > y \} \) or something similar.

cleanup_set_quant
cleanup_set_quant moves restricted and unrestricted quantification in set comprehensions to the condition part, if the bound variables are only used by the condition. This means, that expressions of the form \{ f x | forall (p IN e) ... | P x \} become \{ f x | forall ... | exists (p IN e). P x \} if x is not a member of FV p.

val remove_set_comp_binding : Typed_ast.exp Trans.macro

does not hold. pat_OK is used to configure, which types of restricted quantification are supported by the backend. For example, HOL 4 supports patterns consisting of variables, tuples and wildcard patterns, while Isabelle does not like wildcard ones. This macros tries to turn pattern p into an expression. This is likely to fail for more complex patterns. In these cases, remove_restr_quant fails and pattern compilation is needed.

val remove_restr_quant : (Typed_ast.pat -> bool) -> Typed_ast.exp Trans.macro

remove_restr_quant turns restricted quantification into unrestricted quantification, if then pattern does not satisfy pat_OK. For example, expressions of the from forall \(p \text{ IN } e\). P x becomes forall FV(p). \(p \text{ IN } e \rightarrow P x\). This requires turning pattern p into an expression. This is likely to fail for more complex patterns. In these cases, remove_restr_quant fails and pattern compilation is needed.

val remove_quant : Typed_ast.exp Trans.macro

remove_quant turns quantifiers into iteration. It throws an Trans_error exception, if used on unrestricted quantification. Given an expression forall \(x \text{ IN } X\). P x this returns Set.forall X (fun x -> P x). It also works for existential quantification and quantification over lists.

47
val remove_quant_coq : Typed_ast.exp Trans.macro

remove_quant_coq the same as above but does not apply in the body of lemma or theorem statements. Specific to the Coq backend.

30.4 Pattern Macros

val remove_unit_pats : Trans.pat_macro

remove_unit_pats replaces unit-patterns () with wildcard ones _.

val coq_type_annot_pat_vars : Trans.pat_macro

Add type annotations to pattern variables whose type contains a type variable (only add for arguments to top-level functions)

30.5 Type Class Macros

val remove_method : Target.target -> bool -> Typed_ast.exp Trans.macro

remove_method target add_dict is used to remove occurrences of class methods. If a class method is encountered, the remove_method macro first tries to resolve the type-class instantiation statically and replace the method with it’s instantiation. If this static resolving attempt fails, it is checked, whether the method is inlined for this target. If this is not the case and the flag add_dict is set, the method is replaced with a lookup in a dictionary. This dictionary is added by the Def_trans.class_constraint_to_parameter to the arguments of each definition that has type class constraints.

val remove_method_pat : Trans.pat_macro

remove_method_pat is used to remove occurrences of class methods. If a class method is encountered, remove_method_pat macro tries to resolve the type-class instantiation statically and replace the method with it’s instantiation.

val remove_num_lit : Typed_ast.exp Trans.macro

remove_num_lit replaces L_num (sk, i) with fromNumeral (L_numeral (sk, i)). This is the first step into using type classes to handle numerals.

val remove_class_const : Target.target -> Typed_ast.exp Trans.macro

remove_class_const remove constants that have class constraints by adding explicit dictionary parameters.
30.6 Misc

val remove_function : Typed_ast.exp Trans.macro

remove_function turns function | pat1 -> exp1 ... | patn -> expn end into
fun x -> match x with | pat1 -> exp1 ... | patn -> expn end.

val remove_sets : Typed_ast.exp Trans.macro

Warning: OCaml specific! remove_sets transforms set expressions like \{e1, ..., en\}
into Ocaml.Pset.from_list (type_specific compare) \[e1, ..., en\]

val remove_fun_pats : bool -> Typed_ast.exp Trans.macro

remove_fun_pats keep_tup removes patterns from expressions of the from fun pi ... pn -> e by introducing function expressions. Variable patterns and - if keep_tup is
set - tuple patterns are kept.

30.7 Macros I don’t understand

val add_nexp_param_in_const : Typed_ast.exp Trans.macro
val remove_vector_access : Typed_ast.exp Trans.macro
val remove_vector_sub : Typed_ast.exp Trans.macro
val remove_do : Typed_ast.exp Trans.macro

end

31 Module Typecheck: check_defs backend_targets mod_name filename

mod_in_output env ast typescheck the parsed module ast from
file filename in environment env.

It is assumed that mainly the backends backend_targets will be used later, i.e. only for these
backends problems like missing definitions are reported. However, information for all targets is still
The new definitions are added to the environment as new module mod_name. The result is a new
environment as well as the type-checked ast of the module. The flag mod_in_output is stored in
the resulting module description. It signals, whether the module will be written to file.

val check_defs :
    Target.Targetset.t ->
    Name.t ->
    string ->
    bool ->
    Typed_ast.env ->
    Ast.defs * Ast.lex_skips ->
    Typed_ast.env * (Typed_ast.def list * Ast.lex_skips)
check_defs backend_targets mod_name filename mod_in_output env ast typescheck
the parsed module ast from file filename in environment env. It is assumed that mainly
the backends backend_targets will be used later, i.e. only for these backends problems like
missing definitions are reported. However, information for all targets is still The new
definitions are added to the environment as new module mod_name. The result is a new
environment as well as the type-checked ast of the module. The flag mod_in_output is stored
in the resulting module description. It signals, whether the module will be written to file.

32 Module Typecheck_ctxt: The distinction between cur_env, new_defs
and export_env is interesting.

cur_env contains the local environment as seen by a function inside the module. new_defs in
contrast contains only the definitions made inside the module. It is used to check for duplicate
definitions. export_env is the outside view of the module. It contains all definitions made inside
the module (i.e. new_defs) as well as the included modules (see command include).

type defn_ctxt = {
  all_tdefs : Types.type_defs ;
  ctxt_c_env : Typed.ast.c_env ;
  ctxt_e_env : Typed.ast.mod_descr Types.Pfmap.t ;
  all_instances : Types.i_env ;
  lemmata_labels : Typed.ast.NameSet.t ;
  ctxt_mod_target_rep : Typed.ast.mod_target_rep Target.Targetmap.t ;
  ctxt_mod_in_output : bool ;
  cur_env : Typed.ast.local_env ;
  new_defs : Typed.ast.local_env ;
  export_env : Typed.ast.local_env ;
  new_tdefs : Path.t list ;
  new_instances : Types.instance_ref list ;
}

val add_d_to_ctxt : defn_ctxt ->
Path.t -> Types.tc_def -> defn_ctxt
The distinction between cur_env, new_defs and export_env is interesting. cur_env
contains the local environment as seen by a function inside the module. new_defs in contrast
contains only the definitions made inside the module. It is used to check for duplicate
definitions. export_env is the outside view of the module. It contains all definitions made
inside the module (i.e. new_defs) as well as the included modules (see command include).

val add_p_to_ctxt : defn_ctxt ->
Name.t * (Path.t * Ast.l) -> defn_ctxt

val add_f_to_ctxt :
  defn_ctxt ->
Name.t * Types.const_descr_ref -> defn_ctxt

50
val add_v_to_ctxt :  
  defn_ctxt ->  
  Name.t * Types.const_descr_ref -> defn_ctxt

val union_v_ctxt :  
  defn_ctxt ->  
  Typed_ast.const_descr_ref Typed_ast.Nfmap.t -> defn_ctxt

val add_m_to_ctxt :  
  Ast.l ->  
  defn_ctxt ->  
  Name.t -> Typed_ast.mod_descr -> defn_ctxt

val add_m_alias_to_ctxt :  
  Ast.l ->  
  defn_ctxt -> Name.t -> Path.t -> defn_ctxt

val add_instance_to_ctxt :  
  defn_ctxt ->  
  Types.instance -> defn_ctxt * Types.instance_ref

val add_lemma_to_ctxt : defn_ctxt -> Name.t -> defn_ctxt

val defn_ctxt_to_env : defn_ctxt -> Typed_ast.env

A definition context contains among other things an environment split up over several fields. This function extracts this environment.

val ctxt_c_env_set_target_rep :  
  Ast.l ->  
  defn_ctxt ->  
  Typed_ast.const_descr_ref ->  
  Target.non_ident_target ->  
  Typed_ast.const_target_rep ->  
  defn_ctxt * Typed_ast.const_target_rep option

cxt_c_env_set_target_rep l ctxt c targ new_rep updates the target-representation of constant c for target targ in context ctxt to new_rep. This results into a new environment. If an representation was already stored (and is now overridden), it is returned as well. If it can’t be overridden, an exception is raised.

val ctxt_all_tdefs_set_target_rep :  
  Ast.l ->  
  defn_ctxt ->  
  Path.t ->  
  Target.non_ident_target ->  
  Types.type_target_rep ->  
  defn_ctxt * Types.type_target_rep option

cxt_all_tdefs_set_target_rep l ctxt ty targ new_rep updates the target-representation of type ty for target targ in context ctxt to new_rep. This results into a new environment. If an representation was already stored (and is now overridden), it is returned as well.
val ctxt_begin_submodule : defn_ctxt -> defn_ctxt

cxt_start_submodule ctxt is used when a new submodule is processed. It resets all the
new-information like the field new_defs, but keeps the other informations (including the
current environment) around.

val ctxt_end_submodule :
    Ast.l ->
    defn_ctxt ->
    Name.t list -> Name.t -> defn_ctxt -> defn_ctxt

cxt_end_submodule l ctxt_before mod_path mod_name cxt_submodule is used when a
new submodule is no longer processed. It resets some information (like the local
environment of cxt_submodule back to the values in cxt_before. The context
ctxt_before is supposed to be the one valid before starting to process the submodule. The
new definitions of the submodule are moved to a new module mod_name at path mod_path.

33 Module Typed_ast : Sets of Names

module NameSet :
    Set.S with type elt = Name.t and type t = Set.Make(Name).t
Sets of Names

module Nfmap :
    Finite_map.Fmap with type k = Name.t
Name keyed finite maps

val nfmap_domain : 'a Nfmap.t -> NameSet.t

val no_lskips : lskips
The empty lskip

val space : lskips
A space lskip

val lskips_only_comments : lskips list -> lskips
Get only the comments (and a trailing space)

val lskips_only_comments_first : lskips list -> lskips
Get the first lskip of the list and only comments from the rest
type env_tag =
  | K_let
    A let definition, the most common case. Converts val as well, details see above.
  | K_field
    A field
  | K_constr
    A type constructor
  | K_relation
    A relation
  | K_method
    A class method
  | K_instance
    A method instance

env_tag is used by const_descr to describe the type of constant. Constants can be defined
in multiple ways: the most common way is via a let-statement. Record-type definitions
introduce fields-accessor functions and variant types introduce constructors. There are
methods, instances and relations as well. A let definition can be made via a val definition
and multiple, target specific lets.

type p_env = (Path.t * Ast.l) Nfmap.t
  Maps a type name to the unique path representing that type and the first location this type
  is defined

type lit = (lit_aux, unit) Types.annot

type lit_aux =
  | L_true of lskips
  | L_false of lskips
  | L_zero of lskips
    This is a bit, not a num
  | L_one of lskips
    see above
  | L_numeral of lskips * int * string option
    A numeral literal, it has fixed type "numeral" and is used in patterns and after
    translating L_num to it.
  | L_num of lskips * int * string option
    A number literal. This is like a numeral one wrapped with the "from_numeral"
    function
  | L_char of lskips * char * string option

53
A char literal. It contains the parsed char as well as the original input string (if available).

| L_string of lskips * string * string option |
| A string literal. It contains the parsed string as well as the original input string (if available). |

| L_unit of lskips * lskips |
| L_vector of lskips * string * string |
| For vectors of bits, specified with hex or binary, first string is either 0b or 0x, second is the binary or hex number as a string |

| L_undefined of lskips * string |
| A special undefined value that explicitly states that nothing is known about it. This is useful for expressing underspecified functions. It has been introduced to easier support pattern compilation of non-exhaustive patterns. |

type const_descr_ref = Types.const_descr_ref

type name_kind =
| Nk_typeconstr of Path.t |
| Nk_const of const_descr_ref |
| Nk_constr of const_descr_ref |
| Nk_field of const_descr_ref |
| Nk_module of Path.t |
| Nk_class of Path.t |

type pat = (pat_aux, pat_annot) Types.annot

type pat_annot = {
  pvars : Types.t Nfmap.t ;
}

type pat_aux =
| P_wild of lskips |
| P_as of lskips * pat * lskips * name_l |
| P_typ of lskips * pat * lskips * Types.src_t |
| P_var of Name.lskips_t |
| P_const of const_descr_ref Types.id * pat list |
| P_backend of lskips * Ident.t * Types.t * pat list |
| P_record of lskips |
| (const_descr_ref Types.id * lskips * pat) |
| lskips_seplist * lskips |
| P_vector of lskips * pat lskips_seplist * lskips |
| P_vectorC of lskips * pat list * lskips |
| P_tup of lskips * pat lskips_seplist * lskips |
| P_list of lskips * pat lskips_seplist * lskips |
| P_paren of lskips * pat * lskips |
A type-annotated pattern variable. This is redundant with the combination of the
P_typ and P_var cases above, but useful as a macro target.

type cr_special_fun =
  | CR_special_uncurry of int
  | CR_special_rep of string list * exp list

CR_special_uncurry n formats a function with n arguments curried, i.e. turn the
arguments into a tupled argument, surrounded by parenthesis and separated by ",".

CR_special_rep sr args encodes a user given special representation. replace the
arguments in the expression list and then interleave the results with sr

type const_target_rep =
  | CR_inline of Ast.l * bool * name_lskips_annot list * exp
  | CR_infix of Ast.l * bool * Ast.fixity_decl * Ident.t
  | CR_undefined of Ast.l * bool
  | CR_simple of Ast.l * bool * name_lskips_annot list * exp
  | CR_special of Ast.l * bool * cr_special_fun * name_lskips_annot list

CR_inline (loc, allow_override, vars, e) means inlining the constant with the
expression e and replacing the variable vars inside e with the arguments of the
constant. The flag allow_override signals whether the declaration might be safely
overridden. Automatically generated target-representations (e.g. for ocaml
constructors) should be changeable by the user, whereas multiple user-defined ones
should cause a type error.

CR_infix (loc, allow_override, fixity, i) declares infix notation for the
constant with the giving identifier.

CR_undefined (loc, allow_override) declares undefined constant.

CR_simple (loc, allow_override, vars, e) is similar to CR_inline. Instead of
inlining during macro expansion and therefore allowing further processing afterwards,
CR_simple performs the inlining only during printing in the backend.

CR_special (loc, allow_override, to_out, vars) describes special formatting of
this constant. The (renamed) constant (including path prefix) and all arguments are
transformed to output. to_out represents a function that is then given the formatted
name and the appropriate number of these outputs. The expected arguments are
described by vars. If there are more arguments than variables, they are appended. If
there are less, for expressions local functions are introduced. For patterns, an
exception is thrown. Since values of const_target_rep need to be written out to file
via output_value in order to cache libraries, it cannot be a function of type Output.t
list -> Output.t list directly. Instead, the type cr_special_fun is used as an
indirection.

55
type rel_io =
  | Rel_mode_in
  | Rel_mode_out

rel_io represents whether an argument of an inductive relation is considered as an input or an output

type rel_mode = rel_io list

rel_output_type specifies the type of the result

type rel_output_type =
  | Out_list
    Return a list of possible outputs
  | Out_pure
    Return one possible output or fail if no such output exists
  | Out_option
    Return one possible output or None if no such output exists
  | Out_unique
    Return the output if it is unique or None otherwise

type rel_info = {
  ri_witness : (Path.t * const_descr_ref Nfmap.t) option ;
    Contains the path of the witness type and a mapping from rules to constructors. None
    if no witness type has been generated
  ri_check : const_descr_ref option ;
    A reference to the witness checking function or None if it is not generated
  ri_fns : ((rel_mode * bool * rel_output_type) *
    const_descr_ref)
    list ;
    A list of functions generated from the relation together with their modes
}

rel_info represents information about functions and types generated from this relation

type const_descr = {
  const_binding : Path.t ;
    The path to the definition
  const_tparams : Types.tnvar list ;
    Its type parameters. Must have length 1 for class methods.
  const_class : (Path.t * Types.tnvar) list ;
    Its class constraints (must refer to above type parameters). Must have length 1 for
class methods
If the constant has constraints, i.e. const_class is not empty, we need another constant without constraints for dictionary passing. This field stores the reference to this constant, if one such constant has already been generated.

Its length constraints (must refer to above type parameters). Can be equality or GtEq inequalities

Its type

If the constant is a relation, it might contain additional information about this relation. However, it might be None for some relations as well.

What kind of definition it is.

The set of targets the constant is defined for.

The location for the first occurrence of a definition/specification of this constant.

Target-specific renames of for this constant.

Optional ASCII representation for this constant.

Target-specific representation of for this constant.

An optional warning message that should be printed, if the constant is used

Can termination be proved automatically by various backends?

The description of a top-level definition

type v_env = const_descr_ref Nfmap.t

type f_env = const_descr_ref Nfmap.t

type m_env = Path.t Nfmap.t

type e_env = mod_descr Types.Pfmap.t

type c_env
local_env represents local_environments, i.e. essentially maps from names to the entities they represent

```ocaml
type local_env = {
    m_env : m_env ;
    module map
    p_env : p_env ;
    type map
    f_env : f_env ;
    field map
    v_env : v_env ;
    constructor and constant map
}

type env = {
    local_env : local_env ;
    the current local environment
    c_env : c_env ;
    global map from constant references to the constant descriptions
    t_env : Types.type_defs ;
    global type-information
    i_env : Types.i_env ;
    global instances information
    e_env : e_env ;
    global map from module paths to the module descriptions
}

type mod_target_rep =
    | MR_rename of Ast.l * Name.t
       Rename the module

type mod_descr = {
    mod_binding : Path.t ;
    The full path of this module
    mod_env : local_env ;
    The local environment of the module
    mod_target_rep : mod_target_rep Target.Targetmap.t ;
    how to represent the module for different backends
    mod_filename : string option ;
    the filename the module is defined in (if it is a top-level module)
```
mod_in_output : bool;

    is this module written to a file (true) or an existing file used (false) ?

}

type exp

type exp_subst =
    | Sub of exp
    | Sub_rename of Name.t

type exp_aux = private
    | Var of Name.lskips_t
    | Backend of lskips * Ident.t

    An identifier that should be used literally by a backend. The identifier does not
    contain whitespace. Initial whitespace is represented explicitly.

    | Nvar_e of lskips * Nvar.t
    | Constant of const_descr_ref Types.id
    | Fun of lskips * pat list * lskips * exp
    | Function of lskips
    * (pat * lskips * exp * Ast.l)
    lskips_seplist * lskips
    | App of exp * exp
    | Infix of exp * exp * exp

    The middle exp must be a Var, Constant, or Constructor

    | Record of lskips * fexp lskips_seplist * lskips
    | Recup of lskips * exp * lskips
    * fexp lskips_seplist * lskips
    | Field of exp * lskips * const_descr_ref Types.id
    | Vector of lskips * exp lskips_seplist * lskips
    | VectorSub of exp * lskips * Types.src_nexp * lskips
    * Types.src_nexp * lskips
    | VectorAcc of exp * lskips * Types.src_nexp * lskips
    | Case of bool * lskips * exp * lskips
    * (pat * lskips * exp * Ast.l)
    lskips_seplist * lskips

    The boolean flag as first argument is used to prevent pattern compilation from
    looping in rare cases. If set to true, no pattern compilation is tried. The default value
    is false.

    | Typed of lskips * exp * lskips * Types.src_t
    * lskips
    | Let of lskips * letbind * lskips * exp
    | Tup of lskips * exp lskips_seplist * lskips
    | List of lskips * exp lskips_seplist * lskips
    | Paren of lskips * exp * lskips
    | Begin of lskips * exp * lskips
If of lskips * exp * lskips * exp
Lit of lit
Set of lskips * exp lskips * exp
Setcomp of lskips * exp * lskips * exp
* lskips * NameSet.t
Comp_binding of bool * lskips * exp * lskips * lskips
* quant_binding list * lskips * exp
* lskips
true for list comprehensions, false for set comprehensions
Quant of Ast.q * quant_binding list * lskips * exp
Do of lskips * Path.t Types.id * do_line list
* lskips * exp * lskips
* (Types.t * bind_tyargs_order)
The last argument is the type of the value in the monad

The last argument is the type of the value in the monad

A bind constant of a monad M has type M 'a -> ('a -> M 'b) -> M 'b. Here, I call 'a
the input type and 'b the output type. Depending on how the bind constant is defined in
detail its free type variable list (stored in constant-description record, field const_tparams)
might be either of the form ['a, 'b] or ['b, 'a]. This type is used to distinguish the two
possibilities.

A bind constant of a monad M has type M 'a -> ('a -> M 'b) -> M 'b. Here, I call 'a
the input type and 'b the output type. Depending on how the bind constant is defined in
detail its free type variable list (stored in constant-description record, field const_tparams)
might be either of the form ['a, 'b] or ['b, 'a]. This type is used to distinguish the two
possibilities.
Let_val \( (p, \text{ty} \_\text{opt}, \text{sk}, e) \) describes binding the pattern \( p \) to expr \( e \) in a local let statement, i.e. a statement like \texttt{let} \( p = e \) \texttt{in} ... 

| Let_fun of name_lskips_annot * pat list * (lskips * Types.src_t) option * lskips * exp 
| Let_fun \((n, ps, \text{ty}_\text{opt}, \text{sk}, e)\) describes defining a local function \( f \) with arguments \( ps \) locally. It represents a statement like \texttt{let} \( n \) \( ps = e \) \texttt{in} .... Notice that the arguments of Let_fun are similar to funcl_aux. However, funcl_aux has a constant-references, as it is used in top-level definitions, whereas Let_fun is used only for local functions.

```plaintext
type tyvar = lskips * Ulib.Text.t * Ast.l
type nvar = lskips * Ulib.Text.t * Ast.l

type tnvar =  
| Tn_A of tyvar  
| Tn_N of nvar

type texp =  
| Te_opaque  
| Te_abbrev of lskips * Types.src_t  
| Te_record of lskips * lskips * (name_l * const_descr_ref * lskips * Types.src_t) * lskips_seplist * lskips  
| Te_variant of lskips * (name_l * const_descr_ref * lskips * Types.src_t lskips_seplist) * lskips_seplist

type range =  
| GtEq of Ast.l * Types.src_nexp * lskips * Types.src_nexp  
| Eq of Ast.l * Types.src_nexp * lskips * Types.src_nexp

type constraints =
```

Type expressions for defining types

Type expressions for defining types
type constraint_prefix =
  | Cp_forall of lskips * tnvar list * lskips
  * constraints option

type typschm = constraint_prefix option * Types.src_t

type instschm = constraint_prefix option * lskips * Ident.t * Path.t *
  Types.src_t * lskips

Instance Scheme, constraint prefix, sk, class-ident as printed, resolved class-path the id
points to, instantiation type, sk

val cr_special_fun_uses_name : cr_special_fun -> bool

  cr_special_fun_uses_name f checks, whether f uses it’s first argument, i.e. whether it
uses the formatted name of the constant. This information is important to determine,
whether the constant needs to be renamed.

type targets_opt =
  | Targets_opt_none
    represents the universal set, i.e. all targets
  | Targets_opt_concrete of lskips * Ast.target lskips_seplist * lskips
    (in source '{ t1; ...; tn }') is the set of all targets in the list ‘tl’
  | Targets_opt_neg_concrete of lskips * Ast.target lskips_seplist * lskips
    (in source '˜{ t1; ...; tn }') is the set of all targets not in the list ‘tl’
  | Targets_opt_non_exec of lskips
    (in source ‘non_exec’) is the set of all targets that can handle non-executable
definitions

targets_opt is represents a set of targets

val in_targets_opt : Target.target -> targets_opt -> bool

  in_targets_opt targ targets_opt checks whether the target ‘targ’ is in the set of targets
represented by ‘targets_opt’. If targ is the a human readable target, true is returned.

val in_target_set : Target.target -> Target.Targetset.t -> bool

  in_target_set targ targetset checks whether the target ‘targ’ is in the set of targets
targetset. It is intended for checking whether to output certain parts of the TAST.
Therefore, in_target_set returns true for all human readable targets and only checks for
others.

val targets_opt_to_list : targets_opt -> Target.non_ident_target list

  target_opt_to_list targets_opt returns a distinct list of all the targets in the option.
type val_spec = lskips * name_l * const_descr_ref * Ast.ascii_opt * lskips * typschm

type class_val_spec = lskips * targets_opt * name_l * const_descr_ref * Ast.ascii_opt * lskips * Types.src_t

type fun_def_rec_flag =
| FR_non_rec
| FR_rec of lskips

fun_def_rec_flag is used to encode whether a Fun_def is recursive. The recursive one carries some whitespace for printing after the rec-keyword.

type val_def =
| Let_def of lskips * targets_opt
  * (pat * (Name.t * const_descr_ref) list *
    (lskips * Types.src_t) option * lskips *
    exp)
| Fun_def of lskips * fun_def_rec_flag * targets_opt
  * funcl_aux lskips_seplist
    Fun_def (sk1, rec_flag, topt, clauses) encodes a function definition, which might consist of multiple clauses.
| Let_inline of lskips * lskips * targets_opt
  * name_lskips_annot * const_descr_ref
  * name_lskips_annot list * lskips * exp

type name_sect =
| Name_restrict of (lskips * name_l * lskips * lskips *
  string * lskips)

type indreln_rule_quant_name =
| QName of name_lskips_annot
| Name_typ of lskips * name_lskips_annot * lskips
  * Types.src_t * lskips

type indreln_rule_aux =
| Rule of Name.lskips_t * lskips * lskips
  * indreln_rule_quant_name list * lskips
  * exp option * lskips * name_lskips_annot
  * const_descr_ref * exp list
  A rule of the form Rule(clause_name_opt, sk1, sk2, bound_vars, sk3,
    left_hand_side_opt, sk4, rel_name, c, args) encodes a clause clause_name: forall
    bound_vars. (left_hand_side ==> rel_name args). c is the reference of the relation
    rel_name.

type indreln_rule = indreln_rule_aux * Ast.l

type indreln_witness =
| Indreln_witness of lskips * lskips * Name.lskips_t * lskips
  Name of the witness type to be generated
type indreln_indfn =
| Indreln_fn of Name.lskips_t * lskips * Types.src_t * lskips option

Name and mode of a function to be generated from an inductive relation

type indreln_name =
| RName of lskips * Name.lskips_t * const_descr_ref
* lskips * typeschm * indreln_witness option
* (lskips * Name.lskips_t * lskips) option
* indreln_indfn list option * lskips

Type annotation for the relation and information on what to generate from it. RName(sk1, rel_name, rel_name_ref, sk2, rel_type, witness_opt, check_opt, indfns opt, sk3)

type target_rep_rhs =
| Target_rep_rhs_infix of lskips * Ast.fixity_decl * lskips * Ident.t

Declaration of an infix constant

| Target_rep_rhs_term_replacement of exp
the standard term replacement, replace with the exp for the given backend

| Target_rep_rhs_type_replacement of Types.src_t
the standard term replacement, replace with the type for the given backend

| Target_rep_rhs_special of lskips * lskips * string * exp list
fancy represenation of terms

| Target_rep_rhs_undefined
undefined, don’t throw a problem during typeching, but during output

type declare_def =
| Decl_compile_message of lskips * targets_opt * lskips
* const_descr_ref Types.id * lskips * lskips
* string

Decl_compile_message_decl (sk1, targs, sk2, c, sk3, sk4, message), declares printing waring message message, if constant c is used for one of the targets in targs

| Decl_target_rep_term of lskips * Ast.target * lskips * Ast.component
* const_descr_ref Types.id * name_lskips_annot list
* lskips * target_rep_rhs
Decl_target_rep_term (sk1, targ, sk2, comp, c, args, sk3, rhs) declares a target-representation for target targ and constant c with arguments args. Since fields and constant live in different namespaces, comp is used to declare whether a field or a constant is meant. The rhs contains details about the representation.

| Decl_target_rep_type of lskips * Ast.target * lskips * lskips
* Path.t Types.id * tnvar list * lskips * Types.src_t
Decl_target_rep_type (sk1, targ, sk2, sk3, id, args, sk4, rhs) declares a target-representation. for target targ and type id with arguments args.
type def_aux =
  Type_def of lskips
  * (name_l * tnvar list * Path.t * texp *
    name_sect option)
  lskips_seplist
  Type_def (sk, sl) defines one or more types. The entries of sl are the type
definitions. They contain a name of the type, the full path of the defined type, the free
type variables, the main type definition and restrictions on variable names of this type

| Val_def of val_def
  The list contains the class constraints on those variables

| Lemma of lskips * Ast.lemma_typ * targets_opt * name_l
  * lskips * exp
| Module of lskips * name_l * Path.t * lskips
  * lskips * def list * lskips
| Rename of lskips * name_l * Path.t * lskips
  * Path.t Types.id
  Renaming an already defined module

| OpenImport of Ast.open_import * Path.t Types.id list
  open and/or import modules
| OpenImportTarget of Ast.open_import * targets_opt * (lskips * string) list
  open and/or import modules only for specific targets

| Indreln of lskips * targets_opt
  * indreln_name lskips_seplist
  * indreln_rule lskips_seplist
  Inductive relations

| Val_spec of val_spec
| Class of Ast.class_decl * lskips * name_l * tnvar

65
type def = (def_aux * lskips option) * Ast.l * local_env

A definition consists of a the real definition represented as a def_aux, followed by some white-space. There is also the location of the definition and the local-environment present after the definition has been processed.

val tnvar_to_types_tnvar : tnvar -> Types.tnvar * Ast.l
val empty_local_env : local_env
val empty_env : env
val e_env_lookup : Ast.l -> e_env -> Path.t -> mod_descr
    e_env_lookup l e_env p looks up the module with path p in environment e_env and returns the corresponding description. If this lookup fails, a fatal error is thrown using location l for the error message.

val c_env_lookup : Ast.l -> c_env -> const_descr_ref -> const_descr
    c_env_lookup l c_env c_ref looks up the constant reference c_ref in environment c_env and returns the corresponding description. If this lookup fails, a fatal error is thrown using location l for the error message.

val c_env_store_raw : c_env -> constdescr -> c_env * constdescr_ref
    c_env_store_raw c_env c_d stores the description c_d environment c_env. Thereby, a new unique reference is generated and returned along with the modified environment. It stores the real c_d passed. The function c_env_store preprocesses c_d to add common features like for example capitalizing constructors for the Ocaml backend.

val c_env_update : c_env -> constdescr_ref -> constdescr -> c_env
    c_env_update c_env c_ref c_d updates the description of constant c_ref with c_d in environment c_env.

val env_c_env_update : env -> constdescr_ref -> constdescr -> env
env_c_env_update env c_ref c_d updates the description of constant c_ref with c_d in environment env.

val c_env_all_consts : c_env -> const_descr_ref list
  c_env_all_consts c_env returns the constants defined in c_env

val exp_to_locn : exp -> Ast.l
val exp_to_typ : exp -> Types.t
val append_lskips : lskips -> exp -> exp
  append_lskips adds new whitespace/newline/comments to the front of an expression (before any existing whitespace/newline/comments in front of the expression)

val pat_append_lskips : lskips -> pat -> pat
val alter_init_lskips : (lskips -> lskips * lskips) ->
  exp -> exp * lskips
  alter_init_lskips finds all of the whitespace/newline/comments preceding an expression and passes it to the supplied function in a single invocation. The preceding whitespace/newline/comments are replaced with the fst of the function’s result, and the snd of the function’s result is returned from alter_init_lskips

val pat_alter_init_lskips : (lskips -> lskips * lskips) ->
  pat -> pat * lskips
val def_aux_alter_init_lskips : (lskips -> lskips * lskips) ->
  def_aux -> def_aux * lskips
val def_alter_init_lskips : (lskips -> lskips * lskips) ->
  def -> def * lskips
val oi_alter_init_lskips : (lskips -> lskips * lskips) ->
  Ast.open_import -> Ast.open_import * lskips
val pp_const_descr : Format.formatter -> const_descr -> unit
val pp_env : Format.formatter -> env -> unit
val pp_local_env : Format.formatter -> local_env -> unit
val pp_c_env : Format.formatter -> c_env -> unit
val pp_instances :
  Format.formatter -> Types.instance list Types.Pfmap.t -> unit

type imported_modules =
  | IM_paths of Path.t list
  | IM_targets of targets_opt * string list

type checked_module = {
  filename : string ;
module_path : Path.t ;
imported_modules : imported_modules list ;
imported_modules_rec : imported_modules list ;
untyped_ast : Ast.defs * Ast.lex_skips ;
typed_ast : def list * Ast.lex_skips ;
generate_output : bool ;
}

type var_avoid_f = bool * (Name.t -> bool) * (Ulib.Text.t -> (Name.t -> bool) -> Name.t)

var_avoid_f is a type of a tuple (avoid_ty_vars, name_ok, do_rename). The flag
avoid_ty_vars states, whether clashes with type variables should be avoided. The name_ok
n checks whether the name n is OK. If it is not OK, the function do_rename n_text check
renames n. As input it takes the text of n, a function check that checks whether the new
name clashes with any names to be avoided or existing variable names in the context.

module type Exp_context =

    sig
       val env_opt : Typed_ast.env option
               The environment the expressions are considered in
       val avoid : Typed_ast.var_avoid_f option
               Avoiding certain names for local variables. Given a name and a set of names that must
               be avoided, choose a new name if necessary
    end

module Exps_in_context :
    functor (C : Exp_context) -> sig
    val exp_subst :
        Types.t Types.TNfmap.t * Typed_ast.exp_subst Typed_ast.Nfmap.t ->
        Typed_ast.exp -> Typed_ast.exp
    val push_subst :
        Types.t Types.TNfmap.t * Typed_ast.exp_subst Typed_ast.Nfmap.t ->
        Typed_ast.pat list -> Typed_ast.exp -> Typed_ast.pat list * Typed_ast.exp
    val exp_to_term : Typed_ast.exp -> Typed_ast.exp_aux
    val exp_to_free : Typed_ast.exp -> Types.t Typed_ast.Nfmap.t
    val type_eq : Ast.l -> string -> Types.t -> Types.t -> unit
    val mk_lnumeral :
        Ast.l ->
        Typed_ast.lskips -> int -> string option -> Types.t option -> Typed_ast.lit
    val mk_lnum :
        Ast.l -> Typed_ast.lskips -> int -> string option -> Types.t -> Typed_ast.lit
    val mk_lbool :
val mk_lbit :Ast.l -> Typed_ast.lskips -> int -> Types.t option -> Typed_ast.lit
val mk_lundef :Ast.l -> Typed_ast.lskips -> string -> Types.t -> Typed_ast.lit
val mk_lstring :Ast.l -> Typed_ast.lskips -> string -> Types.t option -> Typed_ast.lit
val mk_twild :Ast.l -> Typed_ast.lskips -> Types.t -> Types.src_t
val mk_tvar :Ast.l -> Typed_ast.lskips -> Tyvar.t -> Types.t -> Types.src_t
val mk_tfn :Ast.l -> Types.src_t -> Typed_ast.lskips -> Types.src_t -> Types.t option -> Types.src_t
val mk_ttup :Ast.l -> Types.src_t -> Typed_ast.lskips -> Types.src_t -> Types.t option -> Types.src_t
val mk_tapp :Ast.l -> Path.t -> Types.id -> Types.src_t list -> Types.t option -> Types.src_t
val mk_tparen :Ast.l -> Typed_ast.lskips -> Types.src_t -> Typed_ast.lskips -> Types.t option -> Types.src_t
val mk_wp wild :Ast.l -> Typed_ast.lskips -> Types.t -> Typed_ast.pat
val mk_pas :Ast.l -> Typed_ast.lskips -> Typed_ast.pat -> Typed_ast.lskips ->
           Typed.ast.name_l -> Types.t option -> Typed.ast.pat
val mk_ptyp :Ast.l -> Typed.ast.lskips -> Typed.ast.pat ->
           Typed.ast.lskips -> Types.src_t ->
           Types.t option -> Typed.ast.pat
val mk_pconst :Ast.l -> Typed.ast.const_descr_ref ->
               Types.id ->
               Types.t option -> Typed.ast.pat
val mk_pbackend :Ast.l ->
               Name.lskips_t ->
               Types.t -> Typed.ast.pat
Ident.t -> Types.t -> Typed_ast.pat list -> Types.t option -> Typed_ast.pat

val mk_precord :
  Ast.l ->
  Typed_ast.lskips ->
  (Typed.ast.const_descr_ref Types.id * Typed.ast.lskips * Typed.ast.pat)
  Typed.ast.lskips_seplist ->
  Typed.ast.lskips -> Types.t option -> Typed.ast.pat

val mk_ptup :
  Ast.l ->
  Typed.ast.lskips ->
  Typed.ast.pat Typed.ast.lskips_seplist ->
  Typed.ast.lskips -> Types.t option -> Typed.ast.pat

val mk_plist :
  Ast.l ->
  Typed.ast.lskips ->
  Typed.ast.pat Typed.ast.lskips_seplist ->
  Typed.ast.lskips -> Types.t -> Typed.ast.pat

val mk_pvector :
  Ast.l ->
  Typed.ast.lskips ->
  Typed.ast.pat Typed.ast.lskips_seplist ->
  Typed.ast.lskips -> Types.t -> Typed.ast.pat

val mk_pvectorc :
  Ast.l ->
  Typed.ast.lskips ->
  Typed.ast.pat list -> Typed.ast.lskips -> Types.t -> Typed.ast.pat

val mk_pparen :
  Ast.l ->
  Typed.ast.lskips ->
  Typed.ast.pat -> Typed.ast.lskips -> Types.t option -> Typed.ast.pat

val mk_pcons :
  Ast.l ->
  Typed.ast.pat ->
  Typed.ast.lskips ->
  Typed.ast.pat -> Types.t option -> Typed.ast.pat

val mk_pnum_add :
  Ast.l ->
  Typed.ast.name_l ->
  Typed.ast.lskips ->
  Typed.ast.lskips -> int -> Types.t option -> Typed.ast.pat

val mk_plit : Ast.l ->
  Typed.ast.lit ->
  Types.t option -> Typed.ast

val mk_pvar_annot :
  Ast.l ->
  Name.lskips_t ->
  Types.src_t ->
  Types.t option -> Typed.ast.pat

val mk_var : Ast.l ->
  Name.lskips_t ->
  Types.t -> Typed.ast.exp

70
val mk_nvar_e : Ast.l -> Typed_ast.lskips -> Nvar.t -> Types.t -> Typed_ast.exp
val mk_backend : Ast.l -> Typed_ast.lskips -> Ident.t -> Types.t -> Typed_ast.exp
val mk_const : Ast.l ->
Typed_ast.const_descr_ref Types.id -> Types.t option -> Typed_ast.exp
val mk_fun : Ast.l ->
Typed_ast.lskips ->
Typed_ast.pat list ->
Typed_ast.lskips -> Typed_ast.exp -> Types.t option -> Typed_ast.exp
val mk_function :
Ast.l ->
Typed_ast.lskips ->
(Typed.ast.pat * Typed.ast.lskips * Typed.ast.exp * Ast.l)
Typed.ast.lskips_seplist ->
Typed.ast.lskips -> Types.t option -> Typed.ast.exp
val mk_app :
Ast.l -> Typed.ast.exp -> Typed.ast.exp -> Types.t option -> Typed.ast.exp
val mk_infix :
Ast.l ->
Typed.ast.exp ->
Typed.ast.exp -> Typed.ast.exp -> Types.t option -> Typed.ast.exp
val mk_record :
Ast.l ->
Typed.ast.lskips ->
(Typed.ast.const_descr_ref Types.id * Typed.ast.lskips * Typed.ast.exp * Ast.l)
Typed.ast.lskips_seplist ->
Typed.ast.lskips -> Types.t option -> Typed.ast.exp
val mk_recup :
Ast.l ->
Typed.ast.lskips ->
Typed.ast.exp ->
Typed.ast.lskips ->
(Typed.ast.const_descr_ref Types.id * Typed.ast.lskips * Typed.ast.exp * Ast.l)
Typed.ast.lskips_seplist ->
Typed.ast.lskips -> Types.t option -> Typed.ast.exp
val mk_field :
Ast.l ->
Typed.ast.exp ->
Typed.ast.lskips ->
Typed_ast.const_descr_ref Types.id -> Types.t option -> Typed_ast.exp

val mk_case :
  bool ->
  Ast.l ->
  Typed_ast.lskips ->
  Typed.ast.exp ->
  Typed.ast.lskips ->
  (Typed.ast.pat * Typed.ast.lskips * Typed.ast.exp * Ast.l)
  Typed.ast.lskips_seplist ->
  Typed.ast.lskips -> Types.t option -> Typed.ast.exp

val mk_typed :
  Ast.l ->
  Typed.ast.lskips ->
  Typed.ast.exp ->
  Typed.ast.lskips ->
  Types.src_t -> Typed.ast.lskips -> Types.t option -> Typed.ast.exp

val mk_let_val :
  Ast.l ->
  Typed.ast.pat ->
  (Typed.ast.lskips * Types.src_t) option ->
  Typed.ast.lskips -> Typed.ast.exp -> Typed.ast.letbind

val mk_let_fun :
  Ast.l ->
  Typed.ast.name_lskips_annot ->
  Typed.ast.pat list ->
  (Typed.ast.lskips * Types.src_t) option ->
  Typed.ast.lskips -> Typed.ast.exp -> Typed.ast.letbind

val mk_let :
  Ast.l ->
  Typed.ast.lskips ->
  Typed.ast.letbind ->
  Typed.ast.lskips -> Typed.ast.exp -> Types.t option -> Typed.ast.exp

val mk_tup :
  Ast.l ->
  Typed.ast.lskips ->
  Typed.ast.exp Typed.ast.lskips_seplist ->
  Typed.ast.lskips -> Types.t option -> Typed.ast.exp

val mk_list :
  Ast.l ->
  Typed.ast.lskips ->
  Typed.ast.exp Typed.ast.lskips_seplist ->
  Typed.ast.lskips -> Types.t -> Typed.ast.exp

val mk_vector :
  Ast.l ->
Typed_ast.lskips ->
Typed_ast.exp Typed_ast.lskips_seplist ->
Typed_ast.lskips -> Types.t -> Typed_ast.exp

val mk_vaccess :
  Ast.l ->
  Typed_ast.exp ->
  Typed_ast.lskips ->
  Types.src_nexp -> Typed_ast.lskips -> Types.t -> Typed_ast.exp

val mk_vaccessr :
  Ast.l ->
  Typed.ast.exp ->
  Typed.ast.lskips ->
  Types.src_nexp ->
  Typed.ast.lskips ->
  Types.src_nexp ->
  Typed.ast.lskips -> Types.t -> Typed.ast.exp

val mk_paren :
  Ast.l ->
  Typed.ast.lskips ->
  Typed.ast.exp ->
  Typed.ast.lskips ->
  Types.t option -> Typed.ast.exp

val mk_begin :
  Ast.l ->
  Typed.ast.lskips ->
  Typed.ast.exp ->
  Typed.ast.lskips ->
  Types.t option -> Typed.ast.exp

val mk_if :
  Ast.l ->
  Typed.ast.lskips ->
  Typed.ast.exp ->
  Typed.ast.lskips ->
  Typed.ast.exp ->
  Typed.ast.lskips ->
  Typied.ast.exp -> Types.t option -> Typed.ast.exp

val mk_lit : Ast.l -> Typed.ast.lit -> Types.t option -> Typed.ast.exp

val mk_set :
  Ast.l ->
  Typed.ast.lskips ->
  Typed.ast.exp ->
  Typed.ast.lskips ->
  Types.t option -> Typed.ast.exp

val mk_setcomp :
  Ast.l ->
  Typed.ast.lskips ->
  Typied.ast.exp ->
  Typed.ast.lskips ->
  Types.t option -> Typed.ast.exp

val mk_comp_binding :
Mutually recursive function definitions may contain multiple clauses for the same function. These can however appear interleaved with clauses for other functions.

`funcl_aux_seplist_group seplist` sorts the clauses according to the function names and states, whether any resorting was necessary. Moreover, the initial lskip is returned, if present.

`val class_path_to_dict_name : Path.t -> Types.tnvar -> Name.t` creates a name for the class `cp` with type argument `tv`. This name is used during dictionary passing. If a function has a type constraint that type-variable `tv` is of type-class `cp`, the function call `class_path_to_dict_name cp tv` is used to generate the name of a new argument. This argument is a dictionary that is used to eliminate the use of type classes.
This design is very fragile and should probably be changed in the future!

The `class_path_to_dict_name` needs to generate names that globally do not clash with anything else, including names generated by `class_path_to_dict_name` itself. The generated name is independently used by both definition macros adding the argument to the definition and expression macros that use the added argument. The name used in both places has to coincide! Therefore, the name cannot be modified depending on the context. Renaming to avoid clashes with other arguments / local variables is not possible.

val ident_get_lskip : 'a Types.id -> Ast.lex_skips
val ident_replace_lskip :
  Types.ident_option -> Ast.lex_skips -> Types.ident_option
val oi_get_lskip : Ast.open_import -> Ast.lex_skips

34 Module Typed_ast_syntax : syntax functions for typed_ast

34.1 Types

val bool_ty : Types.t
  The boolean type
val nat_ty : Types.t
  The natural number type

34.2 Navigating Environments

val lookup_env_opt :
  Typed_ast.env -> Name.t list -> Typed_ast.local_env option
  lookup_env_opt env path is used to navigate inside a environment env. It returns the local environment which is reachable via the path path. If no such environment exists, None is returned.

val lookup_env : Typed_ast.env -> Name.t list -> Typed_ast.local_env
  lookup_env is similar to lookup_env_opt, but reports an internal error instead of returning None, if no environment can be found.

val env_apply :
  Typed_ast.env ->
  Ast.component option ->
  Name.t -> (Typed_ast.name_kind * Path.t * Ast.l) option
  env_apply env comp_opt n looks up the name n in the environment env. If component comp is given, only this type of component is searched. Otherwise, it checks whether n refers to a type, field, constructor or constant. env_apply returns the kind of this name, it’s full path and the location of the original definition.
val lookup_mod_descr_opt : TypedList Astro.env -> string string string -> TypedList Astro.mod_descr option
    lookup_mod_descr_opt env path mod_name is used to navigate inside an environment env. It returns the module with name mod_name, which is reachable via the path path. If no such environment exists, None is returned.

val lookup_mod descr : TypedList Astro.env -> string string string -> TypedList Astro.mod_descr
    lookup_mod_descr env path mod_name is used to navigate inside an environment env. It returns the module with name mod_name, which is reachable via the path path. If no such environment exists, Reporting_basic is used to report an internal error.

val names_get const : TypedList Astro.env -> string string string -> TypedList Astro.const_descr_ref * TypedList Astro.const descr
    names_get const env path n looks up the constant with name n reachable via path path in the environment env.

val strings_get const : TypedList Astro.env -> string string string -> TypedList Astro.const_descr_ref * TypedList Astro.const descr
    strings_get const is a wrapper around names_get const that uses strings instead of names.

val get const : TypedList Astro.env -> string string string -> TypedList Astro.const_descr_ref * TypedList Astro.const descr
    get const env label is a wrapper around string_get const that maps a label to an actual constant description.

val const_exists : TypedList Astro.env -> string string string -> bool
    const_exists env label checks, whether the constant with label label is available in the environment env. If it is get const env label succeeds, otherwise it fails.

val names_get const ref env path n looks up the constant with name n reachable via path path in the environment env.

val const descr to kind : TypedList Astro.const descr ref * TypedList Astro.const descr -> TypedList Astro.name kind
    const descr to kind r d assumes that d is the description associated with reference r. It then determines the kind of constant (field, constructor, constant) depending on the information stored in d.

val strings_get const id : TypedList Astro.env -> string string string -> TypedList Astro.const descr ref TypedList Astro.id * TypedList Astro.const descr
    strings_get const id
strings_get_const_id env l path n inst uses get_const env path n to construct a 
const_descr and then wraps it in an id using location l and instantiations inst.

val get_const_id : 
  Typed_ast.env -> 
  Ast.l -> 
  string -> 
  Types.t list -> Types.const_descr_ref Types.id * Typed_ast.const_descr 
  get_const_id env l label inst uses strings_get_const_id with an indirection to look 
  up a constant for a given label.

val strings_mk_const_exp : 
  Typed_ast.env -> 
  Ast.l -> string list -> string -> Types.t list -> Typed_ast.exp 
  strings_mk_const_exp uses get_const_id to construct a constant expression.

val mk_const_exp : 
  Typed_ast.env -> Ast.l -> string -> Types.t list -> Typed_ast.exp 
  mk_const_exp uses strings_mk_const_exp with an indirection through a label.

val dest_field_types : 
  Ast.l -> 
  Typed_ast.env -> 
  Types.const_descr_ref -> Types.t * Path.t * Typed_ast.const_descr 
  dest_field_types l env f looks up the types of the field f in environment env. It first 
  gets the description f_descr of the field f in env. It then looks up the type of f. For fields, 
  this type is always of the form field_type -> (free_vars) rec_ty_path. 
  dest_field_types checks that free_vars corresponds with the free typevariables of 
  f_descr. If the type of f is not of the described from, or if free_vars does not correspond, 
  the constant did not describe a proper field. In this case, dest_field_types fails. 
  Otherwise, it returns (field_type, rec_ty_path, f_descr).

val get_field_type_descr : 
  Ast.l -> Typed_ast.env -> Types.const_descr_ref -> Types.type_descr 
  get_field_type_descr l env f uses dest_field_types l env f to get the base type of 
  the field f. It then looks up the description of this type in the environment.

val get_field_all_fields : 
  Ast.l -> Typed_ast.env -> Types.const_descr_ref -> Types.const_descr_ref list 
  get_field_all_fields l env f uses get_field_type_descr l env f to look up the type 
  description of the record type of the field f. It then returns a list of all the other fields of 
  this record.

val lookup_class_descr : 
  Ast.l -> Typed_ast.env -> Path.t -> Types.class_descr
lookup_class_descr l env c_path looks up the description of type-class **c_path** in environment **env**. If **c_path** is no valid type-class, an exception is raised.

```ocaml
def lookup_field_for_class_method:
    Ast.l -> Types.class_descr -> Types.const_descr_ref -> Types.const_descr_ref
lookup_field_for_class_method l cd method_ref looks up the field reference corresponding to the method identified by **method_ref** in the description **cd** of a type class. If the reference does not point to a method of this type-class, an exception is raised.
```

```ocaml
def lookup_inst_method_for_class_method:
    Ast.l -> Types.instance -> Types.const_descr_ref -> Types.const_descr_ref
lookup_inst_method_for_class_method l i method_ref looks up the instance method reference corresponding to the method identified by **method_ref** in the instance **i**. If the reference does not point to a method of this instance, an exception is raised.
```

```ocaml
def class_descr_get_dict_type:
    Types.class_descr -> Types.t -> Types.t
class_descr_get_dict_type cd arg generates the type of the dictionary for the class and argument type.
```

```ocaml
def class_all_methods_inlined_for_target:
    Ast.l -> Typed_ast.env -> Target.target -> Path.t -> bool
class_all_methods_inlined_for_target l env targ class_path checks, wether all methods of **class_path** are inlined for target **targ**.
```

```ocaml
def update_const_descr:
    Ast.l ->
    (Typed_ast.const_descr -> Typed_ast.const_descr) ->
    Types.const_descr_ref -> Typed_ast.env -> Typed_ast.env
update_const_descr l up c env updates the description of the constant **c** in environment **env** using the function **up**.
```

```ocaml
def c_env_store:
    Typed_ast.c_env ->
    Typed_ast.c_env -> Types.const_descr_ref
    c_env_store c_env c_d stores the description **c_d** environment **c_env**. Thereby, a new unique reference is generated and returned along with the modified environment.
```

```ocaml
def c_env_save:
    Typed_ast.c_env ->
    Types.const_descr_ref option ->
    Typed_ast.const_descr -> Typed_ast.c_env -> Types.const_descr_ref
    c_env_save c_env option c_d updates the description of the constant **c_d** in environment **c_env** using the function **option**.
```

78
c_env_save c_env c_ref_opt c_d is a combination of c_env_update and c_env_store. If c_ref_opt is given, c_env_update is called, otherwise c_env_store.

34.3 target-representations

val const_target_rep_to_loc : Typed.ast.const_target_rep -> Ast.l
cost_target_rep_to_loc rep returns the location, at which rep is defined.

val const_target_rep_allow_override : Typed.ast.const_target_rep -> bool
cost_target_rep_allow_override rep returns whether this representation can be redefined. Only auto-generated target-reps should be redefinable by the user.

val type_target_rep_to_loc : Types.type_target_rep -> Ast.l
type_target_rep_to_loc rep returns the location, at which rep is defined.

val type_target_rep_allow_override : Types.type_target_rep -> bool
type_target_rep_allow_override rep returns whether this representation can be redefined. Only auto-generated target-reps should be redefinable by the user.

val constant_descr_to_name :
  Target.target -> Typed.ast.const_descr -> bool * Name.t * Name.t option
constant_descr_to_name targ cd looks up the representation for target targ in the constant description cd. It returns a tuple (n_is_shown, n, n_ascii). The name n is the name of the constant for this target, n_ascii an optional ascii alternative. n_is_shown indicates, whether this name is actually printed. Special representations or inline representation might have a name, that is not used for the output.

val const_descr_ref_to_ascii_name :
  Typed.ast.c_env -> Types.const_descr_ref -> Name.t
const_descr_ref_to_ascii_name env c tries to find a simple identifier for constant c. The exact identifier does not matter, but should somehow be familiar to the user. It looks up the constant names, ascii-representations and renamings for various backends. If everything fails, it just makes a name up.

val type_descr_to_name :
  Target.target -> Path.t -> Types.type_descr -> Name.t
type_descr_to_name targ ty td looks up the representation for target targ in the type description td. Since in constrast to constant-description, type-descriptions don't contain the full type-name, but only renamings, the orginal type-name is passed as argument ty. It is assumed that td really belongs to ty.

val constant_descr_rename :
  Target.non_ident_target ->
Name.t ->
Ast.l ->
Typed.ast.const_descr -> Typed.ast.const_descr * (Ast.l * Name.t) option
const_descr_rename targ n' l' cd looks up the representation for target targ in the constant description cd. It then updates this description by renaming to the new name n' and new location l'. The updated description is returned along with information of where the constant was last renamed and to which name.

val mod_target_rep_rename :
  Target.non_ident_target ->
string ->
Name.t ->
Ast.l ->
Typed.ast.mod_target_rep Target.Targetmap.t ->
Typed.ast.mod_target_rep Target.Targetmap.t
  mod_descr_rename targ mod_name n' l' md updates the representation for target targ in the module description md by renaming to the new name n' and new location l'. In case a target representation was already present, a type-check error is raised.

val type_descr_rename :
  Target.non_ident_target ->
 Name.t ->
Ast.l -> Types.type_descr -> Types.type_descr * (Ast.l * Name.t) option
  type_descr_rename targ n' l' td looks up the representation for target targ in the type description td. It then updates this description by renaming to the new name n' and new location l'. The updated description is returned along with information of where the type was last renamed and to which name.

val type_defs_rename_type :
  Ast.l ->
Types.type_defs ->
Path.t -> Target.non_ident_target -> Name.t -> Types.type_defs
  type_def_rename_type l d p t n renames the type with path p in the defs d to the name n for target t. Renaming means that the module structure is kept. Only the name is changed.

val const_descr_has_target_rep :
  Target.target -> Typed.ast.const_descr -> bool
  const_descr_has_target_rep targ d checks whether the description d contains a target-representation for target targ.

34.4 Constructing, checking and destructing expressions

val mk_name_lskips_annot :
  Ast.l -> Name.lskips_t -> Types.t -> Typed.ast.name_lskips_annot
  mk_name_lskips_annot creates an annotated name

val dest_var_exp : Typed.ast.exp -> Name.t option
  Destructor for variable expressions
val is_var_exp : Typed_ast.exp -> bool
    is_var_exp e checks whether e is a variable expression

val dest_tup_exp : int option -> Typed_ast.exp -> Typed_ast.exp list option
    Destructor for tuple expressions. Similar to pattern destructors for tuples an optional
    argument to check the number of elements of the tuple.

val is_tup_exp : int option -> Typed_ast.exp -> bool
    is_tup_exp s_opt e checks whether e is a tuple of size s_opt.

val is_var_tup_exp : Typed_ast.exp -> bool
    is_var_tup_exp e checks, whether e is an expression consisting only of variables and tuples.
    I.e. simple variable expressions, tuples containing only variables and tuples containing other
    variable-tuples are accepted.

val mk_tf_exp : bool -> Typed_ast.exp
    mk_tf_exp creates true and false expressions.

val dest_tf_exp : Typed_ast.exp -> bool option
    dest_tf_exp destructs true and false expressions.

val is_tf_exp : bool -> Typed_ast.exp -> bool
    is_tf_exp v e checks whether e is a true or false expression.

val dest_const_exp : Typed_ast.exp -> Types.const_descr_ref Types.id option
    Destructor for constants expressions

val is_const_exp : Typed_ast.exp -> bool
    is_const_exp e checks whether e is a constant expression

val dest_num_exp : Typed_ast.exp -> int option
    dest_num_exp e destructs a number literal expression.

val is_num_exp : Typed_ast.exp -> bool
    is_num_exp checks whether e is a number literal expression.

val mk_num_exp : Types.t -> int -> Typed_ast.exp
    mk_num_exp creates a number literal expression.

val is_empty_backend_exp : Typed_ast.exp -> bool
    is_empty_backend_exp checks whether the expression is ``

val mk_eq_exp :
    Typed_ast.env -> Typed_ast.exp -> Typed_ast.exp -> Typed_ast.exp
mk_eq_exp env e1 e2 constructs the expression e1 = e2. The environment env is needed to lookup the equality constant.

val mk_and_exp :
Typed_ast.env -> Typed_ast.exp -> Typed_ast.exp -> Typed_ast.exp
mk_and_exp env e1 e2 constructs the expression e1 && e2. The environment env is needed to lookup the conjunction constant.

val mk_and_exps : Typed_ast.env -> Typed_ast.exp list -> Typed_ast.exp
mk_and_exps env es constructs the conjunction of all expressions in es. The environment env is needed to lookup the conjunction constant.

val mk_le_exp :
Typed_ast.env -> Typed_ast.exp -> Typed_ast.exp -> Typed_ast.exp
mk_le_exp env e1 e2 constructs the expression e1 <= e2. The environment env is needed to lookup the less-equal constant.

val mk_sub_exp :
Typed_ast.env -> Typed_ast.exp -> Typed_ast.exp -> Typed_ast.exp
mk_sub_exp env e1 e2 constructs the expression e1 - e2. The environment env is needed to lookup the subtraction constant.

val mk_from_list_exp : Typed_ast.env -> Typed_ast.exp -> Typed_ast.exp
mk_from_list_exp env e constructs the expression Set.from_list e. The environment env is needed to lookup the from-list constant.

val mk_cross_exp :
Typed_ast.env -> Typed_ast.exp -> Typed_ast.exp -> Typed_ast.exp
mk_cross_exp env e1 e2 constructs the expression cross e1 e2. The environment env is needed to lookup the cross constant.

val mk_set_sigma_exp :
Typed_ast.env -> Typed_ast.exp -> Typed_ast.exp -> Typed_ast.exp
mk_set_sigma_exp env e1 e2 constructs the expression set_sigma e1 e2. The environment env is needed to lookup the sigma constant.

val mk_set_filter_exp :
Typed_ast.env -> Typed_ast.exp -> Typed_ast.exp -> Typed_ast.exp
mk_set_filter_exp env e_P e_s constructs the expression Set.filter e_P e_s. The environment env is needed to lookup the constant.

val mk_set_image_exp :
Typed_ast.env -> Typed_ast.exp -> Typed_ast.exp -> Typed_ast.exp
mk_set_image_exp env e_f e_s constructs the expression Set.image e_f e_s. The environment env is needed to lookup the constant.
val mk_fun_exp : Typed_ast.pat list -> Typed_ast.exp -> Typed_ast.exp
    mk_fun_exp [p1, ..., pn] e constructs the expression fun p1 ... pn -> e.

val mk_opt_fun_exp : Typed_ast.pat list -> Typed_ast.exp -> Typed_ast.exp
    mk_opt_fun_exp pL e returns mk_fun_exp pL e if pL is not empty and e otherwise.

val mk_app_exp : Ast.l -> Types.type_defs -> Typed_ast.exp -> Typed_ast.exp
    mk_app_exp d e1 e2 constructs the expression e1 e2. The type definitions d are needed for typechecking.

val mk_list_app_exp : Ast.l -> Types.type_defs -> Typed_ast.exp -> Typed_ast.exp list -> Typed_ast.exp
    mk_list_app_exp d f [a1 ... an] constructs the expression f a1 ... an by repeatedly calling mk_app_exp.

val mk_eta_expansion_exp : Types.type_defs -> Name.t list -> Typed_ast.exp -> Typed_ast.exp
    mk_eta_expansion_exp d vars e for variables vars = [x1, ..., xn] tries to build the expression fun x1 ... xn -> (e x1 ... xn). The variable names might be changed to ensure that they are distinct to each other and all variables already present in e.

val mk_paren_exp : Typed_ast.exp -> Typed_ast.exp
    mk_paren_exp e adds parenthesis around expression e. Standard whitespaces are applied. This means that whitespace (except comments) are deleted before expression e.

val mk_opt_paren_exp : Typed_ast.exp -> Typed_ast.exp
    mk_opt_paren_exp e adds parenthesis around expression e if it seems sensible. For parenthesis, variable expressions and tuples, the parenthesis are skipped, though.

val may_need_paren : Typed_ast.exp -> bool
    may_need_paren e checks, whether e might need parenthesis. If returns, whether mk_opt_paren_exp e would modify the expression.

val mk_case_exp : bool -> Ast.l -> Typed_ast.exp -> (Typed_ast.pat * Typed_ast.exp) list -> Types.t -> Typed_ast.exp
    mk_case_exp final l e rows ty constructs a case (match) expression. In contrast to Typed_ast.mk_case it uses standard spacing and adds parenthesis.

val mk_let_exp : Ast.l -> Name.t * Typed_ast.exp -> Typed_ast.exp -> Typed_ast.exp
    mk_let_exp l -> Name.t * Typed_ast.exp -> Typed_ast.exp -> Typed_ast.exp
mk_let_exp 1 (n, e1) e2 constructs the expression let n = e1 in e2 using default spacing.

val mk_if_exp : Ast.l -> Typed_ast.exp -> Typed_ast.exp -> Typed_ast.exp -> Typed_ast.exp
mk_if_exp l c e_t e_f constructs the expression if c then e_t else e_f using default spacing.

val mk_undefined_exp : Ast.l -> string -> Types.t -> Typed_ast.exp
mk_undefined_exp l m ty constructs an undefined expression of type ty with message m.

val mk_dummy_exp : Types.t -> Typed_ast.exp
mk_dummy_exp ty constructs a dummy expression of type ty. This is an expression that should never be looked at. It is only guaranteed to be an expression of this type.

val dest_app_exp : Typed_ast.exp -> (Typed_ast.exp * Typed_ast.exp) option
dest_app_exp e tries to destruct an function application expression e.

val strip_app_exp : Typed_ast.exp -> Typed_ast.exp * Typed_ast.exp list
strip_app_exp e tries to destruct multiple function applications. It returns a pair (base_fun, arg_list) such that e is of the form base_fun arg_list_1 ... arg_list_n. If e is not a function application expression, the list arg_list is empty.

val dest_infix_exp :
Typed_ast.exp -> (Typed_ast.exp * Typed_ast.exp * Typed_ast.exp) option
dest_infix_exp e tries to destruct an infix expression e. If e is of the form l infix_op r then Some (l, infix_op, r) is returned, otherwise None.

val is_infix_exp : Typed_ast.exp -> bool
is_infix_exp e checks whether e is an infix operation

val strip_infix_exp : Typed_ast.exp -> Typed_ast.exp * Typed_ast.exp list
strip_infix_exp e is similar to dest_infix_exp, but returns the result in the same way as strip_app_exp. If e is of the form l infix_op r then (infix_op, [l;r]) is returned, otherwise (e, []).

val strip_app_infix_exp :
Typed_ast.exp -> Typed_ast.exp * Typed_ast.exp list * bool
strip_app_infix_exp e is a combination of strip_infix_exp and strip_app_exp. The additional boolean result states, whether e is an infix operation. If e is an infix operation strip_infix_exp is called and the additional boolean result is true. Otherwise strip_app_exp is called and the result is set to false.
34.5 Constructing, checking and destructing definitions

val is_type_def_abbrev : Typed_ast.def -> bool

is_type_def_abbrev d checks whether the definition d is a type-abbreviation definition.

val is_type_def_record : Typed_ast.def -> bool

is_type_def_abbrev d checks whether the definition d is a definition of a record_type.

34.6 Collecting information about uses constants, types, modules ...

type used_entities = {
    used_consts : Types.const_descr_ref list ;
    used_consts_set : Types.Cdset.t ;
    used_types : Path.t list ;
    used_types_set : Types.Pset.t ;
    used_modules : Path.t list ;
    used_modules_set : Types.Pset.t ;
    used_tnvars : Types.TNset.t ;
}

The type used_entities collects lists of used constant references, modules and types of some expression, definition, pattern ... used_entities is using lists, because the order in which entities occur might be important for renaming. However, these lists should not contain duplicates.

val empty_used_entities : used_entities

An empty collection of entities

val add_exp_entities : used_entities ->
    Typed_ast.exp -> used_entities

val add_def_aux_entities :
    Target.target ->
    bool ->
    used_entities ->
    Typed_ast.def_aux -> used_entities

add_def_aux_entities targ only_new ue def adds all the modules, types, constants ... used by definition def for target targ to ue. If the flag only_new is set, only the newly defined are added. Notice, that the identity backend won't throw parts of modules away. Therefore the result for the identity backend is the union of the results for all other backends.

val add_def_entities :
    Target.target ->
    bool ->
    used_entities ->
    Typed_ast.def -> used_entities
add_def_entities is called add_def_aux_entities after extracting the appropriate def_aux.

val get_checked_modules_entities : Target.target -> bool -> Typed_ast.checked_module list -> used_entities

get_checked_module_entities targ only_new ml gets all the modules, types, constants ... used by modules ml for target targ. If the flag only_new is set, only the newly defined are returned. Notice, that the identity backend won’t throw parts of modules away. Therefore the result for the identity backend is the union of the results for all other backends.

34.7 Miscellaneous

val remove_init_ws : Ast.lex_skips -> Ast.lex_skips * Ast.lex_skips

remove_init_ws should be used with function like Typed_ast.alter_init_lskips. It removes whitespace expect comments.

val drop_init_ws : Ast.lex_skips -> Ast.lex_skips * Ast.lex_skips

drop_init_ws should be used with function like Typed_ast.alter_init_lskips. It removes whitespace including comments.

val space_init_ws : Ast.lex_skips -> Ast.lex_skips * Ast.lex_skips

space_init_ws should be used with function like Typed_ast.alter_init_lskips. It replaces whitespace including comments with a single space.

val space_com_init_ws : Ast.lex_skips -> Ast.lex_skips * Ast.lex_skips

space_com_init_ws should be used with function like Typed_ast.alter_init_lskips. It replaces whitespace except comments with a single space.

val strip_paren_typ_exp : Typed_ast.exp -> Typed_ast.exp

strip_paren_typ_exp e strips parenthesis and type-annotations form expression e. Warning: This might delete white-space!

val is_recursive_def : Typed_ast.def_aux -> bool * bool

is_recursive_def d checks whether d is recursive. It returns a pair of booleans (is_syntactic_rec, is_real_rec). The flag is_syntactic_rec states, whether the definition was made using the rec-keyword. The flag is_real_rec states, whether the function actually appears inside its own definition.

val try_termination_proof : Target.target -> Typed_ast.c_env -> Typed_ast.def_aux -> bool * bool * bool

try_termination_proof targ c_env d calls is_recursive_def d. It further checks, whether a termination proof for target targ should be tried by checking the termination settings of all involved constants. It returns a triple (is_syntactic_rec, is_real_rec, try_auto_termination).
val is_pp_loc : Ast.l -> bool
  is_pp_loc l checks whether l is of the form Ast.Trans (true, _, _). This means that
  the entity marked with l should be formatted using a pretty printer that calculates
  whitespaces new instead of using the ones provided by the user.

val is_pp_exp : Typed_ast.exp -> bool
val is_pp_def : Typed_ast.def -> bool
val val_def_get_name : Typed_ast.val_def -> Name.lskips_t option
  val_def_get_name d tries to extract the name of the defined function.

val val_def_get_class_constraints_no_target_rep : 
  Typed_ast.env -> Target.target -> Typed_ast.val_def -> (Path.t * Types.tnvar) list
  val_def_get_class_constraints_no_target_rep env targ vd collects the class
  constraints of all top-level function definitions in vd, which don’t have a target-specific
  representation for target targ. Warning: constraints may appear multiple times in the
  resulting list

val val_def_get_class_constraints : 
  Typed_ast.env -> Typed_ast.val_def -> (Path.t * Types.tnvar) list
  val_def_get_class_constraints env vd collects the class constraints of all top-level
  function definitions in vd. Warning: constraints may appear multiple times in the resulting
  list

val val_def_get_free_tnvars : 
  Typed_ast.env -> Typed_ast.val_def -> Types.TNset.t
  val_def_get_free_tnvars env vd returns the set of all free type-variables used by vd.

val env_tag_to_string : Typed_ast.env_tag -> string
  env_tag_to_string tag formats tag as a string. This functions should only be used for
  human-readable output in e.g. error-messages.

val constr_family_to_id : 
  Ast.l ->
  Typed_ast.env ->
  Types.t ->
  Types.constr_family_descr ->
  (Types.const_descr_ref Types.id list *
  (Types.t -> Types.const_descr_ref Types.id) option) option
  constr_family_to_id l env ty cf tries to instantiate the constructor family cf to be
  used on a match statement where the matched type is ty. If it succeeds the properly
  instantiated constructor ids + the instantiated case split function is returned. However,
  returning the case-split function is a bit complicated. It depends on the return type of
  match expression as well. Moreover, it might not be there at all, if the targets build-in
pattern matching should be used to construct one. Therefore, an optional function from a type (the return type) to an id is returned for the case-split function.

```ocaml
val check_constr_family : Ast.l -> Typed_ast.env -> Types.t -> Types.constr_family_descr -> unit

check_constr_family is similar to constr_family_to_id. It does not return the instantiations though, but produces a nicely formatted error, in case no such instantiations could be found.

val check_for_inline_cycles : Target.target -> Typed_ast.c_env -> unit

check_for_inline_cycles targ env checks whether any constant in env would be inlined (possible over several steps) onto itself. If this happens, an exception is thrown.
```

## 35 Module Types: Structural comparison of types, without expanding type abbreviations.

Probably better not to use. Consider using compare_expand instead.

```ocaml
type tnvar =
  | Ty of Tyvar.t
  | Nv of Nvar.t

val pp_tnvar : Format.formatter -> tnvar -> unit
val tnvar_to_rope : tnvar -> Ulib.Text.t
val tnvar_compare : tnvar -> tnvar -> int

module TNvar :
  sig
    type t = Types.tnvar
    val compare : t -> t -> int
    val pp : Format.formatter -> t -> unit
  end

module Pfmap :
  Finite_map.Fmap with type k = Path.t

module Pset :
  Set.S with type elt = Path.t

module TNfmap :
  Finite_map.Fmap with type k = TNvar.t

module TNset :
  sig
    include Set.S
    val pp : Format.formatter -> t -> unit
```

88
end

type t_uvar

type n_uvar

type t = {
mutable t : t_aux;
}

type t_aux =
| Tvar of Tyvar.t
| Tfn of t * t
| Ttup of t list
| Tapp of t list * Path.t
| Tbackend of t list * Path.t
| Tne of nexp
| Tuvar of t_uvar

type nexp = {
mutable nexp : nexp_aux;
}

type nexp_aux =
| Nvar of Nvar.t
| Nconst of int
| Nadd of nexp * nexp
| Nmult of nexp * nexp
| Nneg of nexp
| Nuvar of n_uvar

type range =
| LtEq of Ast.l * nexp
| Eq of Ast.l * nexp
| GtEq of Ast.l * nexp
val range_with : range -> nexp -> range
val range_of_n : range -> nexp
val mk_gt_than : Ast.l -> nexp -> nexp -> range
val mk_eq_to : Ast.l -> nexp -> nexp -> range
val compare : t -> t -> int

Structural comparison of types, without expanding type abbreviations. Probably better not
to use. Consider using compare_expand instead.

val multi_fun : t list -> t -> t
val type_subst : t TNfmap.t -> t -> t
val nexp_subst : t TNfmap.t -> nexp -> nexp
val free_vars : t -> TNset.t
val is_var_type : t -> bool
val is_instance_type : t -> bool
is the type ok to be used in a non-default type-class instantiation?

```ocaml
cval tnvar_to_name : tnvar -> Name.t
cval tnvar_to_type : tnvar -> t
cval tnvar_split : tnvar list -> tnvar list * tnvar list
type const_descr_ref

A reference to a constant description. These constant description references are used by

typed.ast. This module also contains the appropriate mapping functionality to constant
descriptions. However, the references need to be defined here, because types need
information about associated constants. Record types need a list of all their field constants.
Moreover, every type can contain a list of constructor descriptions.

val string_of_const_descr_ref : const_descr_ref -> string

string_of_const_descr_ref formats a reference in a human readable form. No other
guarantees are given. This function should only be used for debugging and reporting internal
errors. Its implementation can change at any point to something completely different and
should not be relied on.

module Cdmap :
Finite_map.Fmap with type k = const_descr_ref
module Cdset :
Set.S with type elt = const_descr_ref
type 'a cdmap

cdmap is a type for maps of const_descr_ref. In contrast to finite maps represented by
module Cdmap, the keys might be autogenerated.

val cdmap_empty : unit -> 'a cdmap

Constructs an empty cdmap

val cdmap_lookup : 'a cdmap -> const_descr_ref -> 'a option

cdmap_lookup m r looks up the reference r in map m

val cdmap_update : 'a cdmap -> const_descr_ref -> 'a -> 'a cdmap

cdmap_update m r v updates map m at reference r with value v.

val cdmap_insert : 'a cdmap -> 'a -> 'a cdmap * const_descr_ref

cdmap_insert m v inserts value v into m. A fresh (not occurring in m) reference is generated
for v and returned together with the modified map.

val cdmap_domain : 'a cdmap -> const_descr_ref list

cdmap_domain m returns the list of all constant description references in the map

val nil_const_descr_ref : const_descr_ref
```

90
nil_const_descr_ref is a nil reference, i.e. a reference that will never be bound by any cdmap.

val is_nil_const_descr_ref : const_descr_ref -> bool
is_nil_constdescr_ref r checks whether r is the nil reference.

type ('a, 'b) annot = {
  term : 'a ;
  locn : Ast.l ;
  typ : t ;
  rest : 'b ;
}

val annot_to_typ : ('a, 'b) annot -> t

type ident_option =
  | Id_none of Ast.lex_skips
  | Id_some of Ident.t

type 'a id = {
  id_path : ident_option ;
  id_locn : Ast.l ;
  descr : 'a ;
  instantiation : t list ;
}

represents a usage of an 'a (usually in constr_descr, field_descr, const_descr)

type src_t = (src_t_aux, unit) annot

type src_t_aux =
  | Typ_wild of Ast.lex_skips
  | Typ_var of Ast.lex_skips * Tyvar.t
  | Typ_len of src_nexp
  | Typ_fn of src_t * Ast.lex_skips * src_t
  | Typ_tup of (src_t, Ast.lex_skips) Seplist.t
  | Typ_app of Path.t id * src_t list
  | Typ_backend of Path.t id * src_t list
    a backend type that should be used literally
  | Typ_paren of Ast.lex_skips * src_t * Ast.lex_skips

type src_nexp = {
type src_nexp_aux =
  | Nexp_var of Ast.lex_skips * Nvar.t
  | Nexp_const of Ast.lex_skips * int
  | Nexp_mult of src_nexp * Ast.lex_skips * src_nexp
  | Nexp_add of src_nexp * Ast.lex_skips * src_nexp
  | Nexp_paren of Ast.lex_skips * src_nexp * Ast.lex_skips

val src_t_to_t : src_t -> t
val src_type_subst : src_t TNfmap.t -> src_t -> src_t
val id_alter_init_lskips : (Ast.lex_skips -> Ast.lex_skips * Ast.lex_skips) ->
  'a id -> 'a id * Ast.lex_skips
val typ_alter_init_lskips : (Ast.lex_skips -> Ast.lex_skips * Ast.lex_skips) ->
  src_t -> src_t * Ast.lex_skips
val nexp_alter_init_lskips : (Ast.lex_skips -> Ast.lex_skips * Ast.lex_skips) ->
  src_nexp -> src_nexp * Ast.lex_skips

type constr_family_descr = {
  constr_list : const_descr_ref list ;
  constr_exhaustive : bool ;
  constr_case_fun : const_descr_ref option ;
  constr_default : bool ;
  constr_targets : Target.Targetset.t ;
}

type type_target_rep =
  | TYR_simple of Ast.l * bool * Ident.t
  | TYR_subst of Ast.l * bool * tnvar list * src_t

  the target representation of a type

type type_descr = {
  type_tparams : tnvar list ;
    a list of type and length parameters
  type_abbrev : t option ;
    if it is an abbreviation, the type it abbreviates
  type_varname_regexp : string option ;
    an optional regular expression that variable names that have the type must match
  type_fields : const_descr_ref list option ;
    if it is a record type, the list of fields

92
type_signature : constr_family_descr list ;
    the constructors of this type

type_rename : (Ast.l * Name.t) Target.Targetmap.t ;
    target representation of the type

type_target_rep : type_target_rep Target.Targetmap.t ;
    target representation of the type

}

  a type description *


type class_descr = {
  class_tparam : tnvar ;
    the type parameter of the type class
  class_record : Path.t ;
    for dictionary style passing a corresponding record is defined, this is its path
  class_methods : (const_descr_ref * const_descr_ref) list ;
    the methods of the class. For each method there is a corresponding record field. Details like the
    names and types can be looked up in the environment.
  class_rename : (Ast.l * Name.t) Target.Targetmap.t ;
  class_target_rep : type_target_rep Target.Targetmap.t ;
  class_is_inline : bool ;
}


type tc_def =
  | Tc_type of type_descr
  | Tc_class of class_descr

type type_defs = tc_def Pfmap.t

val type_defs_update_tc_type :
  Ast.l ->
  type_defs ->
  Path.t -> (type descr -> type descr option) -> type_defs
    type_defs_update_tc_type l d p up updates the description of type p in d using the
    function up. If there is no type p in d or if up returns None, an exception is raised.

val type_defs_update_tc_class :
  Ast.l ->
  type_defs ->
  Path.t -> (class_descr -> class_descr option) -> type_defs
    type_defs_update_tc_class l d p up updates the description of type p in d using the
    function up. If there is no type p in d or if up returns None, an exception is raised.
val type_defs_update_fields : 
  Ast.l ->
  type_defs -> Path.t -> const_descr_ref list -> type_defs
  type_defs_update_fields l d p fl updates the fields of type p in d.

val type_defs_add_constr_family :
  Ast.l ->
  type_defs -> Path.t -> constr_family_descr -> type_defs

val type_defs_get_constr_families :
  Ast.l ->
  type_defs -> Target.target ->
  t -> const_descr_ref -> constr_family_descr list
  type_defs_get_constr_families l d targ t c gets all constructor family descriptions for
  type t for target targ in type environment d, which contain the constant c.

val type_defs_lookup_typ : Ast.l -> type_defs -> t -> type_descr option
  type_defs_lookup_typ l d t looks up the description of type t in defs d.

val type_defs_lookup : Ast.l -> type_defs -> Path.t -> type_descr
  type_defs_lookup l d p looks up the description of type with path p in defs d.

val type_defs_update : type_defs -> Path.t -> type_descr -> type_defs
  type_defs_update d p td updates the description of type with path p in defs d with td.

val mk_tc_type_abbrev : tnvar list -> t -> tc_def
  Generates a type abbreviation

val mk_tc_type : tnvar list -> string option -> tc_def
  mk_tc_type vars reg_exp_opt generates a simple description of a type, which uses the
  type arguments vars and the reg_exp_opt for restricting the names of variables of this type.

val match_types : t -> t -> t TNfmap.t option
  match_types t_pat t tries to match type t_pat against type t. If it succeeds, it returns a
  substitution sub that applied to t_pat returns t. This function is rather simple. It does not
  use type synonyms or other fancy features.

type instance = {
  inst_l : Ast.l ;
  The location, the instance was declared

  inst_is_default : bool ;
  Is it a fallback / default instance or a real one ?

  inst_binding : Path.t ;
}
The path of the instance

```haskell
inst_class : Path.t ;

The type class, that is instantiated

inst_type : t ;

The type, the type-class is instantiated with

inst_tyvars : tnvar list ;

The free type variables of this instance

inst_constraints : (Path.t * tnvar) list ;

Type class constraints on the free type variables of the instance

inst_methods : (const_descr_ref * const_descr_ref) list ;

The methods of this instance. Since each instance method corresponds to one class
method it instantiates, the methods are given as a list of pairs (class_method_ref,
instance_method_ref).

inst_dict : const_descr_ref ;

a dictionary for the instance
```

}  

an instance of a type class

type typ_constraints =

| Tconstraints of TNset.t * (Path.t * tnvar) list * range list

type defns option -> t -> t

val head_norm : type_defs -> t -> t

dest_fn_type : type_defs option -> t -> (t * t) option

dest_fn_type d_opt t tries to destruct a function type t. Before the destruction,
head_norm d t is applied, if d_opt is of the form Some d. If the result is a function type, t1
-> t2, the Some (t1, t2) is returned. Otherwise the result is None.

val strip_fn_type : type_defs option -> t -> t list * t

strip_fn_type d t tries to destruct a function type t by applying dest_fn repeatedly.

val check_equal : type_defs -> t -> t -> bool

check_equal d t1 t2 checks whether t1 and t2 are equal in type environment d. It
expands the type to perform this check. Therefore, it is more reliable than compare t1 t2 =
0, which only performs a structural check, but does not unfold type definitions.

val assert_equal : Ast.l -> string -> type_defs -> t -> t -> unit

assert_equal l m d t1 t2 performs the same check as check_equal d t1 t2. However,
while check_equal returns whether the types are equal, assert_equal raises a type-exception
in case they are not. l and m are used for printing this exception.

val compare_expand : type_defs -> t -> t -> int
compare_expand d t1 t2 is similar check_equal d t1 t2. Instead of just checking for equality, it compare the values though. During this comparison, type abbreviations are unfolded. Therefore, it is in general preferable to the very similar method compare, which perform comparisons without unfolding.

type instance_ref
   A reference to an instance.

val string_of_instance_ref : instance_ref -> string
   string_of_instance_ref formats a reference in a human readable form. No other guarantees are given. This function should only be used for debugging and reporting internal errors. Its implementation can change at any point to something completely different and should not be relied on.

type i_env
   an instance environment carries information about all defined instances

val empty_i_env : i_env
   an empty instance environment

val i_env_add : i_env -> instance -> i_env * instance_ref
   i_env_add i_env i adds an additional instance i to the instance environment. It returns the modified environment as well as the reference of the added instance.

val i_env_lookup : Ast.l -> i_env -> instance_ref -> instance
   i_env_lookup l i_env ref looks up the reference in environment i_env. If this reference is not present, an exception is raised.

val get_matching_instance :
   type_def ->
   Path.t * t ->
   i_env -> (instance * t TNfmap.t) option
   get_matching_instance type_env (class, ty) i_env searches for an instantiation of type class class instantiated with type ty in the type environment i_env. The type environment type_env is necessary to match ty against other instantiations of class. An instance can itself have free type variables. If a matching instance is found, it is returned to together with the substitution, which needs to be applied to the free type variables of the instance in order to match type t exactly. The typevariables of an instances might have attached type constraints. It is not (!) checked, that the found substitution satisfies these constraints. However, they are taken into account to rule out impossible instances, if there are multiple options.

val nexp_from_list : nexp list -> nexp

module type Global_defs =
   sig
      "
val d : Types.type_defns
val i : Types.i_env
end

module Constraint :
  functor (T : Global_defs) -> sig
    val new_type : unit -> Types.t
    val new_nexp : unit -> Types.nexp
    val equate_types : Ast.l -> string -> Types.t -> Types.t -> unit
    val in_range : Ast.l -> Types.nexp -> Types.nexp -> unit
    val add_constraint : Path.t -> Types.t -> unit
    val add_length_constraint : Types.range -> unit
    val add_tyvar : Tyvar.t -> unit
    val add_nvar : Nvar.t -> unit
    val inst_leftover_uvars : Ast.l -> Types.typ_constraints
    val check_numeric_constraint_implication :
      Ast.l -> Types.range -> Types.range list -> unit
  end

val pp_type : Format.formatter -> t -> unit
val pp_nexp : Format.formatter -> nexp -> unit
val pp_range : Format.formatter -> range -> unit
val pp_class_constraint : Format.formatter -> Path.t * tnvar -> unit
val pp_instance : Format.formatter -> instance -> unit
val pp_typschm :
  Format.formatter ->
  tnvar list -> (Path.t * tnvar) list -> t -> unit
val t_to_string : t -> string
val print_debug_typ_raw : string -> t list -> unit
  print_debug_typ_raw s [ty0, ..., tn] prints a debug message s t0, ..., tn using
  Reporting_basic.print_debug.
val t_to_var_name : t -> Name.t

36  Module Tyvar : type of internal(?) type variables

type t
val compare : t -> t -> int
val pp : Format.formatter -> t -> unit
val nth : int -> t
val from_rope : Ulib.Text.t -> t
val to_rope : t -> Ulib.Text.t

37 Module Util: Mixed useful things

module Duplicate:
  functor (S : Set.S) -> sig
    type dups =
      | No_dups of S.t
      | Has_dups of S.elt
    val duplicates : S.elt list -> dups
  end

val remove_duplicates : 'a list -> 'a list
  remove_duplicates l removes duplicate elements from the list l. The elements keep their original order. The first occurrence of an element is kept, all others deleted.

val remove_duplicates_gen : ('a -> 'a -> bool) -> 'a list -> 'a list
  remove_duplicates_gen p l removes duplicate elements from the list l. It is a generalised version of remove_duplicates where the equality check is performed by p.

val get_duplicates : 'a list -> 'a list
  get_duplicates l returns the elements that appear multiple times in the list l.

val get_duplicates_gen : ('a -> 'a -> bool) -> 'a list -> 'a list
  get_duplicates_gen p l returns the elements that appear multiple times in the list l. It is a generalised version of get_duplicates where the equality check is performed by p.

37.1 Option Functions

val option_map : ('a -> 'b) -> 'a option -> 'b option
  option_map f None returns None, whereas option_map f (Some x) returns Some (f x).

val option_cases : 'a option -> ('a -> 'b) -> (unit -> 'b) -> 'b
  option_cases None f_s f_n returns f_n, whereas option_cases (Some x) f_s f_n returns f_s x.

val option_bind : ('a -> 'b option) -> 'a option -> 'b option
  option_bind f None returns None, whereas option_bind f (Some x) returns f x.

val option_default : 'a -> 'a option -> 'a

98
option_default d None returns the default value d, whereas option_default d (Some x) returns x.

val option_default_map : 'a option -> 'b -> ('a -> 'b) -> 'b
option_default_map v d f is short for option_default d (option_map f v). This means that option_default_map None d f returns d, whereas option_default_map (Some x) d f returns f x.

val option_get_exn : exn -> 'a option -> 'a
option_get_exn exn None throws the exception exn, whereas option_get_exn exn (Some x) returns x.

val changed2 : ('a -> 'b -> 'c) ->
(a -> 'a option) -> 'a -> (b -> 'b option) -> 'b -> 'c option
changed2 f g x h y applies g to x and h to y. If both function applications return None, then None is returned. Otherwise f is applied to the results. For this application of f, x is used in case g x returns None and similarly y in case h y returns None.

val option_repeat : ('a -> 'a option) -> 'a -> 'a option_repeat f x applies f to x till nothings changes any more. This means that if f x is None, x is returned. Otherwise option_repeat calls itself recursively on the result of f x.

37.2 List Functions

val list_index : ('a -> bool) -> 'a list -> int option
list_index p l returns the first index i such that the predicate p (l!i) holds. If no such i exists, None is returned.

val list_subset : 'a list -> 'a list -> bool
list_subset l1 l2 tests whether all elements of l1 also occur in l2.

val list_diff : 'a list -> 'a list -> 'a list
list_diff l1 l2 removes all elements from l1 that occur in l2.

val list_longer : int -> 'a list -> bool
list_longer n l checks whether the list l has more than n elements. It is equivalent to List.length l > n, but more efficient, as it aborts counting, when the limit is reached.

val list_null : 'a list -> bool
list_null l checks whether the list l is empty, i.e. if l = [] holds.

val option_first : ('a -> 'b option) -> 'a list -> 'b option
option_first f l searches for the first element x of l such that the f x is not None. If such an element exists, f x is returned, otherwise None.
val map_changed : ('a -> 'a option) -> 'a list -> 'a list option
    map_changed f l maps f over l. If for all elements of l the function f returns None, then
    map_changed f l returns None. Otherwise, it uses x for all elements, where f x returns
    None, and returns the resulting list.

val map_changed_default : 
    ('a -> 'b) -> ('a -> 'b option) -> 'a list -> 'b list option
    map_changed_default d f l maps f over l. If for all elements of l the function f returns
    None, then map_changed_default d f l returns None. Otherwise, it uses d x for all elements x, where
    f x returns None, and returns the resulting list.

val list_mapi : (int -> 'a -> 'b) -> 'a list -> 'b list
    list_mapi f l maps f over l. In contrast to the standard map function, f gets the current
    index of the list as an extra argument. Counting starts at 0.

val list_iter_sep : (unit -> unit) -> ('a -> unit) -> 'a list -> unit
    list_iter_sep sf f [a1; ...; an] applies function f in turn to a1; ...; an and calls sf
    () in between. It is equivalent to begin f a1; sf(); f a2; sf(); ...; f an; () end.

val intercalate : 'a -> 'a list -> 'a list
    intercalate sep as inserts sep between the elements of as, i.e. it returns a list of the form
    a1; sep; ... sep ; an.

val interleave : 'a list -> 'a list -> 'a list
    interleave l1 l2 interleaves lists l1 and l2, by alternatingly taking an element of l1 and
    l2 till one of the lists is empty. Then the remaining elements are added. The first element is
    from l1.

val replicate : int -> 'a -> 'a list
    replicate n e creates a list that contains n times the element e.

val map_filter : ('a -> 'b option) -> 'a list -> 'b list
    map_filter f l maps f over l and removes all entries x of l with f x = None.

val map_all : ('a -> 'b option) -> 'a list -> 'b list option
    map_all f l maps f over l. If at least one entry is None, None is returned. Otherwise, the
    Some function is removed from the list.

val list_to_front : int -> 'a list -> 'a list
    list_to_front i l resorts the list l by bringing the element at index i to the front.

val undo_list_to_front : int -> 'a list -> 'a list
    undo_list_to_front i l resorts the list l by moving the head element to index index i. It's
    the inverse of list_to_front i l.
val split_after : int -> 'a list -> 'a list * 'a list
    split_after n l splits the first n elements from list l, i.e. it returns two lists l1 and l2, with length l1 = n and l1 @ l2 = l. Fails if n is too small or large.

val list_firstn : int -> 'a list -> 'a list
    list_firstn n l gets the first n elements from list l, i.e. it does the same as fst (split_after n l). It fails if n is too small or large.

val list_dropn : int -> 'a list -> 'a list
    list_dropn n l drops the first n elements from list l, i.e. it does the same as snd (split_after n l). It fails if n is too small or large.

val list_dest_snoc : 'a list -> 'a list * 'a
    list_dest_snoc l splits the last entry off a list. This means that list_dest_snoc (l @ [x]) returns (l, x). It raises a Failure "list_dest_snoc" exception, if the list l is empty.

val list_pick : ('a -> bool) -> 'a list -> ('a * 'a list) option
    list_pick p l tries to pick the first element from l that satisfies predicate p. If such an element is found, it is returned together with the list l where this element has been removed.

val compare_list : ('a -> 'b -> int) -> 'a list -> 'b list -> int

37.3 Files

val copy_file : string -> string -> unit
    copy_file src dst copies file src to file dst. Only files are supported, no directories.

val move_file : string -> string -> unit
    move_file src dst moves file src to file dst. In contrast to Sys.rename no error is produced, if dst already exists. Moreover, in case Sys.rename fails for some reason (e.g. because it does not work over filesystem boundaries), copy_file and Sys.remove are used as fallback.

val same_content_files : string -> string -> bool
    same_content_files file1 file2 checks, whether the files file1 and file2 have the same content. If at least one of the files does not exist, false is returned. same_content_files throws an exception, if one of the files exists, but cannot be read.

val absolute_dir : string -> string option
    absolute_dir dir tries to get the absolute path name of directory dir. If this fails (usually, because dir does not exist), None is returned.

val dir_eq : string -> string -> bool
    dir_eq d1 d2 uses absolute_dir to check whether the two directories are equal.
37.4 Strings

val string_to_list : string -> char list
string_to_list 1 translates the string 1 to the list of its characters.

val string_for_all : (char -> bool) -> string -> bool
string_for_all p s checks whether all characters of s satisfy p.

val is_simple_ident_string : string -> bool
is_simple_ident_string s checks whether s is a "simple" identifier. The meaning of simple is fuzzy. Essentially it means that s can be used by all backends. Currently the restricting is that s is non-empty, starts with a letter and contains only letters, numbers and underscores.

val is_simple_char : char -> bool
is_simple_char c checks whether c is an easily printable character. Currently these are the characters which Isabelle's parser and pretty-printer supports. This decision was taken, because Isabelle is the most restrictive of our backend. It might be revised at any point. The user can rely on that is_simple_char only accepts chars that need no escaping for any backend. These are simple letters (A-Z, a-z), digits (0-9) and a few selected special chars (space, parenthesis, punctuation ... )

val string_split : char -> string -> string list * string
string_split c string splits the string into a list of strings on occurrences of the char c. The last remaining string is handed out separately. This encodes, that the resulting string list is never empty.

val uncapitalize_prefix : string -> string
uncapitalize_prefix n tries to uncapitalize the first few letters of n. In contrast to uncapitalize, it continues with the next letter, till a non-uppercase letter is found. The idea is to produce nicer looking names when uncapitalizing. Turning UTF8.lem into uTF8Script.sml for example is strange and utf8Script.sml looks nicer.

val string_map : (char -> char) -> string -> string
string_map f s maps f over all characters of a copy of s. It corresponds to String.map, which is unluckily only available for OCaml 4.

val message_singular_plural : string * string -> 'a list -> string
message_singular_plural (sing_message, multiple_message) l is used to determine whether the singular or plural form should be used in messages. If the list l contains no elements or exactly one element, sing_message is returned. Otherwise, i.e. for multiple elements, the result is multiple_message.

val fresh_string : string list -> string -> string
fresh_string forbidden generates a stateful function gen_fresh that generates fresh strings. gen_fresh s will return a string similar to s that has never been returned before and is not part of forbidden. By storing the result in internal state, it is ensured that the same result is never returned twice. This function is used for example to generate unique labels.

37.5 Useful Sets

module StringSet :
    Set.S with type elt = string
Sets of Integers

module IntSet :
    Set.S with type elt = int
module IntIntSet :
    Set.S with type elt = int * int
module ExtraSet :
    functor (S : Set.S) -> sig
        val add_list : S.t -> S.elt list -> S.t
Add a list of values to an existing set.
        val remove_list : S.t -> S.elt list -> S.t
Removes a list of values to an existing set.
        val from_list : S.elt list -> S.t
Construct a set from a list.
        val list_union : S.t list -> S.t
Builds the union of a list of sets
        val list_inter : S.t list -> S.t
Builds the intersection of a list of sets. If the list is empty, a match exception is thrown.
    end
Some useful extra functions for sets