ICS Manual (Version 1.0)

The ICS group

Computer Science Laboratory, SRI International
333 Ravenswood Avenue, Menlo Park, CA 94025, USA
ruess@csl.sri.com

Contents

1 Introduction

ICS (Integrated Canonizer and Solver) is a decision procedure developed at SRI International. It efficiently decides formulas in a useful combination of theories, and it provides an API that makes it suitable for use in applications with highly dynamic environments such as proof search or symbolic simulation.

The theory decided by ICS is a quantifier-free, first-order theory of equality with terms built from

- uninterpreted function symbols,
- arithmetic operators including inequalities and number predicates,
- tupling operators and tuple projection,
- S-expressions,
- array lookup and update,
- operators on fixed-sized bitvectors.

This theory is particularly interesting for many applications in the realm of software and hardware verification. Combinations of a multitude of datatypes occur naturally in system specifications and the use of uninterpreted function symbols has proven to be essential for many real-world verifications.

The core of ICS is a congruence closure procedure [RS01, ?] for the theory of equality and disequality with both uninterpreted and interpreted function symbols. The concepts of canonization and solving have been extended to include inequalities over linear arithmetic terms. The theory supported by ICS currently includes:

- The usual propositional constants `true`, `false`.
- Equality (`=`) and disequality (`/=`).
- Rational constants and the arithmetic operators `+`, `*`, `-`; multiplication is restricted to multiplication by constants. Arithmetic predicates include an integer test and the usual inequalities `<`, `<=`, `>`, `>=`.
- Lookup a`[`x`]` and update a`[`x`:=`t`]` operations for arrays a.
- Fixed-sized bitvectors including constants such as `0b101`, concatenation (`conc[3,4](`b_1`,`b_2`)`), extraction (b`[`i`:`j`]`), bit-wise operations like bit-wise conjunction (b_1 `&&` b_2), and built-in arithmetic relations such as `add(`b_1`,`b_2`,`b`)`. This latter constraint encodes the fact that the sum of the unsigned interpretations of b_1 and b_2 equals the unsigned interpretation of b. Fixed-sized bitvectors are decided using the techniques described in [MR98].

ICS is capable of deciding sequents such as

- `x+2 = y |- f(a[x:=3][y-2]) = f(y-x+1)`
- `f(y-1)-1 = y+1, f(x)+1 = x-1, x+1 = y |- false`
- `f(f(x)-f(y)) /= f(z), y <= x, y >= x+z, z >= 0 |- false`

These formulas contain uninterpreted function symbols such as `f` and interpreted symbols drawn from the theories of arithmetic and the functional arrays. The list of interpreted theories above is open-ended in the sense that new theories can be added to ICS as long as they are canonizable and algebraically solvable. The modular design of ICS—both the underlying algorithms and their implementation—supports such extensions.

One of the main problems in employing decision procedures effectively is due to the fact that verification conditions usually depend on large contexts. In addition, these contexts change frequently in applications such as symbolic simulation or backtracking proof search. Consequently, decision procedure systems that are effective in these domains must not only be able to build up contexts incrementally but they must also support efficiently switching between a multitude of contexts. ICS meets these criteria in that all of its main algorithm work incrementally and the data structures for representing contexts are persistent, that is, operations on data structures do not alter the previous values of data and *undo* operations are therefore basically for free.

ICS is implemented in Ocaml [Oca], which offers satisfactory run-time performance, efficient garbage collection, and interfaces well with other languages like C. The implementation of ICS is based on optimization techniques such as hash-consing [Fil00] and efficient data structures like Patricia trees [OG98] for representing sets and maps efficiently.

There is a well-defined API for manipulating ICS terms, asserting formulas to the current database, switching between databases, and functions for canonizing terms. This API is packaged as a C library, an Ocaml module, and a CommonLisp interface. The C library API, for example, has been used to connect ICS with PVS [ORS92], and both an interaction and a batch processing capability have been built using this API.

Using ICS as the underlying decision procedure of PVS, we experience speed-ups of several orders of magnitude (compared with the PVS default decision procedures) for selected problems. and for typical problems we are usually able to process several thousand theorems every second. The efficiency and scalability of ICS in processing formulas, the richness of its API, and its ability for fast context-switching should make it possible to use it as a black box for discharging verification conditions not only in a theorem proving context but also in a multitude of applications like static analysis, abstract interpretation, extended type checking, symbolic simulation, model checking, or compiler optimization.

1.1 Availability

For academic, non-commercial use ICS is available free of charge under a license agreement with SRI. ICS is also an integral part of PVS 3.0. The complete sources and documentation of ICS are available at www.icansolve.com

There one can also find the ICS license.

1.2 Organization

This document describes the interfaces and implementation aspects of the ICS decision procedures. The various modules of the implementation are not presented in topological order with respect to their inter-dependencies, but rather in a more logical order. However, it may help to have in mind the graph of dependencies, which is the following:

2 Installation

Distribution. The file `ics.tar.gz` can be downlowded from www.icansolve.com. Unpack this file using

```
> zcat ics.tar.gz | tar xvf -
```

This creates a directory `./ics` with the following files and directory.

`Makefile.in`	:	Template for generating `Makefile`.
`fm-license.in`	:	Noncommercial license.
`bin/`	:	Binaries
`configure`	:	Configuration script
`lib/`	:	Archives and shared object files
`tests/`	:	Various benchmarks
`README`	:	Short installation guide
`doc/`	:	Documentation
`obj/`	:	Object files
`src/`	:	Source files
`ics/`	:	Shell script for invoking ICS interactor

The latest version of ICS can be obtained by using the CVS checkout command

```
> cvs co ics -d $CVS_root
```

where `CVS_root` is set to

```
/project/pvs2/cvsroot
```

Requirements. ICS is written mainly in Ocaml, and it uses uses arbitrary precision rational numbers from the GNU multi-precision library (GMP). To compile ICS one needs to install:

- The autoconf package is freely available at http://www.gnu.org/software/autoconf/.
- Ocaml version 3.04 or later. Freely available at http://caml.inria.fr.
- GNU MP version 3.0 or later. This package is freely available at http://www.swox.com/gmp/.

So far, we have only compiled ICS on Linux Redhat 6.0, but it should be possible to compile it on a wide range of machines and operating systems.

Installation. A configuration `./configure` is generated from `./configure.in` in the ICS home directory using

```
> autoconf
```

The configuration script generates a `Makefile` from the `Makefile.in`.

```
> ./configure [--with-gmp=/path/to/gmp]
```

Then, `make` compiles ICS on your machine.

```
> make
```

Binaries are placed in `./bin/$(ARCH)/` and the libraries in `./lib/$(ARCH)/`, where `ARCH` is the architecture guessed by the configuration script. The build directory is `./obj/$(ARCH)/`, and the generated binary and byte code are put in `./bin/$(ARCH)/`.

3 The ICS interactor

The interactor permits to process formulas interactively and to explore the database. We give an overview of the capabilities of ICS using various little examples.

The interactor is started with `./ics` in the ICS home directory.

```
% ics
ICS interpreter. Copyright (c) 2001 SRI International.

>
```

The '>' is the prompt and ICS is ready to interpret your commands.

3.1 The Command Language

The ICS command language realizes a *ask/tell* interface to a context consisting of known facts. When invoking the interactor, this context is empty.

Asserting facts.

`assert` *atom* `.`

An *atom* is asserted to the current context using the `assert` command. There are three possible outcomes:

1. *atom* is inconsistent with respect to the current context. In this case, `assert` leaves the current context unchanged and outputs `Unsat.` on the standard output.

2. *atom* is valid in the current context. Again, the the current context is left, and now `Valid.` is output.

3. Otherwise, in case *atom* is satisfiable but not valid, the context is modified to include new information obtained from *atom*.

Resetting.

`reset .`

Reinitializes all internal data structures including setting the current logical context to the empty context.

Clearing current logical context.

`forget .`

Resets the current logical context to the empty context. In contrast to `reset`, all other ICS data structures are left unchanged.

Sigmatization.

`sigma` (*term*| *atom*) `.`

Computes the normal form of a term or an atom using theory-specific canonizers for terms in interpreted theories and some builtin simplifications for uninterpreted terms. This command leaves the current state unchanged.

Canonization.

`can` (*term*| *atom*) `.`

For a term `t`, `can t` returns a variable, which is a canonical representative of `t`.

For an atom `a`, `can a` returns a semicanonical form which reduces to the atom `True` if [a] can be shown to hold in the current context, and `False` if `a` can be shown to be inconsistent. Otherwise, an atom [b] is returned which is equivalent to [a] in the current context.

[can] might involve a state change in that renaming variables of the form `v!i` are generated and equalities `v!i = a'` for subterms of `a` are added to the logical context.

Solving.

`solve` (`a` | `t bv`) *term*= *term* `.`

Theory-specific solver for input equality. Returns either a solved list of equalities with variables on the lhs which is equivalent to the input equality or `Unsat.` if the input equality is unsatisfiable. There are solvers for linear arithmetic (`a`), tuples (`t`), and bitvectors (`bv`).

Logical Context.

`ctxt .`

Return the set of atoms asserted in the current logical context. These atoms are not necessarily in canonical form.

Variable Partitioning.

`partition .`

Returns the set of all known equalities between variables.

Solution sets.

`solution` (`u` | `a` | `t bv`) `.`

Returns the solution set corresponding to the specified theory, which consists of a set of equalities `x = a` with `x` a variable and `a` a term built-up from variables and function symbols in the specified theory. Variables on the rhs might either be external variables or fresh variables introduced by a theory-specific solver. For all interpreted theories, that is, for all theories except for the uninterpreted theory `u`, the equations in a solved form are actually solved in that variables `x` on a rhs do not occur in any of the lhs.

Finds in solution sets.

`find` (`u` | `a` | `t bv`) *term* `.`

If `x = t` is in the specified solution set (see command `solution`), then `find . x` returns `t` and otherwise `x`.

Inverse Finds in solution sets.

`find` (`u` | `a` | `t bv`) *term* `.`

If `x = t` is in the specified solution set (see command `solution`), then `inv . t` returns `x` and otherwise `Unsat.`.

Comparison.

term`«` *term* `.`

Syntactic comparison of two terms. `s « t` returns `Less` (`Equal`, `Greater` if `s` is less (equal, greater) than `t` according to the builtin term ordering.

Arithmetic Constraints.

`cnstrnt` *term* `.`

Compute an arithmetic constraint for the argument term in the current context using abstract interpretation on intervals. Returns `None.` if no such constraint could be deduced.

Disequalities.

`diseq` *term* .

Returns a list of variables known to be disequal in the current context.

Displaying the context.

`show .`

Displays the variable partitioning, theory-specific solution sets, and disequality information. See also the commands `partition`, `solution`, and `diseq`.

Term Definition.

`def` *var* `:=` *term* .

Extend the symbol table with a definition *var*for term *term*. In such a context, variable *var*is always expanded to *term*.

Type Definition.

`type` *var* `:=` *cnstrnt* .

Extend the symbol table with a definition *var*for the constraint *cnstrnt*. In such a context, variable *var*can always be used instead of the corresponding constraint.

Signature Declaration.

`sig` *var* `:= bv[int] .`

Declare a variable to be interpreted over the set of bitvectors of width `int`. This context information is used for inferring parameters when applying infix bitvector operators.

Symbol Table.

`symtab` {*var*} .

`symtab` display the current symbol table, and `symtab` *var*displays the symbol table entry for *var*.

Saving the current logical context.

`save` {*var*} .

Adding a symbol table entry for the current logical state.

Restoring logical contexts.

`restore` $\{var\}$ `.`

Updating the current logical state to be the state named by *var*in the symbol table.

Removing symbol table entries

`remove` $\{var\}$ `.`

Remove the symbol table entry corresponding to *var*.

Verbose.

`verbose` *int* `.`

Determines the amount of trace information displayed on standard output. The larger the argument the more information is output.

Garbage Collection.

`gc .`

This command results in a logical context which is equivalent to the current one but redundant variables are eliminated.

Debugging.

`drop .`

The effect of this command is to fall back into the Ocaml interactor when running bytecode; otherwise ICS is terminated. This command is mainly for debugging purposes.

Exiting ICS.

`exit .`

Exit the ICS interactor. Alternatively, `Ctrl-D` can be used.

4 Application Programming Interface

Usually, the capabilites of ICS are not accessed through the interactor but rather through its application programming interface. Currently, we support interfaces for C, Fortran, Lisp, and Ocaml. We first describe the Ocaml interface, since the interfaces for the other programming languages are automatically generated from this one.

4.1 Caml interface

Interface for module Ics.mli

1. Module *Ics*: the application programming interface to ICS for asserting formulas to a logical context, switching between different logical contexts, and functions for manipulating and normalizing terms.

There are two sets of interface functions. The functional interface provides functions for building up the main syntactic categories of ICS such as terms and atoms, and for extending logical contexts using *process*, which is side-effect free.

In contrast to this functional interface, the command interface manipulates a global state consisting, among others, of symbol tables and the current logical context. The *cmd_rep* procedure, which reads commands from the current input channel and manipulates the global structures accordingly, is used to implement the ICS interactor.

Besides functions for manipulating ICS datatypes, this interface also contains a number of standard datatypes such as channels, multiprecision arithmetic, tuples, and lists.

2. Controls. *reset* clears all the global tables. This does not only include the current context but also internal tables used for hash-consing and memoization purposes. *gc* triggers a full major collection of ocaml's garbage collector. *do_at_exit* clears out internal data structures.

val $reset$: $unit \rightarrow unit$

val gc : $unit \rightarrow unit$

val do_at_exit : $unit \rightarrow unit$

3. *set_maxloops* n determines an upper number of loops in the main ICS loop. $n < 0$ determines that there is no such bound; this is also the default.

val $set\_maxloops$: $int \rightarrow unit$

4. Rudimentary control on trace messages, which are sent to *stderr*. These functions are mainly included for debugging purposes, and are usually not being used by the application programmer. *trace_add str* enables tracing of functions associated with trace level *str*. *trace_add* `"all"` enables all tracing. *trace_remove str* removes *str* from the set of active trace levels, and *trace_reset*() disables all tracing. *trace_get*() returns the set of active trace levels.

val $trace\_reset$: $unit \rightarrow unit$
val $trace\_add$: $string \rightarrow unit$
val $trace\_remove$: $string \rightarrow unit$
val $trace\_get$: $unit \rightarrow string\ list$

5. Channels. *inchannel* is the type of input channels. A channel of name *str* is opened with *in_of_string str*. This function raises *Sys_error* in case such a channel can not be opened. *outchannel* is the type of formatting output channels, and channels of this type are opened with out_of_string. $stdin$, $stdout$, and $stderr$ are predefined channels for standard input, standard output, and standard error. $flush$ flushes the $stdout$ channel.

type $inchannel = in\_channel$
type $outchannel = Format.formatter$

val $channel\_stdin : unit \to inchannel$
val $channel\_stdout : unit \to outchannel$
val $channel\_stderr : unit \to outchannel$
val $inchannel\_of\_string : string \to inchannel$
val $outchannel\_of\_string : string \to outchannel$

val $flush : unit \to unit$

6. Multi-precision rational numbers. $num\_of\_int\ n$ injects an integer into this type, $num\_of\_ints\ n\ m$, for $m \neq 0$, constructs a normalized representation of the rational n/m in q, $string\_of\_num\ q$ constructs a string (usually for printout) of a rational number, and $num\_of\_string\ s$ constructs a rational, whenever s is of the form `"n/m"` where n and m are naturals.

type q

val $num\_of\_int : int \to q$
val $num\_of\_ints : int \to int \to q$
val $ints\_of\_num : q \to string \times string$
val $string\_of\_num : q \to string$
val $num\_of\_string : string \to q$

7. Names. $name\_of\_string$ and $name\_to\_string$ coerce between the datatypes of strings and names. These coercions are inverse to each other. $name\_eq$ tests for equality of names in constant time.

type $name$

val $name\_of\_string : string \to name$
val $name\_to\_string : name \to string$
val $name\_eq : name \to name \to bool$

8. Arithmetic constraints. A constraint consists of a domain restriction Int or $Real$, a real interval, and a set of disequality numbers. A real number satisfies such a constraint if, first, it satisfies the domain restriction, second, it is a member of the interval, and, third, it is none of the numbers in the disequality set.

$cnstrnt\_of\_string\ str$ parses the string str according to the nonterminal $cnstrnteof$ in module $Parser$ (see its specification in file parser.mly) and produces the corresponding constraint representation. In contrast, $cnstrnt\_input$ in parses the concrete syntax of constraints from the input channel in. Constraints c are printed to the output channel out using $cnstrnt\_output\ out\ c$ and to the standard output using $cnstrnt\_pp\ c$.

For the definition of constraint constructors see Module $Cnstrnt$. $cnstrnt\_mk\_int()$ constructs an integer constraint, $cnstrnt\_mk\_nat()$ a constraint for the natural numbers,

$cnstrnt\_mk\_singleton\ q$ is the constraint which holds only of q, $cnstrnt\_mk\_diseq\ q$ holds for all reals except for q, $cnstrnt\_mk\_oo\ l\ h$ constructs an open interval with lower bound l and upper bound h, $cnstrnt\_mk\_oc\ l\ h$ is the left-open, right-closed interval with lower bound l and upper bound h, $cnstrnt\_mk\_co\ l\ h$ is the left-closed, right-open interval with lower bound l and upper bound h, $cnstrnt\_mk\_cc\ l\ h$ is the closed interval with lower bound l and upper bound h.

The intersection of two constraints c, d is computed by $cnstrnt\_inter\ c\ d$, that is, a real q is in both c and d iff it is in $cnstrnt\_inter\ c\ d$.

type $cnstrnt$

val $cnstrnt\_of\_string : string \to cnstrnt$
val $cnstrnt\_input : inchannel \to cnstrnt$
val $cnstrnt\_output : outchannel \to cnstrnt \to unit$
val $cnstrnt\_pp : cnstrnt \to unit$

val $cnstrnt\_mk\_int : unit \to cnstrnt$
val $cnstrnt\_mk\_nonint : unit \to cnstrnt$
val $cnstrnt\_mk\_nat : unit \to cnstrnt$
val $cnstrnt\_mk\_singleton : q \to cnstrnt$
val $cnstrnt\_mk\_diseq : q \to cnstrnt$
val $cnstrnt\_mk\_oo : q \to q \to cnstrnt$
val $cnstrnt\_mk\_oc : q \to q \to cnstrnt$
val $cnstrnt\_mk\_co : q \to q \to cnstrnt$
val $cnstrnt\_mk\_cc : q \to q \to cnstrnt$
val $cnstrnt\_mk\_lt : q \to cnstrnt$
val $cnstrnt\_mk\_le : q \to cnstrnt$
val $cnstrnt\_mk\_gt : q \to cnstrnt$
val $cnstrnt\_mk\_ge : q \to cnstrnt$

val $cnstrnt\_inter : cnstrnt \to cnstrnt \to cnstrnt$

9. Abstract interval interpretation. A real number x is in $cnstrnt\_add\ c\ d$ iff there are real numbers y in c and z in d such that $x = y + x$. Likewise, x is in $cnstrnt\_multq\ q\ c$ iff there exists y in c such that $x = q \times y$. In constrast to these exact abstract operators, $cnstrnt\_mult$ and $cnstrnt\_div$ compute overaprroximations, that is, if x in c and y in d then there exists a z in $cnstrnt\_mult\ c\ d$ ($cnstrnt\_div\ c\ d$) such that $z = x \times y$ ($z = x/y$).

val $cnstrnt\_add : cnstrnt \to cnstrnt \to cnstrnt$
val $cnstrnt\_multq : q \to cnstrnt \to cnstrnt$
val $cnstrnt\_mult : cnstrnt \to cnstrnt \to cnstrnt$
val $cnstrnt\_div : cnstrnt \to cnstrnt \to cnstrnt$

10. Theories. A theory is associated with each function symbol. These theories are indexed by naturals between 0 and 8 according to the following table

0 Theory of uninterpreted function symbols. 1 Linear arithmetic theory. 2 Product theory. 3 Bitvector theory. 4 Coproducts. 5 Power products. 6 Theory of function abstraction and application. 7 Array theory. 8 Theory of bitvector interpretation(s).

type $th = int$

val $th\_to\_string : th \to string$

11. Function symbols. These are partitioned into uninterpreted function symbols and function symbols interpreted in one of the builtin theories. For each interpreted function symbol there is a recognizer function is_xxx. Some values of type sym represent families of function symbols. The corresponding indices can be obtained using the destructor d_xxx functions (only after checking that is_xxx holds.

type sym

val $sym\_is\_uninterp : sym \to bool$
val $sym\_is\_interp : th \to sym \to bool$

12. sym_eq tests for equality of two function symbols.

val $sym\_eq : sym \to sym \to bool$

13. sym_cmp provides a total ordering on function symbols. It returns a negative integer if $s < t$, 0 if s is equal to t, and a positive number if $s > t$.

val $sym\_cmp : sym \to sym \to int$

14. Arithmetic function symbols are either numerals, addition, or linear multiplication.

val $sym\_is\_num : sym \to bool$
val $sym\_d\_num : sym \to q$

val $sym\_is\_add : sym \to bool$

val $sym\_is\_multq : sym \to bool$
val $sym\_d\_multq : sym \to q$

Symbols interpreted in the theory of products.

val $sym\_is\_tuple : sym \to bool$

val $sym\_is\_proj : sym \to bool$
val $sym\_d\_proj : sym \to int \times int$

15. Symbols interpreted in the theory of coproducts.

val $sym\_is\_inl : sym \to bool$
val $sym\_is\_inr : sym \to bool$
val $sym\_is\_outl : sym \to bool$
val $sym\_is\_outr : sym \to bool$

16. Symbols in the bitvector theory.

val $sym\_is\_bv\_const : sym \to bool$

val $sym\_is\_bv\_conc : sym \to bool$
val $sym\_d\_bv\_conc : sym \to int \times int$

val $sym\_is\_bv\_sub \;:\; sym \;\rightarrow\; bool$
val $sym\_d\_bv\_sub \;:\; sym \;\rightarrow\; int \times int \times int$

val $sym\_is\_bv\_bitwise \;:\; sym \;\rightarrow\; bool$
val $sym\_d\_bv\_bitwise \;:\; sym \;\rightarrow\; int$

17. Symbols from the theory of power products.

val $sym\_is\_mult \;:\; sym \;\rightarrow\; bool$

val $sym\_is\_expt \;:\; sym \;\rightarrow\; bool$

18. Symbols from the theory of function abstraction and application.

val $sym\_is\_apply \;:\; sym \;\rightarrow\; bool$
val $sym\_d\_apply \;:\; sym \;\rightarrow\; cnstrnt\ option$

val $sym\_is\_abs \;:\; sym \;\rightarrow\; bool$

19. Symbols from the theory of arrays.

val $sym\_is\_select \;:\; sym \;\rightarrow\; bool$

val $sym\_is\_update \;:\; sym \;\rightarrow\; bool$

20. Symbols from the theory of arithmetic interpretations of bitvectors.

val $sym\_is\_unsigned \;:\; sym \;\rightarrow\; bool$

21. Terms. Terms are either variables, application of uninterpreted functions, or interpreted constants and operators drawn from a combination of theories..

$term\_of\_string$ parses a string according to the grammar for the nonterminal $termeof$ in module $Parser$ (see its specification in file parser.mly) and builds a corresponding term. Similary, $term\_input$ builds a term by reading from an input channel.

$term\_output\ out\ a$ prints term a on the output channel out, and $term\_pp\ a$ is equivalent to $term\_output\ stdout\ a$.

Terms are build using constructors, whose names are all of the form mk_xxx. For each constructor mk_xxx there is a corresponding recognizer is_xxx which reduces to true if its argument term has been built with the constructor mk_xxx. Moreover, for each constructor $mk\_xxx\}$ above there is a corresponding desctructor d_xxx for analyzing the components of such a constructor term.

type $term$

val $term\_of\_string \;:\; string \rightarrow\; term$
val $term\_to\_string \;:\; term \;\rightarrow\; string$
val $term\_input \;:\; inchannel \;\rightarrow\; term$
val $term\_output \;:\; outchannel \;\rightarrow\; term \;\rightarrow\; unit$
val $term\_pp \;:\; term \;\rightarrow\; unit$

22. Equality and Comparison. Comparison $cmp\ a\ b$ returns either -1, 0, or 1 depending on whether a is less than b, the arguments are equal, or a is greater than b according to the builtin term ordering (see $Term.(<<<)$). $term\_eq\ a\ b$ is true iff if $term\_cmp\ a\ b$ returns 0.

val $term\_eq \; : \; term \; \rightarrow \; term \; \rightarrow \; bool$
val $term\_cmp \; : \; term \; \rightarrow \; term \; \rightarrow \; int$

23. Given a string s, $term\_mk\_var\ s$ constructs a variable with name s and $term\_mk\_uninterp\ s\ al$ constructs an application of an uninterpreted function symbol s to a list of argument terms. If s is any of the builtin function symbols specified in module *Builtin*, the builtin simplifications are applied to this application.

val $term\_mk\_var \; : \; string \rightarrow \; term$

val $term\_mk\_uninterp \; : \; string \rightarrow \; term\ list \rightarrow \; term$

24. Arithmetic terms include rational constants built from $term\_mk\_num\ q$, linear multiplication $term\_mk\_multq\ q\ a$, addition $term\_mk\_add\ a\ b$ of two terms, n-ary addition $term\_mk\_addl\ al$ of a list of terms al, subtraction $term\_mk\_sub\ a\ b$ of term b from term a, negation $term\_mk\_unary\_minus\ a$, multiplication $term\_mk\_mult\ a\ b$, and exponentiation $term\_mk\_expt\ n\ a$. These constructors build up arithmetic terms in a canonical form as defined in module *Arith*. $term\_is\_arith\ a$ holds iff the toplevel function symbol of a is any of the function symbols interpreted in the theory of arithmetic.

val $term\_mk\_num \; : \; q \; \rightarrow \; term$
val $term\_mk\_multq \; : \; q \; \rightarrow \; term \; \rightarrow \; term$
val $term\_mk\_add \; : \; term \; \rightarrow \; term \; \rightarrow \; term$
val $term\_mk\_addl \; : \; term\ list \rightarrow \; term$
val $term\_mk\_sub \; : \; term \; \rightarrow \; term \; \rightarrow \; term$
val $term\_mk\_unary\_minus \; : \; term \; \rightarrow \; term$

val $term\_is\_arith \; : \; term \; \rightarrow \; bool$

25. Tuples are built using the mk_tuple constructor, and projection of the i-th component of a tuple t of length n is realized using $mk\_proj\ i\ n\ t$.

val $term\_mk\_tuple \; : \; term\ list \rightarrow \; term$
val $term\_mk\_proj \; : \; int \rightarrow \; int \rightarrow \; term \; \rightarrow \; term$

26. Boolean constants.

val $term\_mk\_true \; : \; unit \rightarrow \; term$
val $term\_mk\_false \; : \; unit \rightarrow \; term$

val $term\_is\_true \; : \; term \; \rightarrow \; bool$
val $term\_is\_false \; : \; term \; \rightarrow \; bool$

27. Bitvectors

val $term\_mk\_bvconst \; : \; string \rightarrow \; term$
val $term\_mk\_bvsub \; : \; (int \times int \times int) \; \rightarrow \; term \; \rightarrow \; term$
val $term\_mk\_bvconc \; : \; int \times int \rightarrow \; term \; \rightarrow \; term \; \rightarrow \; term$
val $term\_mk\_bwite \; : \; int \rightarrow \; term \; \times \; term \; \times \; term \; \rightarrow \; term$
val $term\_mk\_bwand \; : \; int \rightarrow \; term \; \rightarrow \; term \; \rightarrow \; term$

val $term\_mk\_bwor$: $int \to term \to term \to term$
val $term\_mk\_bwnot$: $int \to term \to term$

28. Coproducts.

val $term\_mk\_inj$: $int \to term \to term$
val $term\_mk\_out$: $int \to term \to term$

29. Set of terms.

type $terms$

30. Atoms. $atom\_mk\_true()$ is the trivially true atom, $atom\_mk\_false()$ is the trivially false atom, and given terms a, b, the constructor $mk\_equal\ a\ b$ constructs an equality constraint, $mk\_diseq\ a\ b$ a disequality constraint, and $atom\_mk\_in\ a\ c$ constructs a membership constraint for a and a arithmetic constraint c. Atoms are printed to $stdout$ using $atom\_pp$.

type $atom$

val $atom\_pp$: $atom \to unit$

val $atom\_of\_string$: $string \to atom$
val $atom\_to\_string$: $atom \to string$

val $atom\_mk\_equal$: $term \to term \to atom$
val $atom\_mk\_diseq$: $term \to term \to atom$
val $atom\_mk\_in$: $cnstrnt \to term \to atom$
val $atom\_mk\_true$: $unit \to atom$
val $atom\_mk\_false$: $unit \to atom$

31. Derived atomic constraints. $atom\_mk\_int\ t$ restricts the domain of interpretations of term t to the integers. Similarly, $atom\_mk\_real$ restricts its argument to the real numbers. $atom\_mk\_lt\ a\ b$ generates the constraint `'a'` $<$ `'b'`, $atom\_mk\_le\ a\ b$ yields `'a'` $\leq$ `'b'`, $atom\_mk\_gt\ a\ b$ yields `'a'` $>$ `'b'`, and $atom\_mk\_ge\ a\ b$ yields `'a'` $\geq$ `'b'`.

val $atom\_mk\_real$: $term \to atom$
val $atom\_mk\_int$: $term \to atom$
val $atom\_mk\_nonint$: $term \to atom$

val $atom\_mk\_lt$: $term \to term \to atom$
val $atom\_mk\_le$: $term \to term \to atom$
val $atom\_mk\_gt$: $term \to term \to atom$
val $atom\_mk\_ge$: $term \to term \to atom$

32. Solution sets.

type $solution$

val $solution\_apply$: $solution \to term \to term$

val $solution\_find$: $solution \to term \to term$

val $solution\_inv$: $solution$ $\rightarrow$ $term$ $\rightarrow$ $term$

val $solution\_mem$: $solution$ $\rightarrow$ $term$ $\rightarrow$ $bool$

val $solution\_occurs$: $solution$ $\rightarrow$ $term$ $\rightarrow$ $bool$

val $solution\_use$: $solution$ $\rightarrow$ $term$ $\rightarrow$ $terms$

val $solution\_is\_empty$: $solution$ $\rightarrow$ $bool$

33. Logical context. An element of type *state* is a logical context with *state_empty* the empty context. *state_eq s1 s2* is a constant-time predicate for testing for identity of two states. Thus, whenever this predicate holds its arguments states are equivalent, but not necessarily the other way round. Logical contexts are printed using *state_pp*. The set of atoms in a context s are obtained with *state_ctxt_of* s.

type $context$

val $context\_eq$: $context$ $\rightarrow$ $context$ $\rightarrow$ $bool$

val $context\_empty$: $unit \rightarrow$ $context$

val $context\_ctxt\_of$: $context$ $\rightarrow$ $atom\ list$

val $context\_u\_of$: $context$ $\rightarrow$ $solution$

val $context\_a\_of$: $context$ $\rightarrow$ $solution$

val $context\_t\_of$: $context$ $\rightarrow$ $solution$

val $context\_bv\_of$: $context$ $\rightarrow$ $solution$

val $context\_pp$: $context$ $\rightarrow$ $unit$

val $context\_ctxt\_pp$: $context$ $\rightarrow$ $unit$

34. Builtin simplifying constructors.

val $term\_mk\_unsigned$: $term$ $\rightarrow$ $term$

val $term\_mk\_update$: $term$ $\rightarrow$ $term$ $\rightarrow$ $term$ $\rightarrow$ $term$
val $term\_mk\_select$: $term$ $\rightarrow$ $term$ $\rightarrow$ $term$

val $term\_mk\_div$: $term$ $\rightarrow$ $term$ $\rightarrow$ $term$

val $term\_mk\_mult$: $term$ $\rightarrow$ $term$ $\rightarrow$ $term$
val $term\_mk\_multl$: $term\ list \rightarrow$ $term$
val $term\_mk\_expt$: $int \rightarrow$ $term$ $\rightarrow$ $term$
(* *term_mk_expt* n x represent $x\hat{}n$. *)

val $term\_mk\_apply$: $term$ $\rightarrow$ $term\ list \rightarrow$ $term$

val $term\_mk\_arith\_apply$: $cnstrnt$ $\rightarrow$ $term$ $\rightarrow$ $term\ list \rightarrow$ $term$

35. The operation *process* s a adds a new atom a to a logical context s. The codomain of this function is of type *status*, elements of which represent the three possible outcomes of processing a proposition: 1. the atom a is inconsistent in s, 2. it is valid, or, 3., it is satisfiable but not valid. In the third case, a modified state is obtained using the destructor $d\_consistent$.

type $status$

val $is\_consistent : status \to bool$
val $is\_redundant : status \to bool$
val $is\_inconsistent : status \to bool$

val $d\_consistent : status \to context$

val $process : context \to atom \to status$

36. Suggesting finite case split.

val $split : context \to atom\ list$

37. Canonization. Given a logical context s and an atom a, $can\ s\ a$ computes a semi-canonical form of a in s, that is, if a holds in s it returns $Atom.True$, if the negation of a holds in s then it returns $Atom.False$, and, otherwise, an equivalent normalized atom built up only from variables is returned. The returned logical state might contain fresh variables.

val $can : context \to atom \to atom$

38. Given a logical context s and a term a, $cnstrnt\ s\ a$ computes the best possible arithmetic constraint for a in s using constraint informatin in s and abstraction interval interpretation. If no such constraint can be deduced, $None$ is returned.

val $cnstrnt : context \to term \to cnstrnt\ option$

39. An imperative state $istate$ does not only include a logical context of type $state$ but also a symbol table and input and output channels. A global $istate$ variable is manipulated and destructively updated by commands.

40. Initialization. $init\ n$ sets the verbose level to n. The higher the verbose level, the more trace information is printed to $stderr$ (see below). There are no trace messages for $n = 0$. In addition, initialization makes the system to raise the $Sys.Break$ exception upon user interrupt $\hat{}C\hat{}C$. The $init$ function should be called before using any other function in this API.

val $init : int \times bool \times string \times inchannel \times outchannel \to unit$

41. cmd_eval reads a command from the current input channel according to the grammar for the nonterminal $commandeof$ in module $Parser$ (see its specification in file parser.mly, the current internal $istate$ accordingly, and outputs the result to the current output channel.

val $cmd\_rep : unit \to unit$

42. Sleeping for a number of seconds.

val $sleep : int \to unit$

43. Lists.

val is_nil : α $list \rightarrow$ $bool$
val $cons$: α $\rightarrow$ α $list \rightarrow$ α $list$
val $head$: α $list \rightarrow$ α
val $tail$: α $list \rightarrow$ α $list$

44. Pairs. $pair$ a b builds a pair (a, b) and fst $(pair$ a $b)$ returns a and snd $(pair$ b $a)$ returns b.

val $pair$: $\alpha \rightarrow \beta \rightarrow \alpha \times \beta$
val fst : $\alpha \times \beta \rightarrow \alpha$
val snd : $\alpha \times \beta \rightarrow \beta$

45. Triples. Accessors for triples (a, b, c).

val $triple$: $\alpha \rightarrow \beta \rightarrow \gamma \rightarrow \alpha \times \beta \times \gamma$
val fst_of_triple : $\alpha \times \beta \times \gamma \rightarrow \alpha$
val snd_of_triple : $\alpha \times \beta \times \gamma \rightarrow \beta$
val $third\_of\_triple$: $\alpha \times \beta \times \gamma \rightarrow \gamma$

46. Quadruples. Accessors for quadruples (a, b, c, d).

val $fst\_of\_quadruple$: $\alpha \times \beta \times \gamma \times \delta \rightarrow \alpha$
val $snd\_of\_quadruple$: $\alpha \times \beta \times \gamma \times \delta \rightarrow \beta$
val $third\_of\_quadruple$: $\alpha \times \beta \times \gamma \times \delta \rightarrow \gamma$
val $fourth\_of\_quadruple$: $\alpha \times \beta \times \gamma \times \delta \rightarrow \delta$

47. Options. An element of type α $option$ either satisfies the recognizer is_some or is_none. In case, is_some holds, a value of type α can be obtained by $value\_of$.

val is_some : α $option \rightarrow$ $bool$
val is_none : α $option \rightarrow$ $bool$

val $value\_of$: α $option \rightarrow$ α

4.2 C Interface

The API for the C programming language is generated automatically from the Ocaml API described above. The generated C file can be found in

```
./obj/$ARCH/ics_stub.c
```

This file contains a C function declaration `ics_xxx` for each of the interface function `xxx` described above. For example, the definition of the function `ics_mk_var` for the `mk_var` constructor is given by the following C code.

```
value* ics_mk_var(char* x1) {
  value* ics_mk_var(char* x1) {
  value* r = malloc(sizeof(value));
```

```
  register_global_root(r);
  *r = 1;
  *r = callback_exn(*ics_mk_var_rv,copy_string(x1));
  if (!Is_exception_result(*r)) { return r; };
  ocaml_error("ics_mk_var",format_caml_exception(Extract_exception(*r)));
  return (value*) 0;
}
```

These interface function translate C arguments to Ocaml values, call the Ocaml function, and translate back the results. In addition, any Ocaml exceptions are caught and handled by the `ocaml_error` function. Curried signatures of the Ocaml functions are uncurried, and list and tuple arguments must be build using the constructors of the interface. The handling of exceptions is determined by the function `ocaml_error`, which has to be provided by the application programmer.

When using `C++` the following declarations are needed to use ICS. First, declare a function `ics_caml_startup` before including `ics.h`.[1]

```
extern "C" {
void ics_caml_startup(int full, char** argv);
#include<ics.h>
}
```

Second, an application-dependent `ics_error` function such as the one below has to be provided.

```
extern "C" {

void ics_error(char * funname, char * message) {
    cerr << "ICS error at " << funname << " : " << message << endl;
    exit(1);
}
}
```

Third, before calling any ICS functionality, call `ics_caml_startup`.

```
int main(int argc, char ** argv) {
    ics_caml_startup(1, argv);
    ...
}
```

A minimal `C++` program for calling ICS can be found in Figure 1. If this program is stored in a file `hello-ics.cpp`, then it can be compiled using

```
  g++ hello-ics.cpp -lics
```

Notice that `LD_LIBRARY_PATH` variable should be such that the shared object file `libics.so` can be found in the linking stage.

[1] Within the `extern "C"` directive, the `C++` compiler does not rename functions.

```
#include<iostream.h>
extern "C" {
void ics_caml_startup(int full, char** argv);
#include<ics.h>
}

int main(int argc, char ** argv) {
    ics_caml_startup(1, argv);
    cout << "ICS: Hello World\n";
}


extern "C" {
void ics_error(char * funname, char * message) {
    cerr << "ICS error at " << funname << " : " << message << endl;
    exit(1);
}}
```

Figure 1: Minimal setup for calling ICS.

4.3 Lisp Interface

The Lisp API for ICS builds on the C interface and uses the foreign function interface of Allegro Common Lisp 6.0. For each function `xxx` in the API a foreign function declaration `ics_xxx` is generated. These definitions are collected in the file

```
./obj/$ARCH/ics.lisp
```

The Lisp interface ensures that the Lisp and the Ocaml garbage collector cooperate.

5 Data Structures

Terms and atoms are the main syntactic categories of ICS, where atoms are equalities or disequalities over terms or interval constraints over terms.

- A term is either a variable or an application of a function symbol to a list of terms. (see Module **??**)
- Constraints consists of an interval and a set of so-called exclusive numbers.
- An atom is either an equality over terms, a disequality over terms, or a constraint for a term.

A partitioning `(v, d)` is defined as the conjunction of the variable equalities in `v` and variable disequalities in `d`, where `v` is of type `V.t` and `d` is of type `D.t`.

ICS supports the theories of equality over uninterpreted function symbols and a collection of interpreted theories including arithmetic, tuples, and bitvectors. For each function symbol in interpreted theories there is corresponding constructor function for building terms in a well-defined canonical form. In addition, for certain uninterpreted function symbols there are builtin rewrite rules. Solvers are only defined for interpreted theory.

This chapter describes the utility modules. The module *Tools* contained utility functions used in the implementation. The modules *Lexer* and *Parser* provide parsers for terms, equations, commands, etc. (we only give the interface for module *Lexer*). The module *Test* is a tiny toplevel to test the implementation. It allows the user to test the canonization and solver functions, and the algorithm.

6 Modules

6.1 Module sign

Interface for module Sign.mli

* Three-valued datatype for classifying real numbers into negative (*Neg*) numbers, the *Zero* number, and positive numbers (*Pos*).

type $t = Neg \mid Zero \mid Pos$

Module Sign.ml

* Three-valued datatype for classifying real numbers into negative (*Neg*) numbers, the *Zero* number, and positive numbers (*Pos*).

type $t = Neg \mid Zero \mid Pos$

6.2 Module three

Interface for module Three.mli

* Three-valued datatype.

There is an implicit partial ordering with $Yes < X$ and $No < X$.

type $t =$
$\mid Yes$
$\mid No$
$\mid X$

val $is\_sub : t \rightarrow t \rightarrow bool$
(** $is\_sub\ u\ v$ holds if either $u = v$ or $u < v$. *)

val $inter : t \rightarrow t \rightarrow t\ option$
(** $inter\ u\ v$ evaluates to $Some(w)$ if both $is\_sub\ w\ u$ and $is\_sub\ w\ v$ *)

```
val union : t → t → t
(** union u v evaluates to w if both is_sub u w and is_sub v w holds. *)
val is_disjoint : t → t → bool
```

Module Three.ml

```
type t =
  | Yes
  | No
  | X

let is_sub a b =
  match a, b with
    | _, X → true
    | (X | No), Yes → false
    | Yes, Yes → true
    | (X | Yes), No → false
    | No, No → true

let inter =
  let yes = Some(Yes) in (** avoid repetitive creation of constants. *)
  let no = Some(No) in
  let x = Some(X) in
  fun a b → match a, b with
    | No, X → no
    | Yes, X → yes
    | X, No → no
    | X, Yes → yes
    | No, No → no
    | X, X → x
    | Yes, Yes → yes
    | No, Yes → None
    | Yes, No → None

let union a b =
  match a, b with
    | _, X → X
    | X, _ → X
    | No, No → No
    | Yes, Yes → Yes
    | No, Yes → X
    | Yes, No → X

let is_disjoint a b =
  inter a b = None
```

6.3 Module gmp

Interface for module Gmp.mli

module *Z2* : sig
  type *t*
  external *copy* : $t \rightarrow t \rightarrow t$ = "ml_mpz2_copy"
  external *from_int* : $t \rightarrow int \rightarrow t$ = "ml_mpz2_from_int"
  external *from_string* : $t \rightarrow string \rightarrow int \rightarrow t$ = "ml_mpz2_from_string"
  external *from_float* : $t \rightarrow float \rightarrow t$ = "ml_mpz2_from_float"
  external *add* : $t \rightarrow t \rightarrow t \rightarrow t$ = "ml_mpz2_add"
  external *sub* : $t \rightarrow t \rightarrow t \rightarrow t$ = "ml_mpz2_sub"
  external *mul* : $t \rightarrow t \rightarrow t \rightarrow t$ = "ml_mpz2_mul"
  external *pow_ui* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_pow_ui"
  external *pow_ui_ui* : $t \rightarrow int \rightarrow int \rightarrow t$ = "ml_mpz2_ui_pow_ui"
  external *powm* : $t \rightarrow t \rightarrow t \rightarrow t \rightarrow t$ = "ml_mpz2_powm"
  external *powm_ui* : $t \rightarrow t \rightarrow int \rightarrow t \rightarrow t$ = "ml_mpz2_powm_ui"
  external *sqrt* : $t \rightarrow t \rightarrow t$ = "ml_mpz2_sqrt"
  external *add_ui* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_add_ui"
  external *sub_ui* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_sub_ui"
  external *mul_ui* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_mul_ui"
  external *neg* : $t \rightarrow t \rightarrow t$ = "ml_mpz2_neg"
  external *abs* : $t \rightarrow t \rightarrow t$ = "ml_mpz2_abs"
  external *mul2exp* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_mul2exp"
  external *fac_ui* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_fac_ui"
  external *tdiv_q* : $t \rightarrow t \rightarrow t \rightarrow t$ = "ml_mpz2_tdiv_q"
  external *tdiv_r* : $t \rightarrow t \rightarrow t \rightarrow t$ = "ml_mpz2_tdiv_r"
  external *fdiv_q* : $t \rightarrow t \rightarrow t \rightarrow t$ = "ml_mpz2_fdiv_q"
  external *fdiv_r* : $t \rightarrow t \rightarrow t \rightarrow t$ = "ml_mpz2_fdiv_r"
  external *cdiv_q* : $t \rightarrow t \rightarrow t \rightarrow t$ = "ml_mpz2_cdiv_q"
  external *cdiv_r* : $t \rightarrow t \rightarrow t \rightarrow t$ = "ml_mpz2_cdiv_r"
  external *tdiv_q_2exp* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_tdiv_q_2exp"
  external *tdiv_r_2exp* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_tdiv_r_2exp"
  external *fdiv_q_2exp* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_fdiv_q_2exp"
  external *fdiv_r_2exp* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_fdiv_r_2exp"
  external *tdiv_q_ui* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_tdiv_q_ui"
  external *tdiv_r_ui* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_tdiv_r_ui"
  external *fdiv_q_ui* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_fdiv_q_ui"
  external *fdiv_r_ui* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_fdiv_r_ui"
  external *cdiv_q_ui* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_cdiv_q_ui"
  external *cdiv_r_ui* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_cdiv_r_ui"
  external *dmod* : $t \rightarrow t \rightarrow t \rightarrow t$ = "ml_mpz2_mod"
  external *dmod_ui* : $t \rightarrow t \rightarrow int \rightarrow t$ = "ml_mpz2_mod_ui"
  external *divexact* : $t \rightarrow t \rightarrow t \rightarrow t$ = "ml_mpz2_divexact"

```
  external band : t → t → t → t = "ml_mpz2_and"
  external bxor : t → t → t → t = "ml_mpz2_ior"
  external bnot : t → t → t = "ml_mpz2_com"
  external invert : t → t → t → t = "ml_mpz2_invert"
  external setbit : t → int → t = "ml_mpz2_setbit"
  external clrbit : t → int → t = "ml_mpz2_clrbit"
end

module Z : sig
  type t = Z2.t
  external copy : t → t = "ml_mpz_copy"
  external from_int : int → t = "ml_mpz_from_int"
  external from_string : string → int → t = "ml_mpz_from_string"
  external from_float : float → t = "ml_mpz_from_float"
  external int_from : t → int = "ml_int_from_mpz"
  external float_from : t → float = "ml_float_from_mpz"
  external string_from : t → int → string = "ml_string_from_mpz"
  external add : t → t → t = "ml_mpz_add"
  external sub : t → t → t = "ml_mpz_sub"
  external mul : t → t → t = "ml_mpz_mul"
  external pow_ui : t → int → t = "ml_mpz_pow_ui"
  external pow_ui_ui : int → int → t = "ml_mpz_ui_pow_ui"
  external powm : t → t → t → t = "ml_mpz_powm"
  external powm_ui : t → int → t → t = "ml_mpz_powm_ui"
  external sqrt : t → t = "ml_mpz_sqrt"
  external sqrtrem : t → t × t = "ml_mpz_sqrtrem"
  external perfect_square : t → bool = "ml_mpz_perfect_square_p"
  external add_ui : t → int → t = "ml_mpz_add_ui"
  external sub_ui : t → int → t = "ml_mpz_sub_ui"
  external mul_ui : t → int → t = "ml_mpz_mul_ui"
  external neg : t → t = "ml_mpz_neg"
  external abs : t → t = "ml_mpz_abs"
  external mul2exp : t → int → t = "ml_mpz_mul2exp"
  external fac_ui : int → t = "ml_mpz_fac_ui"
  external tdiv_q : t → t → t = "ml_mpz_tdiv_q"
  external tdiv_r : t → t → t = "ml_mpz_tdiv_r"
  external tdiv_qr : t → t → t × t = "ml_mpz_tdiv_qr"
  external fdiv_q : t → t → t = "ml_mpz_fdiv_q"
  external fdiv_r : t → t → t = "ml_mpz_fdiv_r"
  external fdiv_qr : t → t → t × t = "ml_mpz_fdiv_qr"
  external cdiv_q : t → t → t = "ml_mpz_cdiv_q"
  external cdiv_r : t → t → t = "ml_mpz_cdiv_r"
  external cdiv_qr : t → t → t × t = "ml_mpz_cdiv_qr"
  external tdiv_q_2exp : t → int → t = "ml_mpz_tdiv_q_2exp"
  external tdiv_r_2exp : t → int → t = "ml_mpz_tdiv_r_2exp"
```

```
external fdiv_q_2exp : t → int → t = "ml_mpz_fdiv_q_2exp"
external fdiv_r_2exp : t → int → t = "ml_mpz_fdiv_r_2exp"
external tdiv_q_ui : t → int → t = "ml_mpz_tdiv_q_ui"
external tdiv_r_ui : t → int → t = "ml_mpz_tdiv_r_ui"
external tdiv_qr_ui : t → int → t × t = "ml_mpz_tdiv_qr_ui"
external fdiv_q_ui : t → int → t = "ml_mpz_fdiv_q_ui"
external fdiv_r_ui : t → int → t = "ml_mpz_fdiv_r_ui"
external fdiv_qr_ui : t → int → t × t = "ml_mpz_fdiv_qr_ui"
external cdiv_q_ui : t → int → t = "ml_mpz_cdiv_q_ui"
external cdiv_r_ui : t → int → t = "ml_mpz_cdiv_r_ui"
external cdiv_qr_ui : t → int → t × t = "ml_mpz_cdiv_qr_ui"
external dmod : t → t → t = "ml_mpz_mod"
external dmod_ui : t → int → t = "ml_mpz_mod_ui"
external divexact : t → t → t = "ml_mpz_divexact"
external cmp : t → t → int = "ml_mpz_cmp"
external cmp_si : t → int → int = "ml_mpz_cmp_si"
external sgn : t → int = "ml_mpz_sgn"
external band : t → t → t = "ml_mpz_and"
external bxor : t → t → t = "ml_mpz_ior"
external bnot : t → t = "ml_mpz_com"
external popcount : t → int = "ml_mpz_popcount"
external hamdist : t → t → int = "ml_mpz_hamdist"
external scan0 : t → int → int = "ml_mpz_scan0"
external scan1 : t → int → int = "ml_mpz_scan1"
external is_probab_prime : t → int → bool = "ml_mpz_probab_prime_p"
external is_perfect_square : t → bool = "ml_mpz_perfect_square_p"
external gcd : t → t → t = "ml_mpz_gcd"
external gcdext : t → t → t × t × t = "ml_mpz_gcdext"
external invert : t → t → t = "ml_mpz_invert"
external jacobi : t → t → int = "ml_mpz_jacobi"
external legendre : t → t → int = "ml_mpz_legendre"
val setbit : t → int → t
val clrbit : t → int → t

module Infixes : sig
  external ( +! ) : t → t → t = "ml_mpz_add"
  external ( −! ) : t → t → t = "ml_mpz_sub"
  external ( *! ) : t → t → t = "ml_mpz_mul"
  external ( %! ) : t → t → t = "ml_mpz_fdiv_r"
  val ( <! ) : t → t → bool
  val ( <=! ) : t → t → bool
  val ( =! ) : t → t → bool
  val ( >=! ) : t → t → bool
  val ( >! ) : t → t → bool
  val ( <>! ) : t → t → bool
```

```
  end
end

module Q : sig
  type t
  external copy : t -> t = "ml_mpq_copy"
  external from_ints : int -> int -> t = "ml_mpq_from_ints"
  external from_z : Z.t -> t = "ml_mpq_from_z"
  external float_from : t -> float = "ml_float_from_mpq"
  external add : t -> t -> t = "ml_mpq_add"
  external sub : t -> t -> t = "ml_mpq_sub"
  external div : t -> t -> t = "ml_mpq_div"
  external mul : t -> t -> t = "ml_mpq_mul"
  external neg : t -> t = "ml_mpq_neg"
  external inv : t -> t = "ml_mpq_inv"
  external get_num : t -> Z.t = "ml_mpq_get_num"
  external get_den : t -> Z.t = "ml_mpq_get_den"
  external cmp : t -> t -> int = "ml_mpq_cmp"
  external sgn : t -> int = "ml_mpq_sgn"
  external equal : t -> t -> bool = "ml_mpq_equal"
  val from_zs : Z.t -> Z.t -> t

  module Infixes : sig
    external ( +/ ) : t -> t -> t = "ml_mpq_add"
    external ( -/ ) : t -> t -> t = "ml_mpq_sub"
    external ( */ ) : t -> t -> t = "ml_mpq_mul"
    external ( // ) : t -> t -> t = "ml_mpq_div"
    val ( </ ) : t -> t -> bool
    val ( <=/ ) : t -> t -> bool
    external ( =/ ) : t -> t -> bool = "ml_mpq_equal"
    val ( >=/ ) : t -> t -> bool
    val ( >/ ) : t -> t -> bool
    val ( <>/ ) : t -> t -> bool
  end
end

module Q2 : sig
  type t = Q.t
  external copy : t -> t -> t = "ml_mpq2_copy"
  external from_ints : t -> int -> int -> t = "ml_mpq2_from_ints"
  external from_z : t -> Z.t -> t = "ml_mpq2_from_z"
  external add : t -> t -> t -> t = "ml_mpq2_add"
  external sub : t -> t -> t -> t = "ml_mpq2_sub"
  external div : t -> t -> t -> t = "ml_mpq2_div"
  external mul : t -> t -> t -> t = "ml_mpq2_mul"
  external neg : t -> t -> t = "ml_mpq2_neg"
```

```
  external inv : t → t → t = "ml_mpq2_inv"
  external get_num : Z.t → t → Z.t = "ml_mpq2_get_num"
  external get_den : Z.t → t → Z.t = "ml_mpq2_get_den"
  val from_zs : t → Z.t → Z.t → t
end
```

Module Gmp.ml

```
module Z2 = struct
  type t

  external copy : t → t → t = "ml_mpz2_copy"
  external from_int : t → int → t = "ml_mpz2_from_int"
  external from_string : t → string → int → t = "ml_mpz2_from_string"
  external from_float : t → float → t = "ml_mpz2_from_float"

  external add : t → t → t → t = "ml_mpz2_add"
  external sub : t → t → t → t = "ml_mpz2_sub"
  external mul : t → t → t → t = "ml_mpz2_mul"

  external pow_ui : t → t → int → t = "ml_mpz2_pow_ui"
  external pow_ui_ui : t → int → int → t = "ml_mpz2_ui_pow_ui"
  external powm : t → t → t → t → t = "ml_mpz2_powm"
  external powm_ui : t → t → int → t → t = "ml_mpz2_powm_ui"

  external sqrt : t → t → t = "ml_mpz2_sqrt"

  external add_ui : t → t → int → t = "ml_mpz2_add_ui"
  external sub_ui : t → t → int → t = "ml_mpz2_sub_ui"
  external mul_ui : t → t → int → t = "ml_mpz2_mul_ui"

  external neg : t → t → t = "ml_mpz2_neg"
  external abs : t → t → t = "ml_mpz2_abs"

  external mul2exp : t → t → int → t = "ml_mpz2_mul2exp"
  external fac_ui : t → t → int → t = "ml_mpz2_fac_ui"

  external tdiv_q : t → t → t → t = "ml_mpz2_tdiv_q"
  external tdiv_r : t → t → t → t = "ml_mpz2_tdiv_r"

  external fdiv_q : t → t → t → t = "ml_mpz2_fdiv_q"
  external fdiv_r : t → t → t → t = "ml_mpz2_fdiv_r"

  external cdiv_q : t → t → t → t = "ml_mpz2_cdiv_q"
  external cdiv_r : t → t → t → t = "ml_mpz2_cdiv_r"

  external tdiv_q_2exp : t → t → int → t = "ml_mpz2_tdiv_q_2exp"
  external tdiv_r_2exp : t → t → int → t = "ml_mpz2_tdiv_r_2exp"
  external fdiv_q_2exp : t → t → int → t = "ml_mpz2_fdiv_q_2exp"
  external fdiv_r_2exp : t → t → int → t = "ml_mpz2_fdiv_r_2exp"
```

```
  external tdiv_q_ui : t → t → int → t = "ml_mpz2_tdiv_q_ui"
  external tdiv_r_ui : t → t → int → t = "ml_mpz2_tdiv_r_ui"

  external fdiv_q_ui : t → t → int → t = "ml_mpz2_fdiv_q_ui"
  external fdiv_r_ui : t → t → int → t = "ml_mpz2_fdiv_r_ui"

  external cdiv_q_ui : t → t → int → t = "ml_mpz2_cdiv_q_ui"
  external cdiv_r_ui : t → t → int → t = "ml_mpz2_cdiv_r_ui"

  external dmod : t → t → t → t = "ml_mpz2_mod"
  external dmod_ui : t → t → int → t = "ml_mpz2_mod_ui"

  external divexact : t → t → t → t = "ml_mpz2_divexact"

  external band : t → t → t → t = "ml_mpz2_and"
  external bxor : t → t → t → t = "ml_mpz2_ior"
  external bnot : t → t → t = "ml_mpz2_com"

  external invert : t → t → t → t = "ml_mpz2_invert"

  external setbit : t → int → t = "ml_mpz2_setbit"
  external clrbit : t → int → t = "ml_mpz2_clrbit"
end

module Z = struct
  type t = Z2.t

  external copy : t → t = "ml_mpz_copy"
  external from_int : int → t = "ml_mpz_from_int"
  external from_string : string → int → t = "ml_mpz_from_string"
  external from_float : float → t = "ml_mpz_from_float"

  external int_from : t → int = "ml_int_from_mpz"
  external float_from : t → float = "ml_float_from_mpz"
  external string_from : t → int → string= "ml_string_from_mpz"

  external add : t → t → t = "ml_mpz_add"
  external sub : t → t → t = "ml_mpz_sub"
  external mul : t → t → t = "ml_mpz_mul"

  external pow_ui : t → int → t = "ml_mpz_pow_ui"
  external pow_ui_ui : int → int → t = "ml_mpz_ui_pow_ui"
  external powm : t → t → t → t = "ml_mpz_powm"
  external powm_ui : t → int → t → t = "ml_mpz_powm_ui"

  external sqrt : t → t = "ml_mpz_sqrt"
  external sqrtrem : t → t × t = "ml_mpz_sqrtrem"
  external perfect_square : t → bool = "ml_mpz_perfect_square_p"

  external add_ui : t → int → t = "ml_mpz_add_ui"
  external sub_ui : t → int → t = "ml_mpz_sub_ui"
  external mul_ui : t → int → t = "ml_mpz_mul_ui"
```

external neg : $t \to t$ = "ml_mpz_neg"
external abs : $t \to t$ = "ml_mpz_abs"

external $mul2exp$: $t \to int \to t$ = "ml_mpz_mul2exp"
external fac_ui : $int \to t$ = "ml_mpz_fac_ui"

external $tdiv\_q$: $t \to t \to t$ = "ml_mpz_tdiv_q"
external $tdiv\_r$: $t \to t \to t$ = "ml_mpz_tdiv_r"
external $tdiv\_qr$: $t \to t \to t \times t$ = "ml_mpz_tdiv_qr"

external $fdiv\_q$: $t \to t \to t$ = "ml_mpz_fdiv_q"
external $fdiv\_r$: $t \to t \to t$ = "ml_mpz_fdiv_r"
external $fdiv\_qr$: $t \to t \to t \times t$ = "ml_mpz_fdiv_qr"

external $cdiv\_q$: $t \to t \to t$ = "ml_mpz_cdiv_q"
external $cdiv\_r$: $t \to t \to t$ = "ml_mpz_cdiv_r"
external $cdiv\_qr$: $t \to t \to t \times t$ = "ml_mpz_cdiv_qr"

external $tdiv\_q\_2exp$: $t \to int \to t$ = "ml_mpz_tdiv_q_2exp"
external $tdiv\_r\_2exp$: $t \to int \to t$ = "ml_mpz_tdiv_r_2exp"
external $fdiv\_q\_2exp$: $t \to int \to t$ = "ml_mpz_fdiv_q_2exp"
external $fdiv\_r\_2exp$: $t \to int \to t$ = "ml_mpz_fdiv_r_2exp"

external $tdiv\_q\_ui$: $t \to int \to t$ = "ml_mpz_tdiv_q_ui"
external $tdiv\_r\_ui$: $t \to int \to t$ = "ml_mpz_tdiv_r_ui"
external $tdiv\_qr\_ui$: $t \to int \to t \times t$ = "ml_mpz_tdiv_qr_ui"

external $fdiv\_q\_ui$: $t \to int \to t$ = "ml_mpz_fdiv_q_ui"
external $fdiv\_r\_ui$: $t \to int \to t$ = "ml_mpz_fdiv_r_ui"
external $fdiv\_qr\_ui$: $t \to int \to t \times t$ = "ml_mpz_fdiv_qr_ui"

external $cdiv\_q\_ui$: $t \to int \to t$ = "ml_mpz_cdiv_q_ui"
external $cdiv\_r\_ui$: $t \to int \to t$ = "ml_mpz_cdiv_r_ui"
external $cdiv\_qr\_ui$: $t \to int \to t \times t$ = "ml_mpz_cdiv_qr_ui"

external $dmod$: $t \to t \to t$ = "ml_mpz_mod"
external $dmod\_ui$: $t \to int \to t$ = "ml_mpz_mod_ui"

external $divexact$: $t \to t \to t$ = "ml_mpz_divexact"

external cmp : $t \to t \to int$ = "ml_mpz_cmp"
external cmp_si : $t \to int \to int$ = "ml_mpz_cmp_si"
external sgn : $t \to int$ = "ml_mpz_sgn"

external $band$: $t \to t \to t$ = "ml_mpz_and"
external $bxor$: $t \to t \to t$ = "ml_mpz_ior"
external $bnot$: $t \to t$ = "ml_mpz_com"

external $popcount$: $t \to int$ ="ml_mpz_popcount"
external $hamdist$: $t \to t \to int$ ="ml_mpz_hamdist"

external $scan0$: $t \to int \to int$ = "ml_mpz_scan0"
external $scan1$: $t \to int \to int$ = "ml_mpz_scan1"

```
  external is_probab_prime : t -> int -> bool = "ml_mpz_probab_prime_p"
  external is_perfect_square : t -> bool = "ml_mpz_perfect_square_p"
  external gcd : t -> t -> t = "ml_mpz_gcd"
  external gcdext : t -> t -> t * t * t = "ml_mpz_gcdext"
  external invert : t -> t -> t = "ml_mpz_invert"
  external jacobi : t -> t -> int = "ml_mpz_jacobi"
  external legendre : t -> t -> int = "ml_mpz_legendre"

  let setbit x y = Z2.setbit (copy x) y
  let clrbit x y = Z2.clrbit (copy x) y

  module Infixes = struct
    external ( +! ) : t -> t -> t = "ml_mpz_add"
    external ( -! ) : t -> t -> t = "ml_mpz_sub"
    external ( *! ) : t -> t -> t = "ml_mpz_mul"
    external ( /! ) : t -> t -> t = "ml_mpz_fdiv_q"
    external ( %! ) : t -> t -> t = "ml_mpz_fdiv_r"
    let ( <! ) x y = (cmp x y) < 0
    let ( <=! ) x y = (cmp x y) <= 0
    let ( =! ) x y = (cmp x y) = 0
    let ( >=! ) x y = (cmp x y) >= 0
    let ( >! ) x y = (cmp x y) > 0
    let ( <>! ) x y = (cmp x y) <> 0
  end
end

module Q = struct
  type t

  external copy : t -> t = "ml_mpq_copy"
  external from_ints : int -> int -> t = "ml_mpq_from_ints"
  external from_z : Z.t -> t = "ml_mpq_from_z"

  external float_from : t -> float = "ml_float_from_mpq"

  external add : t -> t -> t = "ml_mpq_add"
  external sub : t -> t -> t = "ml_mpq_sub"
  external div : t -> t -> t = "ml_mpq_div"
  external mul : t -> t -> t = "ml_mpq_mul"

  external neg : t -> t = "ml_mpq_neg"
  external inv : t -> t = "ml_mpq_inv"

  external get_num : t -> Z.t = "ml_mpq_get_num"
  external get_den : t -> Z.t = "ml_mpq_get_den"

  external cmp : t -> t -> int = "ml_mpq_cmp"
  external sgn : t -> int = "ml_mpq_sgn"
  external equal : t -> t -> bool = "ml_mpq_equal"
```

```
  let from_zs num den = div (from_z num) (from_z den)

  module Infixes = struct
    external ( +/ ) : t -> t -> t = "ml_mpq_add"
    external ( -/ ) : t -> t -> t = "ml_mpq_sub"
    external ( */ ) : t -> t -> t = "ml_mpq_mul"
    external ( // ) : t -> t -> t = "ml_mpq_div"
    let ( </ ) x y = (cmp x y) < 0
    let ( <=/ ) x y = (cmp x y) <= 0
    external ( =/ ) : t -> t -> bool = "ml_mpq_equal"
    let ( >=/ ) x y = (cmp x y) >= 0
    let ( >/ ) x y = (cmp x y) > 0
    let ( <>/ ) x y = not (equal x y)
  end
end

module Q2 = struct
  type t = Q.t

  external copy : t -> t -> t = "ml_mpq2_copy"
  external from_ints : t -> int -> int -> t = "ml_mpq2_from_ints"
  external from_z : t -> Z.t -> t = "ml_mpq2_from_z"

  external add : t -> t -> t -> t = "ml_mpq2_add"
  external sub : t -> t -> t -> t = "ml_mpq2_sub"
  external div : t -> t -> t -> t = "ml_mpq2_div"
  external mul : t -> t -> t -> t = "ml_mpq2_mul"

  external neg : t -> t -> t = "ml_mpq2_neg"
  external inv : t -> t -> t = "ml_mpq2_inv"

  external get_num : Z.t -> t -> Z.t = "ml_mpq2_get_num"
  external get_den : Z.t -> t -> Z.t = "ml_mpq2_get_den"

  let from_zs r num den = div r (Q.from_z num) (Q.from_z den)
end
```

6.4 Module mpa

Interface for module Mpa.mli

Author: Jean-Christophe Filliatre, Harald Ruess
* Multi-precision arithmetic. The purpose of this module is to abstract the necessary arithmetic functions of any multi-precision package such as Ocaml's bignums, GNU MP etc in order to switch easily between packages.
* 6 iterators

module Z : sig

type t

val $zero$: t
val one : t
val two : t

val add : $t \rightarrow t \rightarrow t$
val sub : $t \rightarrow t \rightarrow t$
val $succ$: $t \rightarrow t$
val $mult$: $t \rightarrow t \rightarrow t$
val $divexact$: $t \rightarrow t \rightarrow t$
val $expt$: $t \rightarrow int \rightarrow t$

val $equal$: $t \rightarrow t \rightarrow bool$
val $compare$: $t \rightarrow t \rightarrow int$
val lt : $t \rightarrow t \rightarrow bool$
val le : $t \rightarrow t \rightarrow bool$
val gt : $t \rightarrow t \rightarrow bool$
val ge : $t \rightarrow t \rightarrow bool$

val of_int : $int \rightarrow t$
val gcd : $t \rightarrow t \rightarrow t$ (∗s Greatest Common Divisor. ∗)
val lcm : $t \rightarrow t \rightarrow t$ (∗s Least Common Multiple. ∗)
val pow : $int \rightarrow int \rightarrow t$

val to_string : $t \rightarrow string$

val pp : $Format.formatter \rightarrow t \rightarrow unit$
end

* 6 iterators

module Q : sig

type t

val $negone$: t
val $zero$: t
val one : t
val two : t
val is_zero : $t \rightarrow bool$
val is_one : $t \rightarrow bool$
val is_negone : $t \rightarrow bool$
val of_int : $int \rightarrow t$

```
  val of_ints : int → int → t

  val add : t → t → t
  val sub : t → t → t
  val minus : t → t
  val mult : t → t → t
  val div : t → t → t
  val inv : t → t
  val expt : t → int → t

  val floor : t → Z.t
  val ceil : t → Z.t

  val compare : t → t → int
  val equal : t → t → bool
  val lt : t → t → bool
  val le : t → t → bool
  val gt : t → t → bool
  val ge : t → t → bool

  type cmp = Equal | Greater | Less

  val cmp : t → t → cmp

  val sign : t → Sign.t

  val denominator : t → Z.t
  val numerator : t → Z.t

  val is_integer : t → bool
  val to_z : t → Z.t
  val of_z : Z.t → t

  val hash : t → int

  val to_string : t → string
  val of_string : string → t

  val pp : Format.formatter → t → unit
end
```

Module Mpa.ml

48. Multi-precision arithmetic.

```
open Gmp

module Z = struct

  type t = Z.t

  let of_int = Z.from_int
```

```
  let zero = of_int 0
  let one = of_int 1
  let two = of_int 2
  let add = Z.add
  let sub = Z.sub
  let succ a = add a one
  let mult = Z.mul
  let divexact = Z.divexact
  let gcd = Z.gcd
  let lcm a b = Z.divexact (Z.mul a b) (Z.gcd a b)
  let pow = Z.pow_ui_ui

  let rec expt x n =
    if n = 0 then one else mult x (expt x (n - 1))

  let compare = Z.cmp
  let equal x y = Z.cmp x y ≡ 0
  let lt x y = Z.cmp x y < 0
  let le x y = Z.cmp x y ≤ 0
  let gt x y = Z.cmp x y > 0
  let ge x y = Z.cmp x y ≥ 0

  let to_string z = Z.string_from z 10

  let pp fmt x = Format.fprintf fmt "%s" (Z.string_from x 10)
end

module Q = struct

  type t = Q.t

  let of_int n = Q.from_ints n 1
  let of_ints = Q.from_ints

  let zero = of_int 0
  let one = of_int 1
  let two = of_int 2
  let negone = of_int (−1)

  let add = Q.add
  let sub = Q.sub
  let minus = Q.neg
  let mult = Q.mul
  let div = Q.div
  let inv = Q.inv

  let rec expt a n =
    assert(n ≥ 0);
    if n = 0 then one
    else mult a (expt a (n - 1))
```

```
  let floor x = Gmp.Z.fdiv_q (Q.get_num x) (Q.get_den x)
  let ceil x = Gmp.Z.cdiv_q (Q.get_num x) (Q.get_den x)

  let compare = Q.cmp
  let equal = Q.equal
  let is_zero x = Q.equal zero x
  let is_one x = Q.equal one x
  let is_negone x = Q.equal (Q.neg one) x
  let lt x y = Q.cmp x y < 0
  let le x y = Q.cmp x y ≤ 0
  let gt x y = Q.cmp x y > 0
  let ge x y = Q.cmp x y ≥ 0

  type cmp = Equal | Greater | Less

  let cmp x y =
    let b = compare x y in
    if b ≡ 0 then Equal else if b > 0 then Greater else Less

  let sign x =
    let b = compare x zero in
    if b ≡ 0 then Sign.Zero else if b > 0 then Sign.Pos else Sign.Neg

  let denominator = Q.get_den
  let numerator = Q.get_num

  let is_integer q = (Gmp.Z.cmp_si (Q.get_den q) 1) = 0
  let to_z = Q.get_num
  let of_z = Q.from_z

  let hash = Hashtbl.hash

  let to_string q =
    let d = Q.get_den q in
    if Gmp.Z.cmp_si d 1 ≡ 0 then
      Gmp.Z.string_from (Q.get_num q) 10
    else
      (Gmp.Z.string_from (Q.get_num q) 10) ^ "/" ^ (Gmp.Z.string_from d 10)

  let of_string s =
    try
      let k = String.index s '/' in
      let l = String.length s in
      let n = Gmp.Z.from_string (String.sub s 0 k) 10 in
      let d = Gmp.Z.from_string (String.sub s (succ k) (l - k - 1)) 10 in
      Q.from_zs n d
    with Not_found →
      Q.from_z (Gmp.Z.from_string s 10)

  let pp fmt x = Format.fprintf fmt "%s" (to_string x)
end
```

6.5 Module binrel

Interface for module Binrel.mli

type α *t* =
  | *Same*
  | *Disjoint*
  | *Sub*
  | *Super*
  | *Singleton* of $Mpa.Q.t$
  | *Overlap* of α

Module Binrel.ml

type α *t* =
  | *Same*
  | *Disjoint*
  | *Sub*
  | *Super*
  | *Singleton* of $Mpa.Q.t$
  | *Overlap* of α

6.6 Module hashcons

Interface for module Hashcons.mli

49. Module *Hashcons*: hashconsing over types with equality.

50. This module implements hashconsing of types with equalities by injecting elements a of this type into a record consisting of the node a itself, a unique integer tag, and a hash key for a.

The main advantage of hashconsing is that equality is reduced to a constant time operation. On the other hand, the penalty to pay is that every entity has to be hashconsed. Besides the time overhead there is also a space overhead for the storage of additional information in hashed elements and the use of global hash tables. These elements are never being garbage collected, since they are all kept in a hash table.

type α *hashed* = {
  hkey : *int*;
  tag : *int*;
  node : α

}

51. Equality for hashconsed entities reduces to identity, and can thus be performed in constant time.

val $(===)$: α *hashed* $\rightarrow$ α *hashed* $\rightarrow$ *bool*

52. Disequality $a\ =/=\ b$ is defined as $\neg(a\ ===\ b)$.

val $(=/=)$: α *hashed* $\rightarrow$ α *hashed* $\rightarrow$ *bool*

The input signature of the functor *Hashcons.Make*. t is the type of the elements to be hashconsed. *equal* specifies the equality relation for hashconsing, and *hash* is a function for computing hash keys. Example: a suitable hashc function is often the generic hash function *hash*.

```
module type HashedType =
  sig
    type t
    val equal : t → t → bool
    val hash : t → int
  end
```

```
module type S =
  sig
    type key
```

53. The type of hash tables for hashconsing elements of type *key*.

```
    type t
```

54. *create* n creates a new, empty hash table, with initial size n. For best results, n should be on the order of the expected number of elements that will be in the table. The table grows as needed, so n is just an initial guess.

```
    val create : int → t
```

55. Empty a hash table.

```
    val clear : t → unit
```

56. Given a table t and a node a, *hashcons* t a returns a hashconsing record for a with a unique tag.

```
    val hashcons : t → key → key hashed
```

mem tbl x checks if x is bound in *tbl*.

```
    val mem : t → key → bool
```

57. *iter* f t applies f in turn to all hashconsed elements of t. The order in which the elements of t are presented to f is unspecified.

```
  val iter : (key hashed → unit) → t → unit
```

58. Prints on standard output some statistics for hash table t such as percentige of used entries, and maximum bucket length.

```
  val stat : t → unit
end
```

59. Functor building an implementation of the hashcons structure given a structure of signature *HashedType*.

```
module Make(H : HashedType) : (S with type key = H.t)
```

Module Hashcons.ml

```
type α hashed = {
  hkey : int;
  tag : int;
  node : α }

type α t = {
  mutable size : int; (* current size *)
  mutable data : α bucketlist array } (* the buckets *)

and α bucketlist =
  | Empty
  | Cons of α hashed × α bucketlist

let create initial_size =
  let s = if initial_size < 1 then 1 else initial_size in
  let s = if s > Sys.max_array_length then Sys.max_array_length else s in
  { size = 0; data = Array.make s Empty }

let clear h =
  for i = 0 to Array.length h.data − 1 do
    h.data.(i) ← Empty
  done

let resize tbl =
  let odata = tbl.data in
  let osize = Array.length odata in
  let nsize = 2 × osize + 1 in
  let nsize =
    if nsize ≤ Sys.max_array_length then nsize else Sys.max_array_length
  in
  if nsize ≠ osize then begin
    let ndata = Array.create nsize Empty in
    let rec insert_bucket = function
```

```
        Empty → ()
      | Cons(data, rest) →
          insert_bucket rest; (* preserve original order of elements *)
          let nidx = data.hkey mod nsize in
          ndata.(nidx) ← Cons(data, ndata.(nidx)) in
    for i = 0 to osize − 1 do
      insert_bucket odata.(i)
    done;
    tbl.data ← ndata
  end

let array_too_small h = Array.length h.data < h.size

let add h hkey info =
  let i = hkey mod (Array.length h.data) in
  let bucket = Cons(info, h.data.(i)) in
  h.data.(i) ← bucket;
  h.size ← h.size + 1;
  if array_too_small h then resize h

let find h key hkey =
  match h.data.(hkey mod (Array.length h.data)) with
    Empty → raise Not_found
  | Cons(d1, rest1) →
      if key = d1.node then d1 else
      match rest1 with
        Empty → raise Not_found
      | Cons(d2, rest2) →
          if key = d2.node then d2 else
          match rest2 with
            Empty → raise Not_found
          | Cons(d3, rest3) →
              if key = d3.node then d3 else begin
                let rec find = function
                    Empty →
                      raise Not_found
                  | Cons(d, rest) →
                      if key = d.node then d else find rest
                in find rest3
              end

let gentag =
  let r = ref 0 in
  fun () → incr r; !r

let hashcons h node =
  let hkey = Hashtbl.hash_param 10 100 node in
```

```
  try
    find h node hkey
  with Not_found →
    let hnode = { hkey = hkey; tag = gentag(); node = node } in
    add h hkey hnode;
    hnode

let mem h node =
  let hkey = Hashtbl.hash_param 10 100 node in
  try
    let _ = find h node hkey in
    true
  with
      Not_found → false

let (===) = (≡)

let (=/=) x y = ¬(x ≡ y)

let iter f h =
  let rec bucket_iter = function
    | Empty → ()
    | Cons (x, l) → f x; bucket_iter l
  in
  Array.iter bucket_iter h.data

let rec bucketlist_length = function
  | Empty → 0
  | Cons (_, bl) → succ (bucketlist_length bl)

let stat h =
  let d = h.data in
  let size = Array.length d in
  let ne = ref 0 in
  let m = ref 0 in
  let t = ref 0 in
  for i = 0 to size − 1 do
    let n = bucketlist_length d.(i) in
    t := !t + n;
    if n > 0 then incr ne;
    if n > !m then m := n
  done;
  let p = 100 × !ne / size in
  Printf.printf
    "%6d val, used = %6d / %6d (%2d%%), max length = %6d\n"
    !t !ne size p !m
```

Functorial interface

```
module type HashedType =
  sig
    type t
    val equal : t → t → bool
    val hash : t → int
  end

module type S =
  sig
    type key
    type t
    val create : int → t
    val clear : t → unit
    val hashcons : t → key → key hashed
    val mem : t → key → bool
    val iter : (key hashed → unit) → t → unit
    val stat : t → unit
  end

module Make(H : HashedType) : (S with type key = H.t) =
  struct
    type key = H.t
    type hashtbl = key t
    type t = hashtbl

    let create = create
    let clear = clear

    let add h hkey info =
      let i = hkey mod (Array.length h.data) in
      let bucket = Cons(info, h.data.(i)) in
      h.data.(i) ← bucket;
      h.size ← h.size + 1;
      if array_too_small h then resize h

    let find h key hkey =
      match h.data.(hkey mod (Array.length h.data)) with
        Empty → raise Not_found
      | Cons(d1, rest1) →
          if H.equal key d1.node then d1 else
          match rest1 with
            Empty → raise Not_found
          | Cons(d2, rest2) →
              if H.equal key d2.node then d2 else
              match rest2 with
                Empty → raise Not_found
              | Cons(d3, rest3) →
```

```
            if H.equal key d3.node then d3 else begin
              let rec find = function
                  Empty ->
                    raise Not_found
                | Cons(d, rest) ->
                    if H.equal key d.node then d else find rest
              in find rest3
            end

  let hashcons h node =
    let hkey = H.hash node in
    try
      find h node hkey
    with Not_found ->
      let hnode = { hkey = hkey; tag = gentag(); node = node } in
      add h hkey hnode;
      hnode

  let mem h node =
    let hkey = Hashtbl.hash_param 10 100 node in
    try
      let _ = find h node hkey in
      true
    with
        Not_found -> false

  let iter = iter

  let stat = stat
end
```

6.7 Module tools

Interface for module Tools.mli

60. The purpose of the *Tools* module is to provide some functions and types, which are used througout the code of ICS.

61. Procedures f without parameters can be registered as exit procedures by *add_to_exit* f. The registered exit procedures are then called by *do_at_exit*(). Exit procedures are usually used for displaying some statistics.

```
val add_at_exit : (unit -> unit) -> unit
val do_at_exit : unit -> unit
```

62. *add_to_reset* f registers f as a reset procedure, which are then called, one-by-one, by *do_at_reset*().

val add_at_reset : $(unit \rightarrow unit) \rightarrow unit$
val do_at_reset : $unit \rightarrow unit$

63. $utime\ f\ a$ returns not only the result of applying f to a but also the time required to compute the function.

val $utime$: $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \times float$

64. *profile str f* profiles function f, and registers an exit function which outputs the number of calls of this function, and the total time spent in this function; the argument *str* is usually just the name of the function.

val $profile$: $string \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$

65. $dynamic\_let\ (x,\ v)\ f\ a$ simulated dynamic binding of value v to the global variable x in the call of f to a.

val $dynamic\_let$: $\alpha\ ref \times \alpha \rightarrow (\beta \rightarrow \gamma) \rightarrow \beta \rightarrow \gamma$

Module Tools.ml

66. Functions to run at exit.

let $at\_exit\_functions = ref\ [\,]$

let $add\_at\_exit\ f$ =
  $at\_exit\_functions := f :: !at\_exit\_functions$

let do_at_exit () =
  $List.iter$ (fun $f \rightarrow f()$) $(List.rev\ !at\_exit\_functions)$

67. Functions to run at reset.

let $at\_reset\_functions = ref\ [\,]$

let $add\_at\_reset\ f$ =
  $at\_reset\_functions := f :: !at\_reset\_functions$

let do_at_reset () =
  $List.iter$ (fun $f \rightarrow f()$) $(List.rev\ !at\_reset\_functions)$

68. Timing functions.

open $Unix$

let $utime\ f\ x$ =
  let $u = (times()).tms\_utime$ in
  let $y = f\ x$ in
  let $ut = (times()).tms\_utime\ -.\ u$ in
  (y, ut)

```
let timers = ref [0.]
let profile str f =
  let timer = ref 0. in
  let calls = ref 0 in
  add_at_exit
    (fun () -> Format.printf "%s: utime = %f  calls = %d\n@ " str !timer !calls);
  fun x ->
    let start = (Unix.times()).tms_utime in
    let y = f x in
    let finish = (Unix.times()).tms_utime in
    timer := !timer +. (finish -. start);
    calls := !calls + 1;
    y
```

69. Simulate dynamic binding.

```
let dynamic_let (x, v) f a =
  let saved = !x in
    try
      let result = (x := v; f a) in
        x := saved;
        result
    with
      | exc ->
          x := saved;
          raise exc
```

6.8 Module status

Interface for module Status.mli

```
type α t =
  | Empty
  | Full
  | Singleton of α
  | Other
```

Module Status.ml

```
type α t =
  | Empty
  | Full
  | Singleton of α
  | Other
```

6.9 Module pretty

Interface for module Pretty.mli

70. Module *Pretty*: Pretty-printing methods.

type α *printer* $=$ *Format.formatter* $\rightarrow$ α $\rightarrow$ *unit*

val *unit* : *unit printer*

val *string* : *string printer*

val *number* : *int printer*

val *list* : α *printer* $\rightarrow$ α *list printer*

val *pair* : α *printer* $\rightarrow$ β *printer* $\rightarrow$ $(\alpha \times \beta)$ *printer*

val *tuple* : α *printer* $\rightarrow$ α *list printer*

val *infix* : α *printer* $\rightarrow$ *string* $\rightarrow$ β *printer* $\rightarrow$ $(\alpha \times \beta)$ *printer*

val *eqn* : α *printer* $\rightarrow$ $(\alpha \times \alpha)$ *printer*

val *solution* : α *printer* $\rightarrow$ $(\alpha \times \alpha)$ *list printer*

val *infixl* : α *printer* $\rightarrow$ *string* $\rightarrow$ α *list printer*

val *set* : α *printer* $\rightarrow$ α *list printer*

val *map* : α *printer* $\rightarrow$ β *printer* $\rightarrow$ $(\alpha \times \beta)$ *list printer*

71. Redirecting pretty-printer output.

val *to_stdout* : α *printer* $\rightarrow$ $(\alpha \rightarrow$ *unit*$)$
val *to_stderr* : α *printer* $\rightarrow$ $(\alpha \rightarrow$ *unit*$)$

val *to_string* : α *printer* $\rightarrow$ $(\alpha \rightarrow$ *string*$)$

Module Pretty.ml

72. type α *printer* $=$ *Format.formatter* $\rightarrow$ α $\rightarrow$ *unit*

let *unit fmt* () $=$
  Format.fprintf fmt "()"

let *string fmt str* $=$
  Format.fprintf fmt "%s" *str*

let *number fmt i* $=$
  Format.fprintf fmt "%d" *i*

let *list* (*pre*, *sep*, *post*) *pp fmt l* $=$

```
  let rec iter = function
    | [] → ()
    | [x] → pp fmt x
    | x :: l → pp fmt x; string fmt sep; Format.fprintf fmt "␣"; iter l
  in
  Format.fprintf fmt "@[%s" pre;
  iter l;
  Format.fprintf fmt "%s@]@?" post

let pair pp1 pp2 fmt (a, b) =
  Format.fprintf fmt "(";
  pp1 fmt a;
  string fmt ",";
  pp2 fmt b;
  Format.fprintf fmt ")@?"

let infix pp1 op pp2 fmt (a, b) =
  Format.fprintf fmt "";
  pp1 fmt a;
  Format.fprintf fmt "␣%s␣" op;
  pp2 fmt b;
  Format.fprintf fmt ""

let eqn pp = infix pp "=" pp

let infixl pp op =
  list ("", op, "") pp

let set pp fmt = list ("{", ",␣", "}") pp fmt

let assign pp1 pp2 fmt (x, a) =
  Format.fprintf fmt "@[";
  pp1 fmt x;
  string fmt "␣|->␣";
  pp2 fmt a;
  Format.fprintf fmt "@]"

let map pp1 pp2 fmt =
  list ("[", ";␣", "]") (assign pp1 pp2) fmt

let tuple pp = list ("(", ",␣", ")") pp

let list pp = list ("[", ";␣", "]") pp

let solution pp = list (eqn pp)
```

73. Redirecting output.

```
let to_stdout pp = pp Format.std_formatter

let to_stderr pp = pp Format.err_formatter
```

```
let to_string pp x =
  pp Format.str_formatter x;
  Format.flush_str_formatter ()
```

6.10 Module exc

Interface for module Exc.mli

74. The module *Exc* defines the exceptions for indicating inconsistencies and validity.

```
exception Inconsistent
exception Valid
exception Unsolved
```

Module Exc.ml

```
exception Inconsistent
exception Valid
exception Unsolved
```

6.11 Module bitv

Interface for module Bitv.mli

75. $Id : bitv.mli, v1.12002/04/0301 : 13 : 04ruessExp$

76. **Module Bitv**. This module implements bit vectors, as an abstract datatype t. Since bit vectors are particular cases of arrays, this module provides the same operations as the module *Array*. It also provides bitwise operations. In the following, false stands for the bit 0 and true for the bit 1.

```
type t
```

77. **Creation, access and assignment.** (*Bitv.create* n b) creates a new bit vector of length n, initialized with b. (*Bitv.init* n f) returns a fresh vector of length n, with bit number i initialized to the result of (f i). (*Bitv.set* v n b) sets the nth bit of v to the value b. (*Bitv.get* v n) returns the nth bit of v. *Bitv.length* returns the length (number of elements) of the given vector.

```
val create : int → bool → t
val init : int → (int → bool) → t
```

val $set : t \rightarrow int \rightarrow bool \rightarrow unit$

val $get : t \rightarrow int \rightarrow bool$

val $length : t \rightarrow int$

78. *max_length* is the maximum length of a bit vector (System dependent).

val $max\_length : int$

79. Copies and concatenations. (*Bitv.copy v*) returns a copy of v, that is, a fresh vector containing the same elements as v. (*Bitv.append v1 v2*) returns a fresh vector containing the concatenation of the vectors *v1* and *v2*. *Bitv.concat* is similar to *Bitv.append*, but catenates a list of vectors.

val $copy : t \rightarrow t$

val $append : t \rightarrow t \rightarrow t$

val $concat : t\ list \rightarrow t$

80. Sub-vectors and filling. (*Bitv.sub v start len*) returns a fresh vector of length *len*, containing the bits number *start* to $start + len - 1$ of vector v. Raise *Invalid_argument* `"Bitv.sub"` if *start* and *len* do not designate a valid subvector of v; that is, if $start < 0$, or $len < 0$, or $start + len > Bitv.length\ a$.

(*Bitv.fill v ofs len b*) modifies the vector v in place, storing b in elements number *ofs* to $ofs + len - 1$. Raise *Invalid_argument* `"Bitv.fill"` if *ofs* and *len* do not designate a valid subvector of v.

(*Bitv.blit v1 o1 v2 o2 len*) copies *len* elements from vector *v1*, starting at element number *o1*, to vector *v2*, starting at element number *o2*. It *does not work* correctly if *v1* and *v2* are the same vector with the source and destination chunks overlapping. Raise *Invalid_argument* `"Bitv.blit"` if *o1* and *len* do not designate a valid subvector of *v1*, or if *o2* and *len* do not designate a valid subvector of *v2*.

val $sub : t \rightarrow int \rightarrow int \rightarrow t$

val $fill : t \rightarrow int \rightarrow int \rightarrow bool \rightarrow unit$

val $blit : t \rightarrow int \rightarrow t \rightarrow int \rightarrow int \rightarrow unit$

81. Iterators. (*Bitv.iter f v*) applies function f in turn to all the elements of v. Given a function f, (*Bitv.map f v*) applies f to all the elements of v, and builds a vector with the results returned by f. *Bitv.iteri* and *Bitv.mapi* are similar to *Bitv.iter* and *Bitv.map* respectively, but the function is applied to the index of the element as first argument, and the element itself as second argument.

(*Bitv.fold_left f x v*) computes f (... (f (f x ($get\ v\ 0$)) ($get\ v\ 1$)) ...) ($get\ v\ (n-1)$), where n is the length of the vector v.

(*Bitv.fold_right f a x*) computes f ($get\ v\ 0$) (f ($get\ v\ 1$) (... (f ($get\ v\ (n-1)$) x) ...)), where n is the length of the vector v.

val $iter : (bool \rightarrow unit) \rightarrow t \rightarrow unit$

val map : $(bool \rightarrow bool) \rightarrow t \rightarrow t$

val $iteri$: $(int \rightarrow bool \rightarrow unit) \rightarrow t \rightarrow unit$
val $mapi$: $(int \rightarrow bool \rightarrow bool) \rightarrow t \rightarrow t$

val $fold\_left$: $(\alpha \rightarrow bool \rightarrow \alpha) \rightarrow \alpha \rightarrow t \rightarrow \alpha$
val $fold\_right$: $(bool \rightarrow \alpha \rightarrow \alpha) \rightarrow t \rightarrow \alpha \rightarrow \alpha$

82. Bitwise operations. *bwand*, *bwor* and *bwxor* implement logical and, or and exclusive or. They return fresh vectors and raise *Invalid_argument* `"Bitv.xxx"` if the two vectors do not have the same length (where `xxx` is the name of the function). *bwnot* implements the logical negation. It returns a fresh vector. *shiftl* and *shiftr* implement shifts. They return fresh vectors. *shiftl* moves bits from least to most significant, and *shiftr* from most to least significant (think lsl and lsr). *all_zeros* and *all_ones* respectively test for a vector only containing zeros and only containing ones.

val bw_and : $t \rightarrow t \rightarrow t$
val bw_or : $t \rightarrow t \rightarrow t$
val bw_xor : $t \rightarrow t \rightarrow t$
val bw_not : $t \rightarrow t$

val $shiftl$: $t \rightarrow int \rightarrow t$
val $shiftr$: $t \rightarrow int \rightarrow t$

val all_zeros : $t \rightarrow bool$
val all_ones : $t \rightarrow bool$

83. Conversions to and from strings. Least significant bit comes first.

val to_string : $t \rightarrow string$
val $from\_string$: $string \rightarrow t$

84. Only if you know what you are doing...

val $unsafe\_set$: $t \rightarrow int \rightarrow bool \rightarrow unit$
val $unsafe\_get$: $t \rightarrow int \rightarrow bool$

Module Bitv.ml

85. $Id : bitv.ml, v1.1 2002/04/03 01:13:04 ruess Exp$

86. Bit vectors. The interface and part of the code are borrowed from the *Array* module of the ocaml standard library (but things are simplified here since we can always initialize a bit vector). This module also provides bitwise operations.

87. We represent a bit vector by a vector of integers (field *bits*), and we keep the information of the size of the bit vector since it can not be found out with the size of the array (field *length*).

```
type t = {
  length : int;
  bits : int array }

let length v = v.length
```

88. Each element of the array is an integer containing *bpi* bits, where *bpi* is determined according to the machine word size. Since we do not use the sign bit, *bpi* is 30 on a 32-bits machine and 62 on a 64-bits machines. We maintain the following invariant: *The unused bits of the last integer are always zeros.* This is ensured by *create* and maintained in other functions using *normalize*. *bit_j*, *bit_not_j*, *low_mask* and *up_mask* are arrays used to extract and mask bits in a single integer.

```
let bpi = Sys.word_size - 2

let max_length = Sys.max_array_length × bpi

let bit_j = Array.init bpi (fun j → 1 lsl j)
let bit_not_j = Array.init bpi (fun j → max_int - bit_j.(j))

let low_mask = Array.create (succ bpi) 0
let _ =
  for i = 1 to bpi do low_mask.(i) ← low_mask.(i - 1) lor bit_j.(pred i) done

let keep_lowest_bits a j = a land low_mask.(j)

let high_mask = Array.init (succ bpi) (fun j → low_mask.(j) lsl (bpi - j))

let keep_highest_bits a j = a land high_mask.(j)
```

89. Creating and normalizing a bit vector is easy: it is just a matter of taking care of the invariant. Copy is immediate.

```
let create n b =
  let initv = if b then max_int else 0 in
  let r = n mod bpi in
  if r = 0 then
    { length = n; bits = Array.create (n / bpi) initv }
  else begin
    let s = n / bpi in
    let b = Array.create (succ s) initv in
    b.(s) ← b.(s) land low_mask.(r);
    { length = n; bits = b }
  end

let normalize v =
  let r = v.length mod bpi in
  if r > 0 then
    let b = v.bits in
    let s = Array.length b in
    b.(s - 1) ← b.(s - 1) land low_mask.(r)
```

```
let copy v = { length = v.length; bits = Array.copy v.bits }
```

90. Access and assignment. The nth bit of a bit vector is the jth bit of the ith integer, where $i = n / bpi$ and $j = n \bmod bpi$. Both i and j and computed by the function *pos*. Accessing a bit is testing whether the result of the corresponding mask operation is non-zero, and assigning it is done with a bitwiwe operation: an *or* with *bit_j* to set it, and an *and* with *bit_not_j* to unset it.

```
let pos n =
  let i = n / bpi and j = n mod bpi in
  if j < 0 then (i - 1, j + bpi) else (i, j)

let unsafe_get v n =
  let (i, j) = pos n in
  ((Array.unsafe_get v.bits i) land (Array.unsafe_get bit_j j)) > 0

let unsafe_set v n b =
  let (i, j) = pos n in
  if b then
    Array.unsafe_set v.bits i
      ((Array.unsafe_get v.bits i) lor (Array.unsafe_get bit_j j))
  else
    Array.unsafe_set v.bits i
      ((Array.unsafe_get v.bits i) land (Array.unsafe_get bit_not_j j))
```

91. The corresponding safe operations test the validiy of the access.

```
let get v n =
  if n < 0 ∨ n ≥ v.length then invalid_arg "Bitv.set";
  let (i, j) = pos n in
  ((Array.unsafe_get v.bits i) land (Array.unsafe_get bit_j j)) > 0

let set v n b =
  if n < 0 ∨ n ≥ v.length then invalid_arg "Bitv.set";
  let (i, j) = pos n in
  if b then
    Array.unsafe_set v.bits i
      ((Array.unsafe_get v.bits i) lor (Array.unsafe_get bit_j j))
  else
    Array.unsafe_set v.bits i
      ((Array.unsafe_get v.bits i) land (Array.unsafe_get bit_not_j j))
```

92. *init* is implemented naively using *unsafe_set*.

```
let init n f =
  let v = create n false in
  for i = 0 to pred n do
    unsafe_set v i (f i)
  done;
```

v

93. Handling bits by packets is the key for efficiency of functions *append*, *concat*, *sub* and *blit*. We start by a very general function *blit_bits* $a\ i\ m\ v\ n$ which blits the bits i to $i+m-1$ of a native integer a onto the bit vector v at index n. It assumes that $i..i+m-1$ and $n..n+m-1$ are respectively valid subparts of a and v. It is optimized when the bits fit the lowest boundary of an integer (case $j \equiv 0$).

```
let blit_bits a i m v n =
  let (i', j) = pos n in
  if j ≡ 0 then
    Array.unsafe_set v i'
      ((keep_lowest_bits (a lsr i) m) lor
       (keep_highest_bits (Array.unsafe_get v i') (bpi − m)))
  else
    let d = m + j − bpi in
    if d > 0 then begin
      Array.unsafe_set v i'
        (((keep_lowest_bits (a lsr i) (bpi − j)) lsl j) lor
         (keep_lowest_bits (Array.unsafe_get v i') j));
      Array.unsafe_set v (succ i')
        ((keep_lowest_bits (a lsr (i + bpi − j)) d) lor
         (keep_highest_bits (Array.unsafe_get v (succ i')) (bpi − d)))
    end else
      Array.unsafe_set v i'
        (((keep_lowest_bits (a lsr i) m) lsl j) lor
         ((Array.unsafe_get v i') land (low_mask.(j) lor high_mask.(−d))))
```

94. *blit_int* implements *blit_bits* in the particular case when $i = 0$ and $m = bpi$ i.e. when we blit all the bits of a.

```
let blit_int a v n =
  let (i, j) = pos n in
  if j ≡ 0 then
    Array.unsafe_set v i a
  else begin
    Array.unsafe_set v i
      ( (keep_lowest_bits (Array.unsafe_get v i) j) lor
       ((keep_lowest_bits a (bpi − j)) lsl j));
    Array.unsafe_set v (succ i)
      ((keep_highest_bits (Array.unsafe_get v (succ i)) (bpi − j)) lor
       (a lsr (bpi − j)))
  end
```

95. When blitting a subpart of a bit vector into another bit vector, there are two possible cases: (1) all the bits are contained in a single integer of the first bit vector, and a single call to *blit_bits* is the only thing to do, or (2) the source bits overlap on several integers of the source array, and then we do a loop of *blit_int*, with two calls to *blit_bits* for the two bounds.

```
let unsafe_blit v1 ofs1 v2 ofs2 len =
  let (bi, bj) = pos ofs1 in
  let (ei, ej) = pos (ofs1 + len − 1) in
  if bi ≡ ei then
    blit_bits (Array.unsafe_get v1 bi) bj len v2 ofs2
  else begin
    blit_bits (Array.unsafe_get v1 bi) bj (bpi − bj) v2 ofs2;
    let n = ref (ofs2 + bpi − bj) in
    for i = succ bi to pred ei do
      blit_int (Array.unsafe_get v1 i) v2 !n;
      n := !n + bpi
    done;
    blit_bits (Array.unsafe_get v1 ei) 0 (succ ej) v2 !n
  end

let blit v1 ofs1 v2 ofs2 len =
  if len < 0 ∨ ofs1 < 0 ∨ ofs1 + len > v1.length
            ∨ ofs2 < 0 ∨ ofs2 + len > v2.length
  then invalid_arg "Bitv.blit";
  unsafe_blit v1.bits ofs1 v2.bits ofs2 len
```

96. Extracting the subvector $ofs..ofs + len - 1$ of v is just creating a new vector of length *len* and blitting the subvector of v inside.

```
let sub v ofs len =
  if ofs < 0 ∨ len < 0 ∨ ofs + len > v.length then invalid_arg "Bitv.sub";
  let r = create len false in
  unsafe_blit v.bits ofs r.bits 0 len;
  r
```

97. The concatenation of two bit vectors *v1* and *v2* is obtained by creating a vector for the result and blitting inside the two vectors. *v1* is copied directly.

```
let append v1 v2 =
  let l1 = v1.length
  and l2 = v2.length in
  let r = create (l1 + l2) false in
  let b1 = v1.bits in
  let b2 = v2.bits in
  let b = r.bits in
  for i = 0 to Array.length b1 − 1 do
    Array.unsafe_set b i (Array.unsafe_get b1 i)
  done;
```

```
  unsafe_blit b2 0 b l1 l2;
  r
```

98. The concatenation of a list of bit vectors is obtained by iterating *unsafe_blit*.

```
let concat vl =
  let size = List.fold_left (fun sz v → sz + v.length) 0 vl in
  let res = create size false in
  let b = res.bits in
  let pos = ref 0 in
  List.iter
    (fun v →
       let n = v.length in
       unsafe_blit v.bits 0 b !pos n;
       pos := !pos + n)
    vl;
  res
```

99. Filling is a particular case of blitting with a source made of all ones or all zeros. Thus we instanciate *unsafe_blit*, with 0 and *max_int*.

```
let blit_zeros v ofs len =
  let (bi, bj) = pos ofs in
  let (ei, ej) = pos (ofs + len − 1) in
  if bi ≡ ei then
    blit_bits 0 bj len v ofs
  else begin
    blit_bits 0 bj (bpi − bj) v ofs;
    let n = ref (ofs + bpi − bj) in
    for i = succ bi to pred ei do
      blit_int 0 v !n;
      n := !n + bpi
    done;
    blit_bits 0 0 (succ ej) v !n
  end

let blit_ones v ofs len =
  let (bi, bj) = pos ofs in
  let (ei, ej) = pos (ofs + len − 1) in
  if bi ≡ ei then
    blit_bits max_int bj len v ofs
  else begin
    blit_bits max_int bj (bpi − bj) v ofs;
    let n = ref (ofs + bpi − bj) in
    for i = succ bi to pred ei do
      blit_int max_int v !n;
      n := !n + bpi
```

```
    done;
    blit_bits max_int 0 (succ ej) v !n
  end

let fill v ofs len b =
  if ofs < 0 ∨ len < 0 ∨ ofs + len > v.length then invalid_arg "Bitv.fill";
  if b then blit_ones v.bits ofs len else blit_zeros v.bits ofs len
```

100. All the iterators are implemented as for traditional arrays, using *unsafe_get*. For *iter* and *map*, we do not precompute (f true) and (f false) since f is likely to have side-effects.

```
let iter f v =
  for i = 0 to v.length − 1 do f (unsafe_get v i) done

let map f v =
  let l = v.length in
  let r = create l false in
  for i = 0 to l − 1 do
    unsafe_set r i (f (unsafe_get v i))
  done;
  r

let iteri f v =
  for i = 0 to v.length − 1 do f i (unsafe_get v i) done

let mapi f v =
  let l = v.length in
  let r = create l false in
  for i = 0 to l − 1 do
    unsafe_set r i (f i (unsafe_get v i))
  done;
  r

let fold_left f x v =
  let r = ref x in
  for i = 0 to v.length − 1 do
    r := f !r (unsafe_get v i)
  done;
  !r

let fold_right f v x =
  let r = ref x in
  for i = v.length − 1 downto 0 do
    r := f (unsafe_get v i) !r
  done;
  !r
```

101. Bitwise operations. It is straigthforward, since bitwise operations can be realized by the corresponding bitwise operations over integers. However, one has to take care of normalizing the result of *bwnot* which introduces ones in highest significant positions.

```
let bw_and v1 v2 =
  let l = v1.length in
  if l ≠ v2.length then invalid_arg "Bitv.bw_and";
  let b1 = v1.bits
  and b2 = v2.bits in
  let n = Array.length b1 in
  let a = Array.create n 0 in
  for i = 0 to n - 1 do
    a.(i) ← b1.(i) land b2.(i)
  done;
  { length = l; bits = a }

let bw_or v1 v2 =
  let l = v1.length in
  if l ≠ v2.length then invalid_arg "Bitv.bw_or";
  let b1 = v1.bits
  and b2 = v2.bits in
  let n = Array.length b1 in
  let a = Array.create n 0 in
  for i = 0 to n - 1 do
    a.(i) ← b1.(i) lor b2.(i)
  done;
  { length = l; bits = a }

let bw_xor v1 v2 =
  let l = v1.length in
  if l ≠ v2.length then invalid_arg "Bitv.bw_xor";
  let b1 = v1.bits
  and b2 = v2.bits in
  let n = Array.length b1 in
  let a = Array.create n 0 in
  for i = 0 to n - 1 do
    a.(i) ← b1.(i) lxor b2.(i)
  done;
  { length = l; bits = a }

let bw_not v =
  let b = v.bits in
  let n = Array.length b in
  let a = Array.create n 0 in
  for i = 0 to n - 1 do
    a.(i) ← max_int land (lnot b.(i))
  done;
  let r = { length = v.length; bits = a } in
  normalize r;
```

```
  r
```

102. Shift operations. It is easy to reuse *unsafe_blit*, although it is probably slightly less efficient than a ad-hoc piece of code.

```
let rec shiftl v d =
  if d ≡ 0 then
    copy v
  else if d < 0 then
    shiftr v (−d)
  else begin
    let n = v.length in
    let r = create n false in
    if d < n then unsafe_blit v.bits 0 r.bits d (n − d);
    r
  end

and shiftr v d =
  if d ≡ 0 then
    copy v
  else if d < 0 then
    shiftl v (−d)
  else begin
    let n = v.length in
    let r = create n false in
    if d < n then unsafe_blit v.bits d r.bits 0 (n − d);
    r
  end
```

103. Testing for all zeros and all ones.

```
let all_zeros v =
  let b = v.bits in
  let n = Array.length b in
  let rec test i =
    (i ≡ n) ∨ ((Array.unsafe_get b i ≡ 0) ∧ test (succ i))
  in
  test 0

let all_ones v =
  let b = v.bits in
  let n = Array.length b in
  let rec test i =
    if i ≡ n − 1 then
      let m = v.length mod bpi in
      (Array.unsafe_get b i) ≡ (if m ≡ 0 then max_int else low_mask.(m))
    else
```

```
    ((Array.unsafe_get b i) ≡ max_int) ∧ test (succ i)
  in
  test 0
```

104. Conversions to and from strings.

```
let to_string v =
  let n = v.length in
  let s = String.make n '0' in
  for i = 0 to n − 1 do
    if unsafe_get v i then s.[i] ← '1'
  done;
  s

let from_string s =
  let n = String.length s in
  let v = create n false in
  for i = 0 to n − 1 do
    let c = String.unsafe_get s i in
    if c = '1' then
      unsafe_set v i true
    else
      if c ≠ '0' then invalid_arg "Bitv.from_string"
  done;
  v
```

6.12 Module extq

Interface for module Extq.mli

105. Module *Extq*: Arithmetic operations on the rationals extended with positive and negative infinity.

```
type extq =
  | Inject of Mpa.Q.t
  | Posinf
  | Neginf

and t
```

106. Constructors.

```
val of_q : Mpa.Q.t → t
val posinf : t
val neginf : t
```

107. Destructor.

val $destruct\ :\ t\ \rightarrow\ extq$

108. Is the extended real a real, and translating to a rational if possible.

val $is\_q :\ t\ \rightarrow\ bool$
val $to\_q\ :\ t\ \rightarrow\ Mpa.Q.t\ option$

val $is\_z :\ t\ \rightarrow\ bool$
val $to\_z\ :\ t\ \rightarrow\ Mpa.Z.t\ option$

109. Test if argument is an integer.

val $is\_int\ :\ t\ \rightarrow\ bool$

110. Printing an extended rational.

val $pp\ :\ Format.formatter\ \rightarrow\ t\ \rightarrow\ unit$

111. Derived constructors.

val $zero\ :\ t$
val $is\_zero\ :\ t\ \rightarrow\ bool$

112. Equality on extended reals.

val $eq\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$

113. Extended reals are ordered as among the reals, if both are real. Moreover, -$inf\ <\ c\ <\ +inf$ for any real c.

val $lt\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$
val $le\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$

114. Comparison.

val $cmp\ :\ t\ \rightarrow\ t\ \rightarrow\ Mpa.Q.cmp$

115. Minimum and maximum.

val $min\ :\ t\ \rightarrow\ t\ \rightarrow\ t$
val $max\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

116. Sign computation.

val $sign\ :\ t\ \rightarrow\ Sign.t$

117. Arithmetic operations.

val $add\ :\ t\ \rightarrow\ t\ \rightarrow\ t$
val $sub\ :\ t\ \rightarrow\ t\ \rightarrow\ t$
val $mult\ :\ t\ \rightarrow\ t\ \rightarrow\ t$
val $div\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

Module Extq.ml

118. type *extq* =

```
  | Inject of Q.t
  | Posinf
  | Neginf
and t = extq
let destruct x = x
let inject q = Inject(q)
let posinf = Posinf
let neginf = Neginf
```

119. Miscellaneous.

```
let zero = inject Q.zero
let is_zero x =
  match x with
    | Inject q → Q.is_zero q
    | _ → false
let is_q x =
  match x with
    | Inject _ → true
    | _ → false
let to_q x =
  match x with
    | Inject q → Some q
    | _ → None
let is_z x =
  match x with
    | Inject(q) when Mpa.Q.is_integer q → true
    | _ → false
let to_z x =
  assert(is_z x);
  match x with
    | Inject(q) → Some(Mpa.Q.to_z q)
    | _ → None
let of_q = inject
let pp fmt x =
  match x with
```

```
    | Inject q → Q.pp fmt q
    | Posinf → Format.fprintf fmt "inf"
    | Neginf → Format.fprintf fmt "-inf"
```

120. Test if argument is an integer.

```
let is_int x =
  match x with
    | Inject q → Q.is_integer q
    | _ → false
```

121. Equality is just pointer comparison for hash-consing.

```
let eq x y =
  match x, y with
    | Inject(q), Inject(p) → Q.equal q p
    | Posinf, Posinf → true
    | Neginf, Neginf → true
    | _ → false
```

122. Ordering relation.

```
let lt x y =
  match x with
    | Inject q →
        (match y with
           | Inject p → Q.lt q p
           | Posinf → true
           | Neginf → false)
    | Neginf → true
    | Posinf → false

let le x y =
  eq x y ∨ lt x y
```

123. Minimum and maximum.

```
let min x y = if lt x y then x else y

let max x y = if lt x y then y else x
```

124. Comparisons.

```
let cmp x y =
  match x, y with
    | Inject u, Inject v → Q.cmp u v
    | Inject _, Posinf → Q.Less
    | Inject _, Neginf → Q.Greater
    | Neginf, Neginf → Q.Equal
    | Neginf, _ → Q.Less
```

```
    | Posinf, Posinf → Q.Equal
    | Posinf, _ → Q.Greater
```

125. Sign computation.

```
let sign x =
  match x with
    | Inject q → Q.sign q
    | Posinf → Sign.Pos
    | Neginf → Sign.Neg
```

126. Arithmetic operations

```
exception Undefined

let add x y =
  match x with
    | Inject p →
        (match y with
           | Inject q → inject (Q.add p q)
           | Posinf → posinf
           | Neginf → neginf)
    | Neginf →
        (match y with
           | Posinf → raise Undefined
           | _ → neginf)
    | Posinf →
        (match y with
           | Neginf → raise Undefined
           | _ → posinf)

let sub x y =
  match x with
    | Inject p →
        (match y with
           | Inject q → inject (Q.sub p q)
           | Posinf → neginf
           | Neginf → posinf)
    | Neginf →
        (match y with
           | Neginf → raise Undefined
           | _ → neginf)
    | Posinf →
        (match y with
           | Posinf → raise Undefined
           | _ → posinf)

let mult x y =
```

```
  match x with
    | Inject p ->
        (match y with
            | Inject q -> inject (Q.mult p q)
            | Neginf ->
                (match Q.sign p with
                    | Zero -> raise Undefined
                    | Neg -> posinf
                    | Pos -> neginf)
            | Posinf ->
                (match Q.sign p with
                    | Zero -> raise Undefined
                    | Neg -> neginf
                    | Pos -> posinf))
    | Posinf ->
        (match y with
            | Inject q ->
                (match Q.sign q with
                    | Neg -> neginf
                    | Zero -> raise Undefined
                    | Pos -> posinf)
            | Posinf -> posinf
            | Neginf -> neginf)
    | Neginf ->
        (match y with
            | Neginf ->
                posinf
            | Inject q ->
                (match Q.sign q with
                    | Neg -> posinf
                    | Zero -> raise Undefined
                    | Pos -> neginf)
            | Posinf ->
                neginf)

let div x y =
  match y with
    | Inject q ->
        (match Q.sign q with
            | Zero ->
                raise Undefined
            | Neg ->
                (match x with
                    | Inject p -> inject (Q.div p q)
                    | Neginf -> neginf
```

```
                | Posinf → posinf)
            | Pos →
                (match x with
                    | Inject p → inject (Q.div p q)
                    | Neginf → posinf
                    | Posinf → neginf))
    | Posinf →
        (match x with
            | Neginf | Posinf → raise Undefined
            | _ → zero)
    | Neginf →
        (match x with
            | Neginf | Posinf → raise Undefined
            | _ → zero)
```

6.13 Module dom

Interface for module Dom.mli

127. Module *Dom*: Various subdomains of numbers. *Real* denotes the domain of the real numbers, *Int* all integers. There is a reflexive subdomain ordering $<$ with $Int < Real$

type $t = Int \mid Nonint \mid Real$

128. Equality on domains.

val $eq : t \rightarrow t \rightarrow bool$

129. *union d1 d2* returns *d* iff the domain of *d* is the union of the domains of *d1* and *d2*.

val $union : t \rightarrow t \rightarrow t$

130. *inter d1 d2* returns *d* iff the domain of *d* is the intersection of the domains of *d1* and *d2*.

exception *Empty*

val $inter : t \rightarrow t \rightarrow t$

sub d1 d2 tests if the domain of *d1* is a subdomain of *d2*.

val $sub : t \rightarrow t \rightarrow bool$

131. *cmp d1 d2* returns *Sub* (*Equal*, *Disjoint*, *Super*) if the domain of *d1* is a subset (is equal, is disjoint, is a superset) of *d2*.

val $cmp : t \rightarrow t \rightarrow unit\ Binrel.t$

132. *of_q u* returns *Int* if the rational *u* is integer and *Real* val of_q : $Mpa.Q.t \rightarrow t$

133. $mem\ q\ d$ tests if q is a member of d.

val mem : $Mpa.Q.t \rightarrow t \rightarrow bool$

134. Pretty-printing

val pp : $Format.formatter \rightarrow t \rightarrow unit$

Module Dom.ml

135. type $t = Int \mid Nonint \mid Real$

let $eq\ d1\ d2 = (d1 = d2)$

136. Union of two domains

```
let union d1 d2 =
  match d1, d2 with
    | Int, Int → Int
    | Nonint, Nonint → Nonint
    | _ → Real
```

137. Intersection of two domains

exception $Empty$

```
let inter d1 d2 =
  match d1, d2 with
    | Real, Real → Real
    | Int, Nonint → raise Empty
    | Nonint, Int → raise Empty
    | Int, _ → Int
    | _, Int → Int
    | Nonint, _ → Nonint
    | _, Nonint → Nonint
```

138. Testing for disjointness.

```
let disjoint d1 d2 =
  match d1, d2 with
    | Int, Nonint → true
    | Nonint, Int → true
    | _ → false
```

139. Testing for subdomains.

let $sub\ d1\ d2$ =

```
  match d1, d2 with
    | _, Real → true
    | Int, Int → true
    | Nonint, Nonint → true
    | _ → false

let cmp d1 d2 =
  match d1, d2 with
    | Int, Int → Binrel.Same
    | Int, Real → Binrel.Sub
    | Int, Nonint → Binrel.Disjoint
    | Real, Int → Binrel.Super
    | Real, Real → Binrel.Same
    | Real, Nonint → Binrel.Super
    | Nonint, Nonint → Binrel.Same
    | Nonint, Real → Binrel.Sub
    | Nonint, Int → Binrel.Disjoint

let of_q q =
  if Mpa.Q.is_integer q then Int else Nonint

let mem q = function
  | Real → true
  | Nonint → ¬ (Mpa.Q.is_integer q)
  | Int → Mpa.Q.is_integer q

let pp fmt d =
  let s = match d with
    | Real → "real"
    | Int → "int"
    | Nonint → "nonint"
  in
  Format.fprintf fmt "%s" s
```

6.14 Module endpoint

Interface for module Endpoint.mli

140. Module *Endpoint*: datatype for endpoints of intervals.

```
type t = Extq.t × bool
```

141. Constructor.

```
val make : Extq.t × bool → t
```

142. Accessors.

val $destruct\ :\ t\ \rightarrow\ Extq.t\ \times\ bool$

val $value\ :\ t\ \rightarrow\ Extq.t$
val $kind\ :\ t\ \rightarrow\ bool$

143. Test if endpoint is a rational or integer.

val $is\_q\ :\ t\ \rightarrow\ bool$

val $is\_z\ :\ t\ \rightarrow\ bool$

144. Get value of a rational/integer endpoint.

val $q\_of\ :\ t\ \rightarrow\ Mpa.Q.t$

val $z\_of\ :\ t\ \rightarrow\ Mpa.Z.t$

145. Strictness/Nonstrictness test.

val $is\_strict\ :\ t\ \rightarrow\ bool$
val $is\_nonstrict\ :\ t\ \rightarrow\ bool$

146. Extreme endpoints

val $neginf\ :\ t$
val $posinf\ :\ t$

val $strict\ :\ Mpa.Q.t\ \rightarrow\ t$
val $nonstrict\ :\ Mpa.Q.t\ \rightarrow\ t$

val $eq\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$

Module Endpoint.ml

147. type $t\ =\ Extq.t\ \times\ bool$

148. Constructor.

let $make\ e\ =\ e$

149. Accessors.

let $destruct\ e\ =\ e$

let $value\ e\ =\ fst\ e$
let $kind\ e\ =\ snd\ e$

150. Extreme endpoints

let $neginf\ =\ (Extq.neginf,$ false$)$
let $posinf\ =\ (Extq.posinf,$ false$)$

```
let strict u = (Extq.of_q u, false)
let nonstrict u = (Extq.of_q u, true)
```

151. Equality.

```
let eq (a, alpha) (b, beta) =
  match Extq.destruct a, Extq.destruct b with
    | Extq.Inject u, Extq.Inject v → alpha = beta ∧ Mpa.Q.equal u v
    | Extq.Posinf, Extq.Posinf → true
    | Extq.Neginf, Extq.Neginf → true
    | _ → false
```

152. Test if endpoint is a rational.

```
let is_q (a, _) = Extq.is_q a

let is_z (a, _) = Extq.is_z a
```

153. Get value of a rational endpoint.

```
let q_of (a, _) =
  assert(Extq.is_q a);
  match Extq.to_q a with
    | Some(q) → q
    | _ → assert false

let z_of (a, _) =
  assert(Extq.is_z a);
  match Extq.to_z a with
    | Some(q) → q
    | _ → assert false
```

154. Strictness/Nonstrictness test.

```
let is_strict (_, alpha) = ¬ alpha

let is_nonstrict (_, alpha) = alpha
```

6.15 Module interval

Interface for module Interval.mli

155. Module *Interval*: Real intervals with rational endpoints (including infinity).

```
type t
```

make d $(a, alpha)$ $(b, beta)$ constructs a general, interval with rational endpoints a and b (including negative and positive infinity) and two bits of information *alpha* and *beta*, with *alpha* (resp. *beta*) specifying whether a (resp. b) belongs to the interval. Each interval denotates a set of reals (or integers), denoted by $D(i)$. $D(make\ d\ (a,\mathsf{true})\ (b,\mathsf{true}))$ equals the closed interval $\{x \text{ in } d \mid a \ \leq \ x \ \leq \ b\}$, $D(make\ d\ (a,\mathsf{true})\ (b,\mathsf{false}))$ is the right-open interval $\{x \text{ in } d \mid a \ \leq \ x \ < \ b\}$, $D(make\ d\ (a,\mathsf{false})\ (b,\ \mathsf{true}))$ denotes the left-open interval $\{x \text{ in } d \mid a \ < \ x \ \leq \ b\}$, and $D(make\ d\ (a,\mathsf{false})\ (b,\mathsf{false}))$ denotes the open interval $\{x \text{ in } d \mid a \ < \ x \ < \ b\}$.

val $make\ :\ Dom.t\ \times\ Endpoint.t\ \times\ Endpoint.t\ \rightarrow\ t$

156. The accessor $destructure\ i$ returns the endpoint information (d, lo, hi) for an interval i with denotation $D(make\ d\ lo\ hi)$. $lo\ i$ returns the lower bound in the form $(a, alpha)$ and $hi\ i$ the upper bound in the form $(b,\ beta)$.

val $destructure\ :\ t\ \rightarrow\ Dom.t\ \times\ Endpoint.t\ \times\ Endpoint.t$

val $dom\ :\ t\ \rightarrow\ Dom.t$
val $lo\ :\ t\ \rightarrow\ Endpoint.t$
val $hi\ :\ t\ \rightarrow\ Endpoint.t$

157. Derived constructors.

val $mk\_zero\ :\ t$
val $mk\_empty\ :\ t$
val $mk\_real\ :\ t$
val $mk\_int\ :\ t$
val $mk\_nonint\ :\ t$
val $mk\_singleton\ :\ Mpa.Q.t\ \rightarrow\ t$

158. $is\_singleton\ i$ returns the single value for an interval whose denotation is a singleton set over the reals.

val $d\_singleton\ :\ t\ \rightarrow\ Mpa.Q.t\ option$
val $is\_empty\ :\ t\ \rightarrow\ bool$
val $is\_full\ :\ t\ \rightarrow\ bool$

159. $rational\_endpoints\ i$ returns the pair of rational endpoints.

val $rational\_endpoints\ :\ t\ \rightarrow\ (Mpa.Q.t\ \times\ Mpa.Q.t)\ option$

160. $mem\ q\ i$ tests if the rational q is a member of the interval i.

val $mem :\ Mpa.Q.t\ \rightarrow\ t\ \rightarrow\ bool$

161. Equality $eq\ i\ j$ on intervals i, j holds if and only if they denote the same set of numbers. In particular, two empty intervals are always identified to be equal.

val $eq :\ t\ \rightarrow\ t\ \rightarrow\ bool$

162. Union and intersection of two intervals. It is the case that the denotation of $union\ i\ j$ (resp. $inter\ i\ j$) is the union (resp. intersection) of the denotations of i and j.

val $union\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

val $inter\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

163. Interval arithmetic. Let i, j be two intervals with denotations $D(i)$ and $D(j)$. Then, $D(add\ i\ j)$ is $D(i)\ +\ D(j)$, $D(sub\ i\ j)$ is $D(i)\ -\ D(j)$, $D(mult\ i\ j)$ is $D(i)\ \times\ D(j)$, and $D(div\ i\ j)$ consists of the set of rationals $\{z|\exists x \in D(i), y \in D(j)$ such that $y/=0$, $z = x/y\}$.

val $add\ :\ t\ \rightarrow\ t\ \rightarrow\ t$
val $subtract\ :\ t\ \rightarrow\ t\ \rightarrow\ t$
val $multq\ :\ Mpa.Q.t\ \rightarrow\ t\ \rightarrow\ t$
val $mult\ :\ t\ \rightarrow\ t\ \rightarrow\ t$
val $expt\ :\ int \rightarrow\ t\ \rightarrow\ t$
val $div\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

164. Comparisons.

val $cmp\ :\ t\ \rightarrow\ t\ \rightarrow\ t\ Binrel.t$

val $sub\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$

val $disjoint\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$

165. Printing an interval.

val $pp\ :\ Format.formatter\ \rightarrow\ t\ \rightarrow\ unit$

Module Interval.ml

166. General real intervals are represented by their endpoints a and b and two bits of information $alpha$ and $beta$, with $alpha$ (resp. $beta$) specifying whether a (resp. b) belongs to the interval. More formally. (a, b,true,true) denotes the closed interval $\{x$ in $R \mid u\ \leq\ x\ \leq\ v\}$, ($a$, b,true,false) denotes the right-open interval $\{x$ in $R \mid u\ \leq\ x\ <\ v\}$, ($a$, b,false,true) denotes the left-open interval $\{x$ in $R \mid u\ <\ x\ \leq\ v\}$, and ($a$, b,false,false) denotes the open interval $\{x$ in $R \mid u\ <\ x\ <\ v\}$.

type $t\ =\ tnode$

and $tnode\ =\ \{$
  $dom\ :\ Dom.t;$
  $lo\ :\ Endpoint.t;$
  $hi\ :\ Endpoint.t$
$\}$

167. Accessors.

let $dom\ i\ =\ i.dom$
let $lo\ i\ =\ i.lo$
let $hi\ i\ =\ i.hi$

```
let destructure i = (i.dom, i.lo, i.hi)
```

168. Equality.

```
let eq i j =
  Dom.eq i.dom j.dom ∧
  Endpoint.eq i.lo j.lo ∧
  Endpoint.eq i.hi j.hi
```

169. Constructing intervals.

```
let mk_empty =
  let z = Endpoint.strict Mpa.Q.zero in
  {dom = Dom.Real; lo = z; hi = z}

let rec make (d, lo, hi) =
  match d with
    | Dom.Int → makeint lo hi
    | Dom.Real → makereal lo hi
    | Dom.Nonint → makenonint lo hi

and makenonint lo hi =
  let (a, l) = Endpoint.destruct lo
  and (b, k) = Endpoint.destruct hi
  in
  let ((a′, _) as lo′) =
    match Extq.destruct a with
      | Extq.Inject u when l ∧ Q.is_integer u → (a, false)
      | _ → lo
  and ((b′, _) as hi′) =
    match Extq.destruct b with
      | Extq.Inject u when k ∧ Q.is_integer u → (b, false)
      | _ → hi
  in
  if Extq.lt b′ a′ then
    mk_empty
  else
    {dom = Dom.Nonint; lo = lo′; hi = hi′}

and makereal lo hi =
  let (a, l) = Endpoint.destruct lo
  and (b, k) = Endpoint.destruct hi
  in
  let is_empty =
    match Extq.destruct a, Extq.destruct b with
      | Extq.Inject u, Extq.Inject v →
          (match Q.cmp u v with
             | Q.Greater → true
```

```
              | Q.Equal → ¬(l ∧ k)
              | Q.Less → false)
       | Extq.Neginf, Extq.Neginf → true
       | Extq.Posinf, _ → true
       | _ → false
   in
   if is_empty then
     mk_empty
   else
     {dom = Dom.Real; lo = lo; hi = hi}

and makeint lo hi =
   let (a, l) = Endpoint.destruct lo
   and (b, k) = Endpoint.destruct hi
   in
   let ((a′, _) as lo′) =
     match Extq.destruct a with
       | Extq.Inject u when ¬(l ∧ Q.is_integer u) →
           (Extq.of_q(Q.of_z(Q.floor(Q.add u Q.one))), true)
       | _ →
           lo
   and ((b′, _) as hi′) =
     match Extq.destruct b with
       | Extq.Inject u when ¬(k ∧ Q.is_integer u) →
           (Extq.of_q(Q.of_z(Q.ceil(Q.sub u Q.one))), true)
       | _ →
           hi
   in
   if Extq.lt b′ a′ then
     mk_empty
   else
     {dom = Dom.Int; lo = lo′; hi = hi′}
```

170. Derived constructors.

```
let mk_real = make (Dom.Real, Endpoint.neginf, Endpoint.posinf)
let mk_int = make (Dom.Int, Endpoint.neginf, Endpoint.posinf)
let mk_nonint = make (Dom.Nonint, Endpoint.neginf, Endpoint.posinf)

let mk_singleton q =
   let bnd = Endpoint.nonstrict q in
   make (Dom.of_q q, bnd, bnd)

let mk_zero = mk_singleton Q.zero
```

Tests for special intervals.

```
let is_empty i = eq i mk_empty
```

```
let is_full i = eq i mk_real

let d_singleton i =
  let (a, l) = Endpoint.destruct i.lo
  and (b, k) = Endpoint.destruct i.hi
  in
  if l ∧ k ∧ Extq.eq a b then
    Extq.to_q a
  else
    None

let is_zero i = (eq i mk_zero)
```

171. Return rational endpoints.

```
let rational_endpoints i =
  let (a, _) = Endpoint.destruct i.lo
  and (b, _) = Endpoint.destruct i.hi
  in
  match Extq.to_q a, Extq.to_q b with
    | Some(q), Some(p) → Some(q, p)
    | _ → None
```

172. Test if q is in the interval i.

```
let mem q i =
  Dom.mem q i.dom ∧
  let (a, alpha) = Endpoint.destruct i.lo in
  let (b, beta) = Endpoint.destruct i.hi in
  match Extq.destruct a with
    | Extq.Inject u →
        (match Extq.destruct b with
            | Extq.Inject v →
                (match alpha, beta with
                    | true, true → Q.le u q ∧ Q.le q v
                    | true, false → Q.le u q ∧ Q.lt q v
                    | false, true → Q.lt u q ∧ Q.le q v
                    | false, false → Q.lt u q ∧ Q.lt q v)
            | Extq.Posinf →
                (match alpha with
                    | true → Q.le u q
                    | false → Q.lt u q)
            | Extq.Neginf →
                false)
    | Extq.Neginf →
        (match Extq.destruct b with
            | Extq.Inject v →
```

```
                (match beta with
                    | true →  Q.le q v
                    | false →  Q.lt q v)
            | Extq.Posinf  →
                true
            | Extq.Neginf  →
                false)
    | Extq.Posinf  →
        false
```

173. Printing an interval.

```
let pp fmt i  =
  let (a, alpha)  =  Endpoint.destruct i.lo in
  let (b, beta)  =  Endpoint.destruct i.hi in
    if Extq.destruct a  =  Extq.Neginf  ∧
        Extq.destruct b  =  Extq.Posinf  ∧
        Endpoint.is_strict i.lo  ∧
        Endpoint.is_strict i.hi
    then
      Dom.pp fmt i.dom
    else
      begin
        Format.fprintf fmt "";
        Dom.pp fmt i.dom;
        Format.fprintf fmt "%s" (if alpha  ∧  Extq.is_q a then "[" else "(");
        Extq.pp fmt a;
        Format.fprintf fmt "..";
        Extq.pp fmt b;
        Format.fprintf fmt "%s" (if beta  ∧  Extq.is_q b then "]" else ")")
      end
```

Union and intersection of intervals.

```
let mu a c alpha gamma  =
  match Extq.cmp a c with
    | Q.Less  →  alpha
    | Q.Equal  →  alpha  ∧  gamma
    | Q.Greater  →  gamma

let nu a c alpha gamma  =
  match Extq.cmp a c with
    | Q.Less  →  alpha
    | Q.Equal  →  alpha  ∨  gamma
    | Q.Greater  →  gamma

let inter i j  =
  let (a, alpha)  =  Endpoint.destruct i.lo
```

```
  and (b, beta) = Endpoint.destruct i.hi
  and (c, gamma) = Endpoint.destruct j.lo
  and (d, delta) = Endpoint.destruct j.hi
  in
  try
    let d' = Dom.inter i.dom j.dom in
    let lo' = Endpoint.make (Extq.max a c, mu a c gamma alpha) in
    let hi' = Endpoint.make (Extq.min b d, mu b d beta delta) in
    make (d', lo', hi')
  with
      Dom.Empty → mk_empty

let union i j =
  let (a, alpha) = Endpoint.destruct i.lo
  and (b, beta) = Endpoint.destruct i.hi
  and (c, gamma) = Endpoint.destruct j.lo
  and (d, delta) = Endpoint.destruct j.hi
  in
  let d' = Dom.union i.dom j.dom
  and lo' = Endpoint.make (Extq.min a c, mu a c alpha gamma)
  and hi' = Endpoint.make (Extq.max b d, mu b d delta beta)
  in
  make (d', lo', hi')
```

174. The implementation of interval arithmetic operations follows the tables given in "Interval Arithmetic: from Principles to Implementation" by Hickey, Ju, and Emden in the Journal of ACM, 2002.

175. Addition and Subtraction.

```
let rec add i j =
  let (a, alpha) = Endpoint.destruct i.lo
  and (b, beta) = Endpoint.destruct i.hi
  and (c, gamma) = Endpoint.destruct j.lo
  and (d, delta) = Endpoint.destruct j.hi
  in
  let d' = add_dom i.dom j.dom
  and lo' = Endpoint.make (Extq.add a c, alpha ∧ gamma)
  and hi' = Endpoint.make (Extq.add b d, beta ∧ delta)
  in
  make (d', lo', hi')

and add_dom d1 d2 =
  match d1, d2 with
    | Dom.Real, _ → Dom.Real
    | _, Dom.Real → Dom.Real
    | Dom.Int, Dom.Int → Dom.Int
```

```
  | Dom.Int, Dom.Nonint → Dom.Nonint
  | Dom.Nonint, Dom.Nonint → Dom.Nonint
  | Dom.Nonint, Dom.Int → Dom.Nonint
```

176. Classification of intervals according to the signs of endpoints.

```
type classification =
 | M (*s (a, b) in M iff a < 0 < b. *)
 | Z (*s (0, 0) is only interval in Z. *)
 | P0 (*s (0, b) in P0 iff b > 0. *)
 | P1 (*s (a, b) in P1 iff 0 < a ≤ b. *)
 | N0 (*s (a, 0) in N0 iff a < 0. *)
 | N1 (*s (a, b) in N1 iff a ≤ b < 0. *)
let classify i =
 let (a, alpha) = Endpoint.destruct i.lo
 and (b, beta) = Endpoint.destruct i.hi
 in
 match Extq.sign a with
  | Zero →
     (match Extq.sign b with
       | Zero → Z
       | Pos → P0
       | Neg → assert false)
  | Pos →
     P1
  | Neg →
     (match Extq.sign b with
       | Zero → N0
       | Neg → N1
       | Pos → M)
```

177. Multiplication of general real intervals.

```
let mult i j =
 let dom = Dom.union i.dom j.dom
 and (a, alpha) = Endpoint.destruct i.lo
 and (b, beta) = Endpoint.destruct i.hi
 and (c, gamma) = Endpoint.destruct j.lo
 and (d, delta) = Endpoint.destruct j.hi
 in
 let kind a c l k =
  (l ∧ k) ∨
  (l ∧ (Extq.is_zero a)) ∨
  (k ∧ (Extq.is_zero c))
 in
 let ( × ) = Extq.mult in
```

```
    let make lo hi = make (dom, lo, hi) in
    match classify i, classify j with
      | (P0 | P1), (P0 | P1) →
          make (a × c, kind a c alpha gamma) (b × d, kind b d beta delta)
      | (P0 | P1), M →
          make (b × c, kind b c beta gamma) (b × d, kind b d beta delta)
      | (P0 | P1), (N0 | N1) →
          make (b × c, kind b c beta gamma) (a × d, kind a d alpha delta)
      | M, (P0 | P1) →
          make (a × d, kind a d alpha delta) (b × d, kind b d beta delta)
      | M, M →
          union
            (make (a × d, kind a d alpha delta) (b × d, kind b d beta delta))
            (make (b × c, kind b c beta gamma) (a × c, kind a c alpha gamma))
      | M, (N0 | N1) →
          make (b × c, kind b c beta gamma) (a × c, kind a c alpha gamma)
      | (N0 | N1), (P0 | P1) →
          make (a × d, kind a d alpha delta) (b × c, kind b c beta gamma)
      | (N0 | N1), M →
          make (a × d, kind a d alpha delta) (a × c, kind a c alpha gamma)
      | (N0 | N1), (N0 | N1) →
          make (b × d, kind b d beta delta) (a × c, kind a c alpha gamma)
      | Z, (P0 | P1 | M | N0 | N1) →
          mk_zero
      | (P0 | P1 | M | N0 | N1 | Z), Z →
          mk_zero

let multq q i =
    if Endpoint.eq i.lo Endpoint.neginf ∧ Endpoint.eq i.hi Endpoint.posinf then
      i
    else
      mult (mk_singleton q) i

let subtract i j =
    add i (multq (Mpa.Q.minus Q.one) j)

let div i j =
    let dom = Dom.Real
    and (a, alpha) = Endpoint.destruct i.lo
    and (b, beta) = Endpoint.destruct i.hi
    and (c, gamma) = Endpoint.destruct j.lo
    and (d, delta) = Endpoint.destruct j.hi
    in
    let make lo hi = make (Dom.Real, lo, hi) in
    let kind a c alpha gamma =
      (alpha ∧ gamma) ∨ (alpha ∧ (Extq.is_zero a)) ∨ (gamma ∧ (Extq.is_zero c))
```

```
  in
  let ( / ) = Extq.div in
  match classify i, classify j with
    | (P0 | P1), P1 ->
        make (a / d, kind a d alpha delta) (b / c, kind b c beta gamma)
    | (P0 | P1), P0 ->
        make (a / d, kind a d alpha delta) posinf
    | (P0 | P1), M ->
        union
          (make neginf (a / c, kind a c alpha gamma))
          (make (a / d, kind a d alpha delta) posinf)
    | (P0 | P1), N0 ->
        make neginf (a / c, kind a c alpha gamma)
    | (P0 | P1), N1 ->
        make (b / d, kind b d beta delta) (a / c, kind a c alpha gamma)
    | M, P1 ->
        make (a / c, kind a c alpha gamma) (b / c, kind b c beta gamma)
    | M, (P0 | M | N0) ->
        make neginf posinf
    | M, N1 ->
        make (b / d, kind b d beta delta) (a / d, kind a d alpha delta)
    | (N0 | N1), P1 ->
        make (a / c, kind a c alpha gamma) (b / d, kind b d beta delta)
    | (N0 | N1), P0 ->
        make neginf (b / d, kind b d beta delta)
    | (N0 | N1), M ->
        union
          (make neginf (b / d, kind b d beta delta))
          (make (b / c, kind b c beta gamma) posinf)
    | (N0 | N1), N0 ->
        make (b / c, kind b c beta gamma) posinf
    | (N0 | N1), N1 ->
        make (b / c, kind b c beta gamma) (a / d, kind a d alpha delta)
    | Z, (P0 | P1 | M | N0 | N1) ->
        mk_zero
    | _, Z ->
        mk_empty

let nonneg = make (Dom.Real, Endpoint.nonstrict Q.zero, Endpoint.posinf)

let expt n l =
  let rec loop = function
    | 0 -> mk_singleton Q.one
    | n -> mult l (loop (n - 1))
  in
```

```
let c = loop n in
if n mod 2 = 0 then
  inter c nonneg
else
  c
```

178. Comparing two intervals.

```
let sub i j =
  let lo_ge (a, alpha) (c, gamma) =
    match Extq.cmp a c with
      | Mpa.Q.Less → false
      | Mpa.Q.Greater → true
      | Mpa.Q.Equal → ¬ alpha ∨ gamma
  in
  let hi_le (b, beta) (d, delta) =
    match Extq.cmp b d with
      | Mpa.Q.Less → true
      | Mpa.Q.Greater → false
      | Mpa.Q.Equal → ¬ beta ∨ delta
  in
  Dom.sub i.dom j.dom ∧
  lo_ge (Endpoint.destruct i.lo) (Endpoint.destruct j.lo) ∧
  hi_le (Endpoint.destruct i.hi) (Endpoint.destruct j.hi)

let disjoint i j =
  let (b, beta) = Endpoint.destruct i.hi
  and (c, gamma) = Endpoint.destruct j.lo in
  match Extq.cmp b c with
    | Mpa.Q.Less → true
    | Mpa.Q.Equal → ¬ (beta ∧ gamma)
    | Mpa.Q.Greater →
        let (d, delta) = Endpoint.destruct j.hi
        and (a, alpha) = Endpoint.destruct i.lo in
        (match Extq.cmp d a with
          | Mpa.Q.Less → true
          | Mpa.Q.Equal → ¬ (delta ∧ alpha)
          | Mpa.Q.Greater → false)

let rec cmp i j =
  if eq i j then
    Binrel.Same
  else if sub i j then
    Binrel.Sub
  else if sub j i then
    Binrel.Super
```

```
  else
    let (b, beta) = Endpoint.destruct i.hi
    and (c, gamma) = Endpoint.destruct j.lo in
    match Extq.cmp b c with
      | Q.Less → Binrel.Disjoint
      | Q.Equal when beta ∧ gamma → Binrel.Singleton (to_q b)
      | Q.Equal → Binrel.Disjoint
      | Q.Greater →
          let (d, delta) = Endpoint.destruct j.hi
          and (a, alpha) = Endpoint.destruct i.lo in
          (match Extq.cmp d a with
             | Q.Less → Binrel.Disjoint
             | Q.Equal when delta ∧ alpha → Binrel.Singleton (to_q a)
             | Q.Equal → Binrel.Disjoint
             | Q.Greater → Binrel.Overlap (inter i j))
and to_q x =
  assert(Extq.is_q x);
  match Extq.to_q x with
    | Some(q) → q
    | None → assert false
```

6.16 Module cnstrnt

Interface for module Cnstrnt.mli

179. Module *Cnstrnt*: real number constraints.

```
type t
```

180. Constructing and destructing intervals.

```
module Diseqs : (Set.S with type elt = Q.t)
val make : Interval.t × Diseqs.t → t
val destruct : t → Interval.t × Diseqs.t
val dom_of : t → Dom.t
val endpoints_of : t → Endpoint.t × Endpoint.t
```

181. Test if the interval part is bounded by a finite bound.

```
val is_unbounded : t → bool
```

182. Has finite extension.

```
val is_finite : t → bool
```

183. Constraint extension contains only positive numbers.

val $is\_pos\ :\ t\ \rightarrow\ bool$

Constraint extension contains only negative numbers.

val $is\_neg\ :\ t\ \rightarrow\ bool$

184. Derived Constructors.

val $mk\_real\ :\ t$
val $mk\_int\ :\ t$
val $mk\_nonint\ :\ t$
val $mk\_nat\ :\ t$

val $mk\_singleton\ :\ Mpa.Q.t\ \rightarrow\ t$
val $mk\_zero\ :\ t$
val $mk\_one\ :\ t$

val $mk\_diseq\ :\ Mpa.Q.t\ \rightarrow\ t$

185. Membership.

val $mem\ :\ Mpa.Q.t\ \rightarrow\ t\ \rightarrow\ bool$

186. Recognizers and Accessors.

val $is\_empty\ :\ t\ \rightarrow\ bool$
val $is\_full\ :\ t\ \rightarrow\ bool$

val $d\_singleton\ :\ t\ \rightarrow\ Mpa.Q.t\ option$

val $d\_lower\ :\ t\ \rightarrow\ (Dom.t\ \times\ bool \times Mpa.Q.t)\ option$

val $d\_upper\ :\ t\ \rightarrow\ (Dom.t\ \times\ Mpa.Q.t\ \times\ bool)\ option$

187. Equality.

val $eq\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$

188. Comparison of constraints.

val $cmp\ :\ t\ \rightarrow\ t\ \rightarrow\ t\ Binrel.t$

189. Test for disjointness.

val $is\_disjoint\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$

190. Test for emptyness, singleton, etc.

val $status\ :\ t\ \rightarrow\ Mpa.Q.t\ Status.t$

Subconstraint.

val $sub\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$

191. Intersection of constraints.

val $inter\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

192. Pretty-printing constraints.

val $pp\ :\ Format.formatter\ \rightarrow\ t\ \rightarrow\ unit$

193. Construct a constraint from an interval.

val $of\_interval\ :\ Interval.t\ \rightarrow\ t$

194. Additional constructors.

val $mk\_oo\ :\ Dom.t\ \rightarrow\ Q.t\ \rightarrow\ Q.t\ \rightarrow\ t$
val $mk\_oc\ :\ Dom.t\ \rightarrow\ Q.t\ \rightarrow\ Q.t\ \rightarrow\ t$
val $mk\_co\ :\ Dom.t\ \rightarrow\ Q.t\ \rightarrow\ Q.t\ \rightarrow\ t$
val $mk\_cc\ :\ Dom.t\ \rightarrow\ Q.t\ \rightarrow\ Q.t\ \rightarrow\ t$

val $mk\_lower\ :\ Dom.t\ \rightarrow\ Q.t\ \times\ bool \rightarrow\ t$
val $mk\_upper\ :\ Dom.t\ \rightarrow\ bool \times Q.t\ \rightarrow\ t$

val $mk\_lt\ :\ Dom.t\ \rightarrow\ Q.t\ \rightarrow\ t$
val $mk\_le\ :\ Dom.t\ \rightarrow\ Q.t\ \rightarrow\ t$
val $mk\_gt\ :\ Dom.t\ \rightarrow\ Q.t\ \rightarrow\ t$
val $mk\_ge\ :\ Dom.t\ \rightarrow\ Q.t\ \rightarrow\ t$

val $mk\_pos\ :\ Dom.t\ \rightarrow\ t$
val $mk\_neg\ :\ Dom.t\ \rightarrow\ t$
val $mk\_nonneg\ :\ Dom.t\ \rightarrow\ t$
val $mk\_nonpos\ :\ Dom.t\ \rightarrow\ t$

195. Abstract interpretation.

val $addq\ :\ Q.t\ \rightarrow\ t\ \rightarrow\ t$
val $add\ :\ t\ \rightarrow\ t\ \rightarrow\ t$
val $addl\ :\ t\ list \rightarrow\ t$
val $subtract\ :\ t\ \rightarrow\ t\ \rightarrow\ t$
val $mult\ :\ t\ \rightarrow\ t\ \rightarrow\ t$
val $multl\ :\ t\ list \rightarrow\ t$
val $expt\ :\ int \rightarrow\ t\ \rightarrow\ t$
val $multq\ :\ Q.t\ \rightarrow\ t\ \rightarrow\ t$
val $div\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

Module Cnstrnt.ml

196. Set of rational numbers.

module $Diseqs\ =\ Set.Make($
  struct

```
    type t = Q.t
    let compare = Q.compare
  end)
```

197. A constraint consists an interval and a set of rational numbers.

```
type t = Interval.t × Diseqs.t

let destruct c = c

let eq (i, qs) (j, ps) =
  Interval.eq i j ∧ Diseqs.equal qs ps

let dom_of (i, _) =
  let (d, _, _) = Interval.destructure i in
  d

let endpoints_of (i, _) =
  let (_, lo, hi) = Interval.destructure i in
    (lo, hi)

let is_unbounded (i, _) =
  let (_, lo, hi) = Interval.destructure i in
    Endpoint.eq Endpoint.neginf lo ∧
    Endpoint.eq Endpoint.posinf hi
```

198. Empty constraint.

```
let mk_empty = (Interval.mk_empty, Diseqs.empty)

let is_empty (i, _) = Interval.is_empty i

let is_full (i, qs) =
  Interval.is_full i ∧ Diseqs.is_empty qs

let is_finite (i, _) =
  let (d, l, h) = Interval.destructure i in
  d = Dom.Int ∧ Endpoint.is_q l ∧ Endpoint.is_q h

let is_pos (i, _) =
  let (eq, alpha) = Endpoint.destruct (Interval.lo i) in
    match Extq.destruct eq with
      | Extq.Posinf → true
      | Extq.Inject(q) →
          if alpha then
            Mpa.Q.gt q Mpa.Q.zero
          else
            Mpa.Q.ge q Mpa.Q.zero
      | _ → false

let is_neg (i, _) =
  let (eq, beta) = Endpoint.destruct (Interval.hi i) in
```

```
match Extq.destruct eq with
  | Extq.Neginf → true
  | Extq.Inject(q) →
      if beta then
        Mpa.Q.lt q Mpa.Q.zero
      else
        Mpa.Q.le q Mpa.Q.zero
  | _ → false
```

199. Membership.

```
let mem q (i, qs) =
  Interval.mem q i ∧ ¬(Diseqs.mem q qs)
```

200. Constructing a constraint from components

```
let of_interval i = (i, Diseqs.empty)

exception Found of Mpa.Q.t

let rec make (i, qs) =
  if Interval.is_empty i then
    mk_empty
  else if Diseqs.is_empty qs then
    (i, Diseqs.empty)
  else
    let (d, lo, hi) = Interval.destructure i in
    if d = Dom.Int then
      let (a, alpha) = Endpoint.destruct lo in
      match alpha, endpoint a qs with
        | true, Some(p) →
            let i' = Interval.make (Dom.Int, Endpoint.make (a,false), hi) in
            make (i', Diseqs.remove p qs)
        | _ →
            let (b, beta) = Endpoint.destruct hi in
            match beta, endpoint b qs with
              | true, Some(p) →
                  let i' = Interval.make (Dom.Int, lo, Endpoint.make (b,false)) in
                  make (i', Diseqs.remove p qs)
              | _ →
                  normalize (i, qs)
    else
      normalize (i, qs)

and endpoint a qs =
  match Extq.to_q a with
    | None → None
    | Some(q) →
```

```
        try
          Diseqs.iter
            (fun p →
               if Mpa.Q.equal q p then raise (Found p))
            qs;
          None
        with
            Found(p) → Some(p)

and normalize (i, qs) =
  let qs' = Diseqs.filter (fun q → Interval.mem q i) qs in
  (i, qs')
```

201. Constraint for the real number line, the integers, and the natural numbers.

```
let mk_real =
  of_interval Interval.mk_real

let mk_int =
  of_interval Interval.mk_int

let mk_nonint =
  of_interval Interval.mk_nonint

let mk_nat =
  let i = Interval.make (Dom.Int, Endpoint.nonstrict Q.zero, Endpoint.posinf) in
  of_interval i
```

202. Constructing singleton constraints.

```
let mk_singleton q =
  of_interval (Interval.mk_singleton q)

let mk_zero = mk_singleton Mpa.Q.zero
let mk_one = mk_singleton Mpa.Q.one

let d_singleton (i, qs) =
  match Interval.d_singleton i with
    | (Some(q) as res)
        when ¬(Diseqs.mem q qs) →
        res
    | _ →
        None

let d_lower (i, qs) =
  if ¬(Diseqs.is_empty qs) then None else
    let (dom, lo, hi) = Interval.destructure i in
    let (a, alpha) = Endpoint.destruct lo in
    let (b, _) = Endpoint.destruct hi in
      match Extq.destruct a, Extq.destruct b with
```

```
        | Extq.Inject(q), Extq.Posinf →
            Some(dom, alpha, q)
        | _ →
            None

let d_upper (i, qs) =
  if ¬(Diseqs.is_empty qs) then None else
    let (dom, lo, hi) = Interval.destructure i in
    let (a, _) = Endpoint.destruct lo in
    let (b, beta) = Endpoint.destruct hi in
      match Extq.destruct a, Extq.destruct b with
        | Extq.Neginf, Extq.Inject(p) →
            Some(dom, p, beta)
        | _ →
            None

let mk_zero = mk_singleton Mpa.Q.zero
let mk_one = mk_singleton Mpa.Q.one
```

203. Disequality constraint.

```
let mk_diseq q = (Interval.mk_real, Diseqs.singleton q)
```

204. Checks wether c is a subconstraint of d.

```
let sub (i, qs) (j, ps) =
  Interval.sub i j ∧
  Diseqs.subset ps qs
```

205. Intersection of two constraints

```
let inter (i, qs) (j, ps) =
  make (Interval.inter i j, Diseqs.union qs ps)
```

206. Comparison.

```
let rec cmp c d =
  let (i, qs) = destruct c in
  let (j, ps) = destruct d in
  match Interval.cmp i j with
    | Binrel.Disjoint →
        Binrel.Disjoint
    | Binrel.Overlap(k) →
        analyze (make (k, Diseqs.union qs ps))
    | Binrel.Same when Diseqs.equal qs ps →
        Binrel.Same
    | Binrel.Same when Diseqs.subset qs ps →
        Binrel.Super
    | Binrel.Same when Diseqs.subset ps qs →
```

```
        Binrel.Sub
      | Binrel.Same →
          analyze (make (i, Diseqs.union qs ps))
      | Binrel.Singleton(q) when Diseqs.mem q qs ∨ Diseqs.mem q ps →
          Binrel.Disjoint
      | Binrel.Singleton(q) →
          Binrel.Singleton(q)
      | Binrel.Sub when Diseqs.subset ps qs →
          Binrel.Sub
      | Binrel.Sub →
          analyze (make (i, Diseqs.union qs ps))
      | Binrel.Super when Diseqs.subset qs ps →
          Binrel.Super
      | Binrel.Super →
          analyze (make (j, Diseqs.union qs ps))

and analyze c =
  if is_empty c then
    Binrel.Disjoint
  else
    match d_singleton c with
      | Some(q) → Binrel.Singleton(q)
      | None → Binrel.Overlap(c)
```

207. Status.

```
let status c =
  if is_empty c then
    Status.Empty
  else
    match d_singleton c with
      | Some(q) → Status.Singleton(q)
      | None → Status.Other
```

208. Are c and d disjoint.

```
let is_disjoint c d =
  is_empty (inter c d)
```

209. Printing constraints.

```
let pp fmt c =
  let (i, qs) = destruct c in
  Format.fprintf fmt "@[";
  Interval.pp fmt i;
  if ¬(Diseqs.is_empty qs) then
    begin
```

```
    Format.fprintf fmt "\\";
    Pretty.set Mpa.Q.pp fmt (Diseqs.elements qs)
  end
```

210. Additional constructors.

```
let of_endpoints (dom, lo, hi) =
  of_interval (Interval.make (dom, lo, hi))

let mk_oo dom u v = of_endpoints (dom, Endpoint.strict u, Endpoint.strict v)
let mk_oc dom u v = of_endpoints (dom, Endpoint.strict u, Endpoint.nonstrict v)
let mk_co dom u v = of_endpoints (dom, Endpoint.nonstrict u, Endpoint.strict v)
let mk_cc dom u v = of_endpoints (dom, Endpoint.nonstrict u, Endpoint.nonstrict v)

let mk_lower dom (u, beta) =
  of_endpoints (dom, Endpoint.neginf, Endpoint.make (Extq.of_q u, beta))

let mk_upper dom (alpha, u) =
  of_endpoints (dom, Endpoint.make (Extq.of_q u, alpha), Endpoint.posinf)

let mk_lt dom u = of_endpoints (dom, Endpoint.neginf, Endpoint.strict u)
let mk_le dom u = of_endpoints (dom, Endpoint.neginf, Endpoint.nonstrict u)
let mk_gt dom u = of_endpoints (dom, Endpoint.strict u, Endpoint.posinf)
let mk_ge dom u = of_endpoints (dom, Endpoint.nonstrict u, Endpoint.posinf)

let mk_neg dom = mk_lt dom Q.zero
let mk_pos dom = mk_gt dom Q.zero
let mk_nonneg dom = mk_ge dom Q.zero
let mk_nonpos dom = mk_le dom Q.zero
```

211. Abstract interpretation.

```
let addq q (j, ps) =
  if Mpa.Q.is_zero q then
    make (j, ps)
  else
    let i = Interval.mk_singleton q in
    let j' = Interval.add i j in
    let ps' = Diseqs.fold (fun p → Diseqs.add (Q.add q p)) ps Diseqs.empty in
      make (j', ps')

let add (i, _) (j, _) =
  of_interval (Interval.add i j)

let subtract (i, _) (j, _) =
  of_interval (Interval.subtract i j)

let rec addl = function
  | [] → mk_zero
  | [c] → c
  | [c; d] → add c d
```

```
  | c :: cl → add c (addl cl)
let multq q ((i, qs) as c) =
  if Mpa.Q.equal Mpa.Q.one q then
    make (i, qs)
  else if Mpa.Q.equal Mpa.Q.zero q then
    mk_zero
  else if Interval.is_full i then
    if Diseqs.is_empty qs then
      c
    else
      let qs' = Diseqs.fold (fun p → Diseqs.add (Mpa.Q.mult q p)) qs Diseqs.empty in
      make (i, qs')
  else
    let j' = Interval.multq q i in
    let qs' = Diseqs.fold (fun p → Diseqs.add (Mpa.Q.mult q p)) qs Diseqs.empty in
      make (j', qs')
let mult (i, _) (j, _) =
  of_interval (Interval.mult i j)
let rec multl = function
  | [] → mk_singleton Q.one
  | [c] → c
  | c :: cl → mult c (multl cl)
let expt n (i, _) =
  if n = 0 then
    mk_one
  else if n < 0 then
    mk_real
  else
    of_interval (Interval.expt n i)
let div (i, _) (j, _) = mk_real
```

6.17 Module name

Interface for module Name.mli

212. Module *Name*: Datatype of names.

```
type t
val of_string : string → t
val to_string : t → string
val eq : t → t → bool
```

```
val cmp : t → t → int
val pp : Format.formatter → t → unit
val hash : t → int
module Set : (Set.S with type elt = t)
module Map : (Map.S with type key = t)
val pp_map : (Format.formatter → α → unit)
               → Format.formatter → α Map.t → unit
```

Module Name.ml

```
type t = string
let of_string s = s
let to_string s = s
let eq = (=)
let cmp n m =
  Pervasives.compare n m
let pp fmt s =
  Format.fprintf fmt "%s" s
let hash = Hashtbl.hash
type name = t (* avoid type-check error below *)
module Set = Set.Make(
  struct
    type t = name
    let compare = Pervasives.compare
  end)
module Map = Map.Make(
  struct
    type t = name
    let compare = Pervasives.compare
  end)
let pp_map p fmt =
  Map.iter
    (fun x y →
       pp fmt x;
       Format.fprintf fmt " |-> ";
       p fmt y;
       Format.fprintf fmt "\n")
```

6.18 Module var

Interface for module Var.mli

213. Module *Var*: datatype for variables.

type t

val $name\_of$: $t \rightarrow Name.t$

val eq : $t \rightarrow t \rightarrow bool$

val cmp : $t \rightarrow t \rightarrow int$

val $(<<<)$: $t \rightarrow t \rightarrow bool$

214. Constructors.

val mk_var : $Name.t \rightarrow t$

val k : $int\ ref$

val mk_fresh : $Name.t \rightarrow int\ option \rightarrow t$

val mk_slack : $int\ option \rightarrow t$

val mk_free : $int \rightarrow t$

215. Recognizers.

val is_var : $t \rightarrow bool$
val is_fresh : $t \rightarrow bool$
val is_slack : $t \rightarrow bool$
val is_free : $t \rightarrow bool$

216. Get index of a bound variable.

val d_free : $t \rightarrow int$

217. Printing variables.

val pp : $Format.formatter \rightarrow t \rightarrow unit$

218. Sets and maps of terms.

type $var = t$

module Set : ($Set.S$ with type $elt = var$)

module Map : ($Map.S$ with type $key = var$)

Module Var.ml

219. Variables.

```
type t =
  | External of Name.t
  | Internal of Name.t × int
  | Bound of int

let name_of = function
  | External(n) → n
  | Internal(n, i) →
      let str = Format.sprintf "%s!%d" (Name.to_string n) i in
        Name.of_string str
  | Bound(n) →
      let str = Format.sprintf "!%d" n in
        Name.of_string str

let eq x y =
  match x, y with
    | External(n), External(m) →
        Name.eq n m
    | Internal(n, i), Internal(m, j) →
        Name.eq n m ∧ i = j
    | Bound(n), Bound(m) →
        n = m
    | _ →
        false

let cmp x y =
  match x, y with
    | External _, Internal _ → −1
    | Internal _, External _ → 1
    | External(n), External(m) →
        Name.cmp n m
    | Internal(n, i), Internal(m, j) →
        let c1 = Name.cmp n m in
        if c1 ≢ 0 then c1 else Pervasives.compare i j
    | Bound(n), Bound(m) →
        Pervasives.compare n m
    | _ →
        Pervasives.compare x y

let (<<<) x y = (cmp x y ≤ 0)
```

220. Sets and maps of terms.

```
type var = t
```

```
module Set = Set.Make(
  struct
    type t = var
    let compare = cmp
  end)

module Map = Map.Make(
  struct
    type t = var
    let compare = cmp
  end)
```

221. Constructors.

```
let mk_var x = External(x)

let k = ref 0
let _ = Tools.add_at_reset (fun () → k := 0)

let mk_fresh x = function
  | Some(k) →
      Internal(x, k)
  | None →
      incr(k);
      Internal(x, !k)

let mk_slack =
  let slackname = Name.of_string "k" in
    mk_fresh slackname

let mk_free i = Bound(i)
```

222. Recognizers.

```
let is_var = function External _ → true | _ → false
let is_fresh = function Internal _ → true | _ → false
let is_free = function Bound _ → true | _ → false

let d_free = function
  | Bound(i) → i
  | _ → assert false

let is_slack =
  let slackname = Name.of_string "k" in
    function
      | Internal(k, _) →
          Name.eq k slackname
      | _ →
          false
```

223. Printer.

let $pp\ fmt\ x\ =$
  $Name.pp\ fmt\ (name\_of\ x)$

6.19 Module sym

Interface for module Sym.mli

Author: Harald Ruess

* Module *Sym*: Interpreted and uninterpreted function symbols
* Interpreted symbols

type $arith\ =$
  $\mid Num$ of $Mpa.Q.t$
  $\mid Add$
  $\mid Multq$ of $Mpa.Q.t$

type $product\ =$
  $\mid Tuple$
  $\mid Proj$ of $int \times int$

type $coproduct\ =\ InL \mid InR \mid OutL \mid OutR$

type $bv\ =$
  $\mid Const$ of $Bitv.t$
  $\mid Conc$ of $int \times int$
  $\mid Sub$ of $int \times int \times int$
  $\mid Bitwise$ of int

type $pprod\ =$
  $\mid Mult$
  $\mid Expt$ of int

type $apply\ =$
  $\mid Apply$ of $Cnstrnt.t\ option$
  $\mid Abs$

type $arrays\ =$
  $\mid Select$
  $\mid Update$

type $bvarith\ =$

| *Unsigned*

* Symbols

type t =
  | *Uninterp* of *Name.t* (∗ Uninterpreted function symbols. ∗)
  | *Arith* of *arith* (∗ Linear arithmetic function symbols. ∗)
  | *Product* of *product* (∗ N-ary products ∗)
  | *Coproduct* of *coproduct* (∗ 2-ary coproducts ∗)
  | *Bv* of *bv* (∗ Bitvector function symbols. ∗)
  | *Pp* of *pprod* (∗ Power products. ∗)
  | *Fun* of *apply* (∗ Lambda abstraction and application ∗)
  | *Arrays* of *arrays* (∗ Theory of arrays. ∗)
  | *Bvarith* of *bvarith* (∗ Bitvector interpretations. ∗)

val *eq* : $t \rightarrow t \rightarrow$ *bool*
(∗* Equality test ∗)

val *cmp* : $t \rightarrow t \rightarrow$ *int*
(∗* Comparison. ∗)

val *pp* : *Format.formatter* $\rightarrow t \rightarrow$ *unit*
(∗* Pretty printing ∗)

val *width* : $t \rightarrow$ *int option*
(∗* Width of a bitvector symbol. ∗)

* Miscellaneous symbols

val *tuple* : t
val *car* : t
val *cdr* : t

Module Sym.ml

224. Interpreted symbols.

type *arith* =
  | *Num* of *Mpa.Q.t*
  | *Add*
  | *Multq* of *Mpa.Q.t*

type *product* =
  | *Tuple*
  | *Proj* of *int* × *int*

type *coproduct* = *InL* | *InR* | *OutL* | *OutR*

type *bv* =
  | *Const* of *Bitv.t*

```
  | Conc of int × int
  | Sub of int × int × int
  | Bitwise of int

type pprod =
  | Mult
  | Expt of int

type apply =
  | Apply of Cnstrnt.t option
  | Abs

type arrays =
  | Select
  | Update

type bvarith =
  | Unsigned
```

225. Symbols.

```
type t =
  | Uninterp of Name.t
  | Arith of arith
  | Product of product
  | Coproduct of coproduct
  | Bv of bv
  | Pp of pprod
  | Fun of apply
  | Arrays of arrays
  | Bvarith of bvarith
```

226. Comparison.

```
let cmp = Pervasives.compare
```

227. Equality.

```
let rec eq s t =
  match s, t with
    | Uninterp(x), Uninterp(y) → Name.eq x y
    | Arith(f), Arith(g) → eq_arith f g
    | Product(f), Product(g) → eq_product f g
    | Coproduct(op1), Coproduct(op2) → op1 = op2
    | Bv(f), Bv(g) → eq_bv f g
    | Pp(f), Pp(g) → eq_pp f g
    | Fun(f), Fun(g) → eq_apply f g
    | Bvarith(f), Bvarith(g) → eq_interp f g
    | Arrays(f), Arrays(g) → eq_arrays f g
```

```
    | _ → false
and eq_interp f g =
  match f, g with
    | Unsigned, Unsigned → true
and eq_bv f g =
  match f, g with
    | Const(b1), Const(b2) →
        Pervasives.compare b1 b2 = 0
    | Conc(n1, m1), Conc(n2, m2) →
        n1 = n2 ∧ m1 = m2
    | Sub(n1, i1, j1), Sub(n2, i2, j2) →
        n1 = n2 ∧ i1 = i2 ∧ j1 = j2
    | Bitwise(n1), Bitwise n2 →
        n1 = n2
    | _ → false
and eq_product f g =
  match f, g with
    | Tuple, Tuple → true
    | Proj(i, n), Proj(j, m) → i = j ∧ n = m
    | _ → false
and eq_apply f g =
  match f, g with
    | Apply _, Apply _ → true
    | Abs, Abs → true
    | _ → false
and eq_arith f g =
  match f, g with
    | Num(q1), Num(q2) → Q.equal q1 q2
    | Multq(q1), Multq(q2) → Q.equal q1 q2
    | Add, Add → true
    | _ → false
and eq_arrays f g =
  match f, g with
    | Select, Select → true
    | Update, Update → true
    | _ → false
and eq_pp f g =
  match f, g with
    | Expt(n), Expt(m) → n = m
    | Mult, Mult → true
    | _ → false
```

```
let pp fmt s =
  let rec sym s =
    match s with
      | Uninterp(f) → Name.pp fmt f
      | Arith(op) → arith op
      | Product(op) → product op
      | Bv(op) → bv op
      | Coproduct(op) → coproduct op
      | Arrays(op) → array op
      | Pp(op) → pprod op
      | Fun(op) → apply op
      | Bvarith(op) → interp op
  and arith op =
    match op with
      | Num(q) → Mpa.Q.pp fmt q
      | Add → Format.fprintf fmt "+"
      | Multq(q) → Mpa.Q.pp fmt q; Format.fprintf fmt "*"
  and product op =
    match op with
      | Tuple → Format.fprintf fmt "tuple"
      | Proj(i, n) → Format.fprintf fmt "proj[%d,%d]" i n
  and coproduct op =
    match op with
      | InL → Format.fprintf fmt "inl"
      | InR → Format.fprintf fmt "inr"
      | OutL → Format.fprintf fmt "outl"
      | OutR → Format.fprintf fmt "outr"
  and bv op =
    match op with
      | Const(b) → Format.fprintf fmt "0b%s" (Bitv.to_string b)
      | Conc(n, m) → Format.fprintf fmt "conc[%d,%d]" n m
      | Sub(n, i, j) → Format.fprintf fmt "sub[%d,%d,%d]" n i j
      | Bitwise(n) → Format.fprintf fmt "ite[%d]" n
  and array op =
    match op with
      | Select → Format.fprintf fmt "select"
      | Update → Format.fprintf fmt "update"
  and interp op =
    match op with
      | Unsigned → Format.fprintf fmt "unsigned"
  and apply op =
    match op with
```

```
    | Apply(Some(c)) ->
        Pretty.string fmt ("apply[" ^ Pretty.to_string Cnstrnt.pp c ^ "]")
    | Apply(None) ->
        Pretty.string fmt "apply"
    | Abs ->
        Pretty.string fmt "lambda"
and pprod op =
  match op with
    | Mult ->
        Format.fprintf fmt "."
    | Expt(n) ->
        Format.fprintf fmt "^%d" n
in
sym s
```

228. Width of bitvector function symbols.

```
let rec width f =
  match f with
    | Bv(b) -> Some(width_bv b)
    | _ -> None

and width_bv b =
  match b with
    | Const(c) ->
        Bitv.length c
    | Sub(n, i, j) ->
        assert(0 ≤ i ∧ i ≤ j ∧ j < n);
        j - i + 1
    | Conc(n, m) ->
        assert(0 ≤ n ∧ 0 ≤ m);
        n + m
    | Bitwise(n) ->
        assert(n ≥ 0);
        n
```

229. Predefined symbol.

```
let add = Arith(Add)
```

230. Function symbols for tuple theory.

```
let tuple = Product(Tuple)
let car = Product(Proj(0, 2))
let cdr = Product(Proj(1, 2))
```

6.20 Module term

Interface for module Term.mli

231. Terms are the basic data structures of ICS.

232. Terms. A term is either a variable $Var(s)$, where the name s is a string, an application $App(f, l)$ of a 'function symbol' to a list of arguments, an update expression $Update(a, i, v)$, or a term interpreted in one of the theories of linear arithmetic *Arith*, propositional logic, *Prop*, propositional sets *Set*, tuples *Tuple*, or bitvectors *Bv*. By definition, all entitities of type t are hash-consed. Therefore, equality tests between terms can be done in constant time using the equality $===$ from the module *Hashcons*.

Arithmetic terms are either numerals of the form $Num(q)$, n-ary addition $Add(l)$, linear multiplication $Multq(q, a)$, nonlinear multiplication $Mult(l)$, or division. Arithmetic terms built up solely from *Num*, *Add*, and *Multq* are considered to be interpreted, since *Mult* and *Div* are considered to be uninterpreted, in general. However, certain simplification rules for these uninterpreted function symbols are built-in.

Propositional terms are either *False*, *True*, or conditionals $Ite(a, b, c)$. Other propositional connectives can be encoded using these constructor.

A tuple term is either a tuple $Tup(l)$ or the i-th projection $Proj(i, n, \_)$ from an n-tuple.

Set of terms are implemented using Patricia trees. Operations on these sets are described below in the submodule *Set*.

```
type t =
  | Var of Var.t
  | App of Sym.t × t list
```

233. Constructing and destructing terms

```
val mk_var : Name.t → t
val mk_const : Sym.t → t
val mk_app : Sym.t → t list → t

val mk_fresh_var : Name.t → int option → t
val is_fresh_var : t → bool
val name_of : t → Name.t

val destruct : t → Sym.t × t list

val sym_of : t → Sym.t
val args_of : t → t list
```

234. Equality of terms.

```
val eq : t → t → bool

val eql : t list → t list → bool

val cmp : t → t → int
```

val $(<<<) : t \rightarrow t \rightarrow bool$

val $orient : t \times t \rightarrow t \times t$

val $max : t \rightarrow t \rightarrow t$
val $min : t \rightarrow t \rightarrow t$

235. Test if term is a constant.

val $is\_const : t \rightarrow bool$

val $is\_var : t \rightarrow bool$
val $is\_app : t \rightarrow bool$

val $to\_var : t \rightarrow Var.t$

val $is\_interp\_const : t \rightarrow bool$

val $is\_equal : t \rightarrow t \rightarrow Three.t$

236. Fold operator on terms.

val $fold : (t \rightarrow \alpha \rightarrow \alpha) \rightarrow t \rightarrow \alpha \rightarrow \alpha$

237. Iteration operator on terms.

val $iter : (t \rightarrow unit) \rightarrow t \rightarrow unit$

238. Predicate holds for all subterms.

val $for\_all : (t \rightarrow bool) \rightarrow t \rightarrow bool$

239. $subterm\ a\ b$ holds if a is a subterm of b.

val $subterm : t \rightarrow t \rightarrow bool$

240. $occurs\ x\ a$ holds if term x occurs in a.

val $occurs : t \rightarrow t \rightarrow bool$

241. Mapping over list of terms. Avoids unnecessary consing.

val $mapl : (t \rightarrow t) \rightarrow t\ list \rightarrow t\ list$

242. Association lists for terms.

val $assq : t \rightarrow (t \times \alpha)\ list \rightarrow \alpha$

243. Printer.

val $pretty : bool\ ref$

val $pp : Format.formatter \rightarrow t \rightarrow unit$

val $to\_string : t \rightarrow string$

244. Pretty-printing pairs as equalities/disequalities/constraints.

```
val pp_equal : (t × t) Pretty.printer
val pp_diseq : (t × t) Pretty.printer
val pp_in : (t × Cnstrnt.t) Pretty.printer
```

245. Sets and maps of terms.

```
type trm = t

module Set : (Set.S with type elt = trm)

module Map : (Map.S with type key = trm)
```

246. Return set of variables.

```
val vars_of : t → Set.t
```

Module Term.ml

247. Terms.

```
type t =
  | Var of Var.t
  | App of Sym.t × t list

let rec eq a b =
  match a, b with
    | Var(x), Var(y) →
        Var.eq x y
    | App(f, l), App(g, m) →
        Sym.eq f g ∧ eql l m
    | _ →
        false

and eql al bl =
  try List.for_all2 eq al bl with Invalid_argument _ → false
```

248. Constructors.

```
let mk_var x = Var(Var.mk_var x)

let mk_const f = App(f, [])
let mk_app f l = App(f, l)

let mk_fresh_var x k = Var(Var.mk_fresh x k)

let is_fresh_var = function
  | Var(x) → Var.is_fresh x
  | _ → false
```

249. Recognizers.

```
let is_var = function Var _ → true | _ → false
let is_app = function App _ → true | _ → false
let is_const = function App(_, []) → true | _ → false
```

250. Destructors.

```
let to_var = function
  | Var(x) → x
  | _ → assert false

let name_of a =
  assert(is_var a);
  match a with Var(x) → Var.name_of x | _ → assert false

let destruct a =
  assert(is_app a);
  match a with App(f, l) → (f, l) | _ → assert false

let sym_of a =
  assert(is_app a);
  match a with App(f, _) → f | _ → assert false

let args_of a =
  assert(is_app a);
  match a with App(_, l) → l | _ → assert false
```

251. Structural comparison.

```
let rec cmp a b =
  match a, b with
    | Var _, App _ → 1
    | App _, Var _ → −1
    | Var(x), Var(y) → Var.cmp x y
    | App(f, l), App(g, m) →
        let c1 = Sym.cmp f g in
        if c1 ≢ 0 then c1 else cmpl l m

and cmpl l m =
  let rec loop c l m =
    match l, m with
      | [], [] → c
      | [], _ → −1
      | _, [] → 1
      | x :: xl, y :: yl →
          if c ≢ 0 then loop c xl yl else loop (cmp x y) xl yl
  in
  loop 0 l m
```

```
let (<<<) a b = (cmp a b ≤ 0)
```

252. *cmp* forces function applications to be smaller than constants. Thus, *cmp* makes sure that e.g. $f(x) = c$ is added in this order. Otherwise, the term ordering is arbitrary and we use *Term.cmp* for ordering all the other cases.

```
let orient ((a, b) as e) =
  if cmp a b ≥ 0 then e else (b, a)

let min a b =
  if a <<< b then a else b

let max a b =
  if a <<< b then b else a
```

253. Some recognizers.

```
let is_interp_const = function
  | App((Arith _ | Bv _ | Product _), []) → true
  | _ → false

let is_interp = function
  | App((Arith _ | Bv _ | Product _), _) → true
  | _ → false

let is_uninterpreted = function
  | App(Uninterp _, _) → true
  | _ → false

let is_equal a b =
  if eq a b then
    Three.Yes
  else match a, b with (* constants from within a theory are *)
    | App((Arith _ as c), []), App((Arith _ as d), []) (* assumed to interpreted differently
*)
        when ¬(Sym.eq c d) → Three.No
    | App((Bv _ as c), []), App((Bv _ as d), [])
        when ¬(Sym.eq c d) → Three.No
    | _ →
        Three.X
```

254. Mapping over list of terms. Avoids unnecessary consing.

```
let rec mapl f l =
  match l with
    | [] → []
    | a :: l1 →
        let a' = f a and l1' = mapl f l1 in
        if eq a' a ∧ l1 ≡ l1' then l else a' :: l1'
```

255. Association lists for terms.

```
let rec assq a = function
  | [] → raise Not_found
  | (x, y) :: xl → if eq a x then y else assq a xl
```

256. Iteration over terms.

```
let rec fold f a acc =
  match a with
    | Var _ → f a acc
    | App(_, l) → f a (List.fold_right (fold f) l acc)

let rec iter f a =
  f a;
  if is_app a then
    List.iter (iter f) (args_of a)

let rec for_all p a =
  p a ∧
  match a with
    | Var _ → true
    | App(_, l) → List.for_all (for_all p) l

let rec subterm a b =
  eq a b ∨
  match b with
    | Var _ → false
    | App(_, l) → List.exists (subterm a) (args_of b)

let occurs x b = subterm x b
```

257. Printer.

```
let pretty = ref true (* Infix/Mixfix output when pretty is true. *)

let rec pp fmt a =
  let str = Pretty.string fmt in
  let term = pp fmt in
  let args = Pretty.tuple pp fmt in
  let app f l = Sym.pp fmt f; Pretty.tuple pp fmt l in
  let infixl x = Pretty.infixl pp x fmt in
  match a with
    | Var(x) → Var.pp fmt x
    | App(f, l) when ¬(!pretty) → app f l
    | App(f, l) →
        (match f, l with
           | Arith(Num q), [] →
                Mpa.Q.pp fmt q
           | Arith(Add), _ →
                infixl "␣+␣" l
```

```
  | Arith(Multq(q)) , [x] →
      Pretty.infix Mpa.Q.pp "*" pp fmt (q, x)
  | Product(Proj(0, 2)), [App(Coproduct(OutR), [x])] →
      str "hd"; str "("; term x; str ")"
  | Product(Proj(1, 2)), [App(Coproduct(OutR), [x])] →
      str "tl"; str "("; term x; str ")"
  | Product(Proj(0,2)), [_] →
      str "car"; args l
  | Product(Proj(1,2)), [_] →
      str "cdr"; args l
  | Product(Tuple), [_; _] →
      str "cons"; args l
  | Pp(Mult), xl →
      infixl "*" xl
  | Pp(Expt _), [x] →
      term x; Sym.pp fmt f
  | Bv(Const(b)), [] →
      str ("0b" ^ Bitv.to_string b)
  | Bv(Conc _), l →
      infixl " ++ " l
  | Bv(Sub(_, i, j)), [x] →
      term x; Format.fprintf fmt "[%d:%d]" i j
  | Coproduct(InL), [App(Product(Tuple), [x; xl])] →
      Pretty.infix pp "::" pp fmt (x, xl)
  | Coproduct(InR), [App(Product(Tuple), [])] →
      str "[]"
  | Arrays(Update), [x; y; z] →
      term x; str "["; term y; str " := "; term z; str "]"
  | Arrays(Select), [x; y] →
      term x; str "["; term y; str "]"
  | _ →
      app f l)
let to_string =
  Pretty.to_string pp
```

258. Pretty-printing of equalities/disequalities/constraints.

```
let pp_equal fmt (x, y) =
  Pretty.infix pp "=" pp fmt (x, y)
let pp_diseq fmt (x, y) =
  Pretty.infix pp "<>" pp fmt (x, y)
let pp_in fmt (x, c) =
  Pretty.infix pp "in" Cnstrnt.pp fmt (x, c)
```

259. Sets and maps of terms.

```
type trm = t (* avoid type-check error below *)
module Set = Set.Make(
  struct
    type t = trm
    let compare = cmp
  end)
module Map = Map.Make(
  struct
    type t = trm
    let compare = cmp
  end)
```

260. Set of variables.

```
let rec vars_of a =
  match a with
    | Var _ →
        Set.singleton a
    | App(_, al) →
        List.fold_left
          (fun acc b →
              Set.union (vars_of b) acc)
          Set.empty
        al
```

6.21 Module trace

Interface for module Trace.mli

261. The purpose of module *Trace* is to provide some rudimentary abstractions for a tracer.

```
type level = string
val reset : unit → unit
val add : level → unit
val remove : level → unit
val get : unit → level list
val call : level → string → α → α Pretty.printer → unit
val exit : level → string → α → α Pretty.printer → unit
val msg : level → string → α → α Pretty.printer → unit
val func : level → string → α Pretty.printer → β Pretty.printer
             → (α → β) → α → β
val proc : level → string → α Pretty.printer
             → (α → unit) → α → unit
```

Module Trace.ml

```
type level = string

module Levels = Set.Make(
  struct
    type t = level
    let compare = Pervasives.compare
  end)

let levels = ref Levels.empty
let _ = Tools.add_at_reset (fun () → levels := Levels.empty)

let reset () = (levels := Levels.empty)

let add l = (levels := Levels.add l !levels)

let remove l = (levels := Levels.remove l !levels)

let get () = Levels.elements !levels

let is_active l =
  Levels.mem l !levels ∨
  Levels.mem "all" !levels

let call level op args pp =
  if is_active level then
    begin
      Format.eprintf "%s:␣%s␣<--␣" level op;
      pp Format.err_formatter args;
      Format.eprintf "@."
    end

let exit level op res pp =
  if is_active level then
    begin
       Format.eprintf "%s:␣%s␣-->␣" level op;
       pp Format.err_formatter res;
       Format.eprintf "@."
    end

let msg level op args pp =
  if is_active level then
    begin
      Format.eprintf "%s:␣%s\t" level op;
      pp Format.err_formatter args;
      Format.eprintf "@."
    end

let rec whitespace level n =
  if is_active level then
```

```
    begin
      match n with
        | 0 → ()
        | n → (Format.eprintf "%d␣" n; whitespace level (n − 1))
    end

let indent = ref 0

let func level =
  fun name pp qq f a →
    try
      whitespace level !indent;
      indent := !indent + 1;
      call level name a pp;
      let b = f a in
        indent := !indent − 1;
        whitespace level !indent;
        exit level name b qq;
        b
    with
      | exc →
          begin
            indent := !indent − 1;
            whitespace level !indent;
            (if is_active level then
               Format.eprintf "Exit:␣%s@." (Printexc.to_string exc));
            raise exc
          end

let proc level =
  let qq fmt () = Format.fprintf fmt "()" in
    (fun name pp → func level name pp qq)
```

6.22 Module fact

Interface for module Fact.mli

```
type t

type rule = string

type justification =
  | Axiom
  | Rule of rule × justification list

val mk_axiom : justification option
val mk_rule : rule → justification option list → justification option
```

```
type equal
type diseq
type cnstrnt

val mk_equal : Term.t → Term.t → justification option → equal
val mk_diseq : Term.t → Term.t → justification option → diseq
val mk_cnstrnt : Term.t → Cnstrnt.t → justification option → cnstrnt

val d_equal : equal → Term.t × Term.t × justification option
val d_diseq : diseq → Term.t × Term.t × justification option
val d_cnstrnt : cnstrnt → Term.t × Cnstrnt.t × justification option

val of_equal : equal → t
val of_diseq : diseq → t
val of_cnstrnt : cnstrnt → t

val pp : t Pretty.printer

val pp_equal : equal Pretty.printer
val pp_diseq : diseq Pretty.printer
val pp_cnstrnt : cnstrnt Pretty.printer

module Equalset : (Set.S with type elt = equal)
```

Module Fact.ml

```
type t =
  | Equal of equal
  | Diseq of diseq
  | Cnstrnt of cnstrnt

and justification =
  | Axiom
  | Rule of string × justification list

and equal = Term.t × Term.t × justification option
and diseq = Term.t × Term.t × justification option
and cnstrnt = Term.t × Cnstrnt.t × justification option

and rule = string

let mk_axiom =
  Some(Axiom)

let mk_rule str jl =
  try
    let jl' = List.map
                (function
                   | Some(j) → j
                   | None → raise Not_found)
```

```
                    jl
      in
         Some(Rule(str, jl'))
   with
         Not_found -> None

let mk_equal x y j =
   let (x, y) = Term.orient (x, y) in
      Trace.msg "fact" "Equal" (x, y) Term.pp_equal;
      (x, y, j)

let mk_diseq x y j =
   let (x, y) = Term.orient (x, y) in
      Trace.msg "fact" "Diseq" (x, y) Term.pp_diseq;
      (x, y, j)

let mk_cnstrnt x c j =
   Trace.msg "fact" "Cnstrnt" (x, c) Term.pp_in;
   (x, c, j)

let d_equal e = e
let d_diseq d = d
let d_cnstrnt c = c

let of_equal e = Equal(e)
let of_diseq d = Diseq(d)
let of_cnstrnt c = Cnstrnt(c)

let rec pp fmt = function
   | Equal(x, y, _) -> Pretty.infix Term.pp "=" Term.pp fmt (x, y)
   | Diseq(x, y, _) -> Pretty.infix Term.pp "<>" Term.pp fmt (x, y)
   | Cnstrnt(x, i, _) -> Pretty.infix Term.pp "in" Cnstrnt.pp fmt (x, i)

and pp_equal fmt e =
   let (x, y, _) = d_equal e in
   Pretty.infix Term.pp "=" Term.pp fmt (x, y)

and pp_diseq fmt d =
   let (x, y, _) = d_diseq d in
   Pretty.infix Term.pp "<>" Term.pp fmt (x, y)

and pp_cnstrnt fmt c =
   let (x, i, _) = d_cnstrnt c in
   Pretty.infix Term.pp "in" Cnstrnt.pp fmt (x, i)

module Equalset = Set.Make(
   struct
      type t = equal
      let compare e1 e2 =
         let (x1, y1, _) = d_equal e1 in
```

```
    let (x2, y2, _) = d_equal e2 in
      if Term.eq x1 x2 ∧ Term.eq y1 y2 then
        0
      else
        Pervasives.compare e1 e2
end)
```

6.23 Module arith

Interface for module Arith.mli

262. Module *Arith*: Constructors, recognizers, and accessors for arithmetic terms. Solvers for rational and integer arithmetic. Processing arithmetic equalities.

263. *is_interp a* holds iff *a* is a inear arithmetic terms; that is, *a* is either a numeral (constructed with *num*, see below), a linear multiplication (*multq*), or an addition (*add*, *add2*).

```
val is_interp : Term.t → bool
```

264. *fold f a e* applies *f* at uninterpreted positions of *a* and accumulates the results starting with *e*.

```
val fold : (Term.t → α → α) → Term.t → α → α
```

265. Some normalization functions.

```
val poly_of : Term.t → Mpa.Q.t × Term.t list
val of_poly : Mpa.Q.t → Term.t list → Term.t

val mono_of : Term.t → Mpa.Q.t × Term.t
val of_mono : Mpa.Q.t → Term.t → Term.t

val monomials : Term.t → Term.t list
```

266. Constructors for building up arithmetic terms. Arithmetic terms are always in polynomial normal form. That is, they either represent a rational *num q*, a power product *x* of the form $x1 \times ... * xn$, for $n \geq 1$, where the *xi* are uninterpreted, a monomial $q \times x$, where the coefficient *q* is a non-zero rational, and *x* a power product, or a (flattened) sum of monomials in ascending order, with respect to the *Term.cmp* comparison, from left-to-right.

num q builds a rational numeral with value *q*, *zero* just abbreviates *num* `'0'`, and *one* abbreviates *num* `'1'`. *add2* (a, b) sums up two terms, *add l* sums the list *l* of terms, and *incr a* increments its argument term. *sub* (a, b) is subtraction, while *mult2* and *mult* construct terms for multiplication, and *div* divides two terms. In addition, these constructors implement a number of simplifications such as *div x* (*mult x x*) = *div* (*num* `'1'`) *x*.

```
val mk_num : Mpa.Q.t → Term.t
val mk_zero : Term.t
```

val mk_one : $Term.t$
val mk_two : $Term.t$

val mk_add : $Term.t \rightarrow Term.t \rightarrow Term.t$
val mk_addl : $Term.t\ list \rightarrow Term.t$
val mk_incr : $Term.t \rightarrow Term.t$
val mk_sub : $Term.t \rightarrow Term.t \rightarrow Term.t$
val mk_neg : $Term.t \rightarrow Term.t$
val mk_multq : $Mpa.Q.t \rightarrow Term.t \rightarrow Term.t$

267. Recognizers.

val is_num : $Term.t \rightarrow bool$

val is_zero : $Term.t \rightarrow bool$
val is_one : $Term.t \rightarrow bool$
val is_q : $Mpa.Q.t \rightarrow Term.t \rightarrow bool$

268. Destructors.

val d_num : $Term.t \rightarrow Q.t\ option$
val d_add : $Term.t \rightarrow Term.t\ list\ option$
val d_multq : $Term.t \rightarrow (Q.t \times Term.t)\ option$

269. Given an arithmetic operation $Num(q)$, $Multq(q)$, Add, $Mult$, or Div as declared in *term.mli*, *sigma op l* builds a normalized application by applying the corresponding constructor to the list l of terms. Hereby, it is assumed that l is empty for $Num(q)$, the singleton list for $Multq(q)$, a list of length two for Div, and lists of length greater or equal to two for both Add and $Mult$.

val $sigma$: $Sym.arith \rightarrow Term.t\ list \rightarrow Term.t$

270. Given a substitution s and a term a, *norm s a* replaces uninterpreted occurrences x in a with term y, if there is a binding $x \mid - > \ y$ in s. The resulting term is in polynomial normal form. Thus, *norm* can be thought of as the the composition of applying substitution s to a followed by sigmatizing each arithmetic subterm by the function *sigma* above.

val map : $(Term.t \rightarrow Term.t) \rightarrow Term.t \rightarrow Term.t$

271. *solve* x (a, b) solves the equation $a = b$ over the rationals. If this equation is inconsistent, then it raises *Exc.Inconsistent*. In case the equation holds trivially it returns the empty solution *None*. Otherwise, it returns the solution $x = t$ as $Some(x, t)$, where x is one of the power products of either a or b, and x does not occur in t. In addition, t is in normalized form. There are usually several choices for the power product x to solve for, and *qsolve* chooses to solve for the largest power product according to the term ordering «<.

val $solve\_for$: $(Term.t \rightarrow bool) \rightarrow Fact.equal \rightarrow Fact.equal\ option$

val $solve$: $Fact.equal \rightarrow Fact.equal\ option$

272. Abstract interpretation in the domain of constraints. Given a context f, which associates uninterpreted subterms of a with constraints, *cnstrnt f a* recurses over the interpreted structure of a and accumulates constraints by calling f at uninterpreted positions and abstractly interpreting the interpreted arithmetic operators in the domain of constraints. May raise the exception *Not_found*, when some uninterpreted subterm of a is not in the domain of f.

```
val tau : (Term.t → Cnstrnt.t) → Sym.arith → Term.t list → Cnstrnt.t
```

Module Arith.ml

273. Theory-specific recognizers

```
let is_interp a =
  match a with
    | App(Arith _, _) → true
    | _ → false
```

274. Fold functional

```
let rec fold f a e =
  match a with
    | App(Arith(op), l) →
        (match op, l with
           | Num _, [] → e
           | Add, l → List.fold_right (fold f) l e
           | Multq _, [x] → fold f x e
           | _ → assert false)
    | _ →
        f a e
```

275. Destructors.

```
let d_num = function
  | App(Arith(Num(q)), []) → Some(q)
  | _ → None

let d_multq = function
  | App(Arith(Multq(q)), [x]) → Some(q, x)
  | _ → None

let d_add = function
  | App(Arith(Add), xl) → Some(xl)
  | _ → None

let monomials = function
  | App(Arith(Add), xl) → xl
  | x → [x]
```

276. Recognizers.

```
let is_num = function
  | App(Arith(Num _), []) → true
  | _ → false

let is_zero = function
  | App(Arith(Num(q)), []) → Q.is_zero q
  | _ → false

let is_one = function
  | App(Arith(Num(q)), []) → Q.is_one q
  | _ → false

let is_q q = function
  | App(Arith(Num(p)), []) → Q.equal q p
  | _ → false
```

277. Constants.

```
let mk_num q = App(Arith(Num(q)), [])

let mk_zero = mk_num(Q.zero)
let mk_one = mk_num(Q.one)
let mk_two = mk_num(Q.of_int 2)
```

278. Some normalization functions.

```
let poly_of a =
  match a with
    | App(Arith(op), l) →
        (match op, l with
           | Num(q), [] → (q, [])
           | Multq _, _ → (Q.zero, [a])
           | Add, ((x :: xl') as xl) →
               (match d_num x with
                  | Some(q) → (q, xl')
                  | None → (Q.zero, xl))
           | _ → assert false)
    | _ →
        (Q.zero, [a])

let of_poly =
  let addsym = Arith(Add) in
  fun q l →
    let m = if Q.is_zero q then l else mk_num q :: l in
      match m with
        | [] → mk_zero
        | [x] → x
        | _ → Term.mk_app addsym m
```

```
let mono_of = function
  | App(Arith(Multq(q)), [x]) → (q, x)
  | a → (Q.one, a)

let of_mono q x =
  if Q.is_zero q then
    mk_zero
  else if Q.is_one q then
    x
  else
    match d_num x with
      | Some(p) → mk_num (Q.mult q p)
      | None → App(Arith(Multq(q)), [x])
```

279. Constructors.

```
let rec mk_multq q a =
  let rec multq q = function
    | [] → []
    | m :: ml →
        let (p, x) = mono_of m in
          (of_mono (Q.mult q p) x) :: (multq q ml)
  in
    if Q.is_zero q then
      mk_zero
    else if Q.is_one q then
      a
    else
      let (p, ml) = poly_of a in
        of_poly (Q.mult q p) (multq q ml)

and mk_add a b =
  Trace.call "linarith" "Add" (a, b) (Pretty.infix Term.pp "␣+␣" Term.pp);
  let rec map2 f l1 l2 = (* Add two polynomials *)
    match l1, l2 with
      | [], _ → l2
      | _, [] → l1
      | m1 :: l1', m2 :: l2' →
          let (q1, x1) = mono_of m1 in
          let (q2, x2) = mono_of m2 in
          let cmp = Term.cmp x1 x2 in
          if cmp = 0 then
            let q = f q1 q2 in
            if Q.is_zero q then
              map2 f l1' l2'
            else
```

```
          (of_mono q x1) :: (map2 f l1' l2')
        else if cmp < 0 then
          m2 :: map2 f l1 l2'
        else (* cmp > 0 *)
          m1 :: map2 f l1' l2
  in
  let (q, l) = poly_of a in
  let (p, m) = poly_of b in
  let c = of_poly (Q.add q p) (map2 Q.add l m) in
    Trace.exit "linarith" "Add" c Term.pp;
    c

and mk_addl l =
  match l with
    | [] → mk_zero
    | [x] → x
    | x :: xl → mk_add x (mk_addl xl)

and mk_incr a =
  let (q, l) = poly_of a in
  of_poly (Q.add q Q.one) l

and mk_neg a =
  mk_multq (Q.minus (Q.one)) a

and mk_sub a b =
  mk_add a (mk_neg b)
```

280. Apply term transformer f at uninterpreted positions.

```
let rec map f =
  Trace.func "linarith" "Map" Term.pp Term.pp
    (fun a →
      match a with
        | App(Arith(op), l) →
            (match op, l with
              | Num _, [] →
                  a
              | Multq(q), [x] →
                  let x' = map f x in
                    if x ≡ x' then a else
                      mk_multq q x'
              | Add, [x; y] →
                  let x' = map f x and y' = map f y in
                    if x ≡ x' ∧ y ≡ y' then a else
                      mk_add x' y'
              | Add, xl →
                  let xl' = Term.mapl (map f) xl in
```

```
                  if xl ≡ xl' then a else
                     mk_addl xl'
               | _ →
                  assert false)
      | _ →
         let b = f a in
            Trace.msg "map" "Apply" (a, b) (Pretty.infix Term.pp "␣|->␣" Term.pp);
            b)
```

281. Interface for sigmatizing arithmetic terms.

```
let rec sigma op l =
   match op, l with
      | Num(q), [] → mk_num q
      | Add, [x; y] → mk_add x y
      | Add, _ :: _ :: _ → mk_addl l
      | Multq(q), [x] → mk_multq q x
      | _ → assert false
```

282. Abstract interpretation in the domain of constraints.

```
let rec cnstrnt ctxt = function
   | (App(Arith(op), l) as a) →
         let c =
            match op, l with
               | Sym.Num(q), [] →
                     Cnstrnt.mk_singleton q
               | Sym.Multq(q), [x] →
                     Cnstrnt.multq q (cnstrnt ctxt x)
               | Sym.Add, l →
                     Cnstrnt.addl (List.map (cnstrnt ctxt) l)
               | _ →
                     assert false
         in
            (try
                  Cnstrnt.inter (ctxt a) c
               with
                     Not_found → c)
   | a →
         ctxt a
```

283. Solving of an equality $a = b$ in the rationals and the integers. Solve for maximal monomial which satisfies predicate *pred*.

```
let rec solve_for pred e =
   let (a, b, j) = Fact.d_equal e in
   let (q, l) = poly_of (mk_sub a b) in
```

```
  if l = [] then
    if Q.is_zero q then None else raise(Exc.Inconsistent)
  else
    try
      let ((p, x), ml) = destructure pred l in
      assert(¬(Q.is_zero p)); (*s case q + p × x + ml = 0 *)
      let b = mk_multq (Q.minus (Q.inv p)) (of_poly q ml) in
      if Term.eq x b then
        None
      else
        let (x, b) = orient pred x b in
        Some(Fact.mk_equal x b j)
    with
        Not_found →
          raise Exc.Unsolved

and orient pred x b =
  if is_interp b then
    (x, b)
  else if is_var b then
    Term.orient (x, b)
  else if x <<< b ∧ pred b then
    (b, x)
  else
    (x, b)
```

284. Destructuring a polynomial into the monomial which satisfies predicate f and the remaining polynomial.

```
and destructure pred l =
    let rec loop acc l =
      match l with
        | m :: ml →
            let (p, x) = mono_of m in
            if pred x then
              ((p, x), List.rev acc @ ml)
            else
              loop (m :: acc) ml
        | [] →
            raise Not_found
    in
    loop [] l
```

285. Solver.

```
let rec solve e =
  try
```

```
    solve_for not_is_slack_var e
  with
      Exc.Unsolved →
        solve_for is_var e

and not_is_slack_var a =
  match a with
    | Var(x) when ¬(Var.is_slack x) → true
    | _ → false
```

286. Constraints.

```
let tau c op al =
  try
    (match op, al with
       | Num(q), [] →
           Cnstrnt.mk_singleton q
       | Multq(q), [x] →
           Cnstrnt.multq q (c x)
       | Add, [x; y] →
           Cnstrnt.add (c x) (c y)
       | Add, _ →
           Cnstrnt.addl (List.map c al)
       | _ →
           Cnstrnt.mk_real)
  with
      Not_found → Cnstrnt.mk_real
```

6.24 Module tuple

Interface for module Tuple.mli

287. Module *Tuple*: canonizer and solver for the theory of tuples.

The signature of this theory consists of the nary function symbol *Product* for constructing tuples and of the family of unary function symbols $Proj(i, n)$, for integers $0 \leq i < n$, for projecting the i-th component (starting with 0 and addressing in increasing order from left to right).

The theory of tuples is given as the initial algebra generated by the axioms

288. *is_interp* a holds iff a is a projection of the form $Proj(i, n)(x)$ or a tuple term $Product(xl)$. Terms for which *is_interp* is false are considered to be uninterpreted.

```
val is_interp : Term.t → bool
```

289. If the argument list l is of length 1, then this term is returned. Otherwise, *tuple l* constructs the corresponding tuple term.

val $mk\_tuple\ :\ Term.t\ list \rightarrow\ Term.t$

290. $proj\ i\ n\ a$ is the constructor for the family of i-th projections from n-tuples, where i is any integer value between 0 and $n-1$. This constructor simplifies $proj\ i\ n\ (tuple \backslash list\{a\_0; ...; a\_n-1\})$ to a_i.

val $mk\_proj\ :\ int \rightarrow\ int \rightarrow\ Term.t\ \rightarrow\ Term.t$

291. $sigma\ op\ l$ applies the function symbol op from the tuple theory to the list l of terms. For the function symbol $Proj(i, n)$ and the list a, it simply applies the constructor $proj\ i\ n\ a$, and for $Tuple$ and it applies $tuple\ l$. All other inputs result in a run-time error.

val $sigma\ :\ Sym.product\ \rightarrow\ Term.t\ list \rightarrow\ Term.t$

292. $fold\ f\ a\ e$ applies f at uninterpreted positions of a and accumulates the results starting with e.

val $fold\ :\ (Term.t\ \rightarrow\ \alpha\ \rightarrow\ \alpha)\ \rightarrow\ Term.t\ \rightarrow\ \alpha\ \rightarrow\ \alpha$

293. $map\ f\ a$ applies f to all top-level uninterpreted subterms of a, and rebuilds the interpreted parts in order.

val $map:\ (Term.t\ \rightarrow\ Term.t)\ \rightarrow\ Term.t\ \rightarrow\ Term.t$

294. $solve\ (a, b)$ returns a solved form for the equation $a = b$. If this equation is inconsistent, then the exception $Exc.Inconsistent$ is raised. Otherwise, the solved form $(x1, e1), ..., (xn, en)$ is returned, where xi are uninterpreted in the tuple theory and the ei are canonized. The ei may also contain fresh variables.

val $solve\ :\ Fact.equal\ \rightarrow\ Fact.equal\ list$

Module Tuple.ml

295. let $is\_interp\ =$ function

  | $App(Product\ \_,\ \_)\ \rightarrow$ true
  | $\_\ \rightarrow$ false

296. Destructors.

let $d\_tuple\ =$ function
  | $App(Product(Tuple),\ xl)\ \rightarrow\ Some(xl)$
  | $\_\ \rightarrow\ None$

let $d\_proj\ =$ function
  | $App(Product(Proj(i,\ n)),\ [x])\ \rightarrow\ Some(i,\ n,\ x)$
  | $\_\ \rightarrow\ None$

297. Fold iterator

```
let rec fold f a e =
  match a with
    | App(Product(Tuple), xl) →
        List.fold_right (fold f) xl e
    | App(Product(Proj _), [x]) →
        fold f x e
    | _ →
        f a e
```

298. Constructors for tuples and projections.

```
let mk_tuple =
  let product = Product(Tuple) in
    function
      | [x] → x
      | ([x; y] as xl) →
          (match x, y with
             | App(Product(Proj(0,2)), [z1]),
               App(Product(Proj(1,2)), [z2]) when Term.eq z1 z2 →
                 z1
             | _ →
                 Term.mk_app product xl)
      | xl →
          Term.mk_app product xl

let mk_proj i n a =
  match a with
    | App(Product(Tuple), xl) →
        List.nth xl i
    | _ →
        Term.mk_app (Product(Proj(i, n))) [a]
```

299. Apply term transformer f at uninterpreted positions.

```
let rec map f a =
  match a with
    | App(Product(Tuple), xl) →
        let xl' = Term.mapl (map f) xl in
          if xl ≡ xl' then a else
            mk_tuple xl'
    | App(Product(Proj(i, n)), [x]) →
        let x' = map f x in
          if x ≡ x' then a else
            mk_proj i n x'
    | _ →
        f a
```

300. Sigmatizing.

```
let sigma op l =
  match op, l with
    | Tuple, _ →
        mk_tuple l
    | Proj(i, n), [x] →
        mk_proj i n x
    | _ →
        assert false
```

301. Fresh variables.

```
let mk_fresh =
  let name = Name.of_string "t" in
    fun () → Var(Var.mk_fresh name None)
```

302. Solving tuples.

```
let rec solve e =
  let (a, b, _) = Fact.d_equal e in
    solvel [(a, b)] []

and solvel el sl =
  match el with
    | [] → sl
    | (a, b) :: el1 →
        solve1 (a, b) el1 sl

and solve1 (a, b) el sl =
  if Term.eq a b then
    solvel el sl
  else if Term.is_var b then
    solvevar (b, a) el sl
  else match a with
    | App(Product(Proj(i, n)), [x]) →
        let e' = proj_solve i n x b in
        solvel (e' :: el) sl
    | App(Product(Tuple), xl) →
        solvel (tuple_solve xl b el) sl
    | _ →
        solvevar (a, b) el sl

and solvevar (x, b) el sl =
  if is_var b then
    solvel el (add (Term.orient (x, b)) sl)
  else if Term.occurs x b then
    raise Exc.Inconsistent
  else
```

```
    solvel el (add (x, b) sl)
```

303. $(a0, ..., a\{n-1\}) = b$ iff $a0 = proj\{0, n\}(b)$ and ... and $a\{n-1\} = proj\{n-1, n\}(b)$

```
and tuple_solve al b acc =
  let n = List.length al in
  let rec loop i al acc =
    match al with
      | [] -> acc
      | a :: al' ->
          let b' = mk_proj i n b in
          let acc' = (a, b') :: acc in
            loop (i + 1) al' acc'
  in
  loop 0 al acc
```

304. $solve\ (proj\ i\ n\ s,\ t) = (s, \backslash list\{c0, ..., t, ...cn-1\})$ where ci are fresh, s at i-th position.

```
and proj_solve i n s t =
  let rec args j acc =
    if j = -1 then acc
    else
      let a =
        if i = j then
          t
        else
            mk_fresh()
      in (* fresh var equals mk_proj j n s *)
        args (j - 1) (a :: acc)
  in
    (s, mk_tuple (args (n - 1) []))
and add (a, b) sl =
  if Term.eq a b then
    sl
  else
    let e = Fact.mk_equal a b None in (* does not swap terms *)
      e :: (substl a b sl) (* since a and b are oriented *)
and substl a b =
  List.map
    (fun e ->
       let (x, y, _) = Fact.d_equal e in
         Fact.mk_equal x (subst1 y a b) None)
and subst1 a x b = (* substitute x by b in a. *)
  map (fun y -> if Term.eq x y then b else y) a
```

6.25 Module coproduct

Interface for module Coproduct.mli

305. Module *Coproduct*: canonizer and solver for the theory of tuples.

```
val mk_inl : Term.t → Term.t

val mk_inr : Term.t → Term.t

val mk_outl : Term.t → Term.t
val mk_outr : Term.t → Term.t

val mk_inj : int → Term.t → Term.t
val mk_out : int → Term.t → Term.t

val sigma : Sym.coproduct → Term.t list → Term.t

val fold : (Term.t → α → α) → Term.t → α → α

val map : (Term.t → Term.t) → Term.t → Term.t

val solve : Fact.equal → Fact.equal list
```

Module Coproduct.ml

306.
```
let is_interp = function
  | App(Coproduct _, _) → true
  | _ → false
```

307. Fold iterator

```
let rec fold f a e =
  match a with
    | App(Coproduct(_), [x]) → fold f x e
    | _ → f a e
```

308. Constructors for tuples and projections.

```
let mk_inl =
  let sym = Coproduct(InL) in
    function
      | App(Coproduct(OutL), [x]) → x
      | x → mk_app sym [x]

let mk_inr =
  let sym = Coproduct(InR) in
    function
      | App(Coproduct(OutR), [x]) → x
```

```
        | x → mk_app sym [x]
let mk_outr =
  let sym = Coproduct(OutR) in
    function
      | App(Coproduct(InR), [x]) → x
      | x → mk_app sym [x]
let mk_outl =
  let sym = Coproduct(OutL) in
    function
      | App(Coproduct(InL), [x]) → x
      | x → mk_app sym [x]
let rec mk_inj i x =
  if i ≤ 0 then
    mk_inl x
  else if i = 1 then
    mk_inr x
  else
    mk_inr (mk_inj (i − 1) x)
let rec mk_out i x =
  if i ≤ 0 then
    mk_outl x
  else if i = 1 then
    mk_outr x
  else
    mk_outr (mk_out (i − 1) x)
```

309. Sigmatizing.

```
let sigma op l =
  match op, l with
    | InL, [x] → mk_inl x
    | InR, [x] → mk_inr x
    | OutL, [x] → mk_outl x
    | OutR, [x] → mk_outr x
    | _ → assert false
```

310. Apply term transformer f at uninterpreted positions.

```
let rec map f a =
  match a with
    | App(Coproduct(op), [x]) →
        let x' = map f x in
          if x ≡ x' then a else sigma op [x']
    | _ →
```

```
      f a
```

311. Solving tuples.

```
let rec solve e =
  let (a, b, _) = Fact.d_equal e in
    solvel [(a, b)] []

and solvel el sl =
  match el with
    | [] → sl
    | (a, b) :: el1 →
        solve1 (a, b) el1 sl

and solve1 (a, b) el sl =
  if Term.eq a b then
    solvel el sl
  else if Term.is_var b then
    solvevar (b, a) el sl
  else match a with
    | App(Coproduct(op), [x]) → (*s solve inY(x) = b is x = outY(b). *)
        let rhs' = match op with (*s solve outY(x) = b is x = inY(b). *)
          | InL → mk_outl b
          | InR → mk_outr b
          | OutL → mk_inl b
          | OutR → mk_inr b
        in
          solvel ((x, rhs') :: el) sl
    | _ →
        solvevar (a, b) el sl

and solvevar (x, b) el sl =
  if is_var b then
    solvel el (add (Term.orient (x, b)) sl)
  else if Term.occurs x b then
    raise Exc.Inconsistent
  else
    solvel el (add (x, b) sl)

and add (a, b) sl =
  if Term.eq a b then sl else
    let e = Fact.mk_equal a b None in (* does not swap terms *)
      e :: (substl a b sl) (* since a and b are oriented *)

and substl a b =
  List.map
    (fun e →
       let (x, y, _) = Fact.d_equal e in
```

$Fact.mk\_equal\ x\ (subst1\ y\ a\ b)\ None)$

and $subst1\ a\ x\ b\ =\ (*$ substitute x by b in a. $*)$
  map (fun $y\ \rightarrow$ if $Term.eq\ x\ y$ then b else $y)\ a$

6.26 Module bitvector

Interface for module Bitvector.mli

312. Module Bv: Constructors, recognizers, and accessors for bitvector terms, and a solver for equations over bitvectors. Bitvectors of width n, where n is a natural number, are arrays of bits with positions 0 through $n - 1$ in increasing order from left-to-right.

313. $is\_interp\ a$ holds iff the top-level function symbol of a is interpreted in the theory of bitvectors (see also module Sym.

val $is\_interp\ :\ Term.t\ \rightarrow\ bool$

314. Computes the width of a bitvector term.

val $width\ :\ Term.t\ \rightarrow\ int\ option$

315. $iter\ f\ a$ applies f for all toplevel subterms of a which are not interpreted in the theory of bitvectors.

val $iter\ :\ (Term.t\ \rightarrow\ unit)\ \rightarrow\ Term.t\ \rightarrow\ unit$

316. $fold\ f\ a\ e$ applies f at uninterpreted positions of a and accumulates the results starting with e.

val $fold\ :\ (Term.t\ \rightarrow\ \alpha\ \rightarrow\ \alpha)\ \rightarrow\ Term.t\ \rightarrow\ \alpha\ \rightarrow\ \alpha$

317. $mk\_const\ c$ is the constructor for building constant bitvectors, in which all bits are known to be either 0 or 1. $mk\_zero\ n$ is just defined to be the constant zero bitvector of length n, $mk\_one\ n$ is the constant one bitvector of length n, and mk_eps is the constant bitvector of length 0.

val $mk\_eps\ :\ Term.t$
val $mk\_one\ :\ int \rightarrow\ Term.t$
val $mk\_zero\ :\ int \rightarrow\ Term.t$
val $mk\_const\ :\ Bitv.t\ \rightarrow\ Term.t$

318. $is\_zero\ a$ holds iff all bits in a are 0.

val $is\_zero\ :\ Term.t\ \rightarrow\ bool$

319. $is\_one\ a$ holds iff all bits in a are 1.

val is_one : $Term.t \to bool$

320. $mk\_conc\ n\ m\ b1\ b2$ concatenates a bitvector $b1$ of width n and a bitvector $b2$ of width m. mk_conc terms are built up in a right-associative way, argument bitvectors of length 0 are ignored, constant argument bitvectors are combined into the corresponding concatenated constant bitvector, and concatentations of extractions such as $mk\_sub\ x\ i\ j$ and $mk\_sub\ x\ (j+1)\ k$ are merged to $mk\_sub\ x\ i\ k$.

val mk_conc : $int \to int \to Term.t \to Term.t \to Term.t$

321. $mk\_sub\ n\ i\ j\ x$ returns the representation of the bitvector for the contiguous extraction of the $j-i+1$ bits i through j in the bitvector x of length n. It is assumed that $0 \leq i, j < n$. If $j < i$ then the empty bitvector mk_eps is returned, and if $i = j$ then the result is a bitvector of width 1. Simplifications include extraction of bitvector terms, mk_sub distributes over concatenation and bitwise operators, and successive extractions are merged.

val mk_sub : $int \to int \to int \to Term.t \to Term.t$

322. $mk\_bitwise\ n\ a\ b\ c$ builds bitvector BDDs of width n. Bitvector BDDs are a canonical form for bitwise conditionals of width n, and they are similar to a binary decision diagram (BDD) in that $mk\_bitwise\ n\ a\ b\ c$ reduces to c when $is\_zero\ a$ holds, it reduces to b when $is\_one\ a$ holds, and to b if b is syntactically equal to c. Furthermore the conditional structure is reduced in that there are no identical substructures, and the conditional parts are ordered from top to bottom (this ordering is fixed but unspecified). Since bitwise operations distribute over concatenation and extraction, all terms in a bitvector BDD are either bitvector constants, uninterpreted terms, or a (single) extraction from an uninterpreted term.

val $mk\_bitwise$: $int \to Term.t \to Term.t \to Term.t \to Term.t$

323. Derived bitwise operators for bitwise conjunction (mk_bwconj), disjunction (mk_bwconj), negation (mk_bwconj), implication (mk_bwconj), and equivalence (mk_bwconj).

val mk_bwconj : $int \to Term.t \to Term.t \to Term.t$
val mk_bwdisj : $int \to Term.t \to Term.t \to Term.t$
val mk_bwneg : $int \to Term.t \to Term.t$
val mk_bwimp : $int \to Term.t \to Term.t \to Term.t$
val mk_bwiff : $int \to Term.t \to Term.t \to Term.t$

324. Given a bitvector symbol f (see module Sym) and a list l of arguments, $sigma\ f\ l$ returns a concatenation normal form (CNF) of the application $'f(l)'$. A CNF is either a simple bitvector or a right-associative concatenation of simple bitvectors, and a simple bitvector is either an uninterpreted term, a bitvector constant, an extraction, or a bitvector BDD (see above). All the simplifications for mk_sub, mk_conc, $mk\_bitwise$ are applied.

val $sigma$: $Sym.bv \to Term.t\ list \to Term.t$

325. $map\ f\ a$ applies f to all top-level uninterpreted subterms of a, and rebuilds the interpreted parts in order. It can be thought of as replacing every toplevel uninterpreted a

with $'f(a)'$ if Not_found is not raised by applying a, and with a otherwise, followed by a sigmatization of all interpreted parts using mk_sigma.

val map : $(Term.t \rightarrow Term.t) \rightarrow Term.t \rightarrow Term.t$

326. $solve\ b$ either fails, in which case b is unsatisfiable in the given bitvector theory or it returns a list of equations $\backslash list\{(x1, e1); ...(xn, en)\}$ such that xi is a non bitvector term, all the xi are pairwise disjoint, none of the xi occurs in any of the terms ej, and, viewed as a conjunction of equivalences, the result is equivalent (in the theory of bitvectors) with b. The terms ei may contain fresh bitvector constants.

val $solve$: $Fact.equal \rightarrow Fact.equal\ list$

Module Bitvector.ml

327.
```
let is_bvsym = function
  | Bv _ -> true
  | _ -> false

let d_bvsym = function
  | Bv(op) -> Some(op)
  | _ -> None

let is_interp = function
  | App(Bv _, _) -> true
  | _ -> false

let d_interp = function
  | App(Bv(op), l) -> Some(op, l)
  | _ -> None
```

Constant bitvectors

```
let mk_const c =
  Term.mk_const(Bv(Const(c)))

let mk_eps =
  mk_const(Bitv.from_string "")

let is_eps a =
  match d_interp a with
    | Some(Const b, []) -> Bitv.length b = 0
    | _ -> false

let mk_zero n =
  assert(n > 0);
  mk_const(Bitv.create n false)

let is_zero a =
```

```
  match d_interp a with
    | Some(Const b, []) → Bitv.all_zeros b
    | _ → false

let mk_one n =
  mk_const(Bitv.create n true)

let is_one a =
  match d_interp a with
    | Some(Const b, []) → Bitv.all_ones b
    | _ → false
```

328. Creating fresh bitvector variables for solver. The index variable are always reset to the current value when solver is called.

```
let mk_fresh =
  let name = Name.of_string "bv" in
  fun n →
    assert (n ≥ 0);
    if n = 0 then
      mk_eps
    else
        Var(Var.mk_fresh name None)
```

329. Bitvector symbols

```
let width a =
  if Term.is_var a then None else
    Sym.width (Term.sym_of a)

let iter f a =
  match d_interp a with
    | Some(Sym.Const(_), []) → ()
    | Some(Sym.Sub(_, _, _), [x]) → iter f x
    | Some(Sym.Conc(n, m), [x; y]) → iter f x; iter f y
    | Some(Sym.Bitwise(n), [x; y; z]) → iter f x; iter f y; iter f z
    | _ → f a
```

330. Fold functional.

```
let rec fold f a e =
  match d_interp a with
    | Some(Sym.Const(_), []) → e
    | Some(Sym.Sub(_, _, _), [x]) → fold f x e
    | Some(Sym.Conc(_, _), [x; y]) → fold f x (fold f y e)
    | Some(Sym.Bitwise(_), [x; y; z]) → fold f x (fold f y (fold f z e))
    | _ → f a e

let is_bitwise a =
```

```
  match d_interp a with
    | Some(Bitwise _, [_; _; _]) → true
    | _ → false

let d_bitwise a =
  match d_interp a with
    | Some(Bitwise(n), [x; y; z]) →
        Some(n, x, y, z)
    | _ →
        None

let d_conc a =
  match d_interp a with
    | Some(Conc(n, m), [x; y]) → Some(n, m, x, y)
    | _ → None

let d_const a =
  match d_interp a with
    | Some(Const(c), []) → Some(c)
    | _ → None

let d_sub a =
  match d_interp a with
    | Some(Sub(n, i, j), [x]) → Some(n, i, j, x)
    | _ → None
```

331. Building up Bitvector BDDs

```
let is_bvbdd a =
  is_zero a ∨ is_one a ∨ is_bitwise a

let cofactors x a =
  match d_interp a with
    | Some(Bitwise _, [y; pos; neg])
        when Term.eq x y →
          (pos, neg)
    | _ →
        (a, a)

let topvar x s2 s3 =
  let max x y = if (x <<< y) then y else x in
  match d_bitwise s2, d_bitwise s3 with
    | Some(_, y, _, _), Some(_, z, _, _) → max x (max y z)
    | Some(_, y, _, _), None → max x y
    | None, Some(_, z, _, _) → max x z
    | None, None → x

module H3 = Hashtbl.Make(
  struct
```

```
    type t = Term.t × Term.t × Term.t
    let equal (a1, a2, a3) (b1, b2, b3) =
      Term.eq a1 b1 ∧ Term.eq a2 b2 ∧ Term.eq a3 b3
    let hash = Hashtbl.hash
  end)

let ht = H3.create 17

let rec build n s3 =
  try
    H3.find ht s3
  with Not_found →
    let b = build_fun n s3 in
    H3.add ht s3 b; b

and build_fun n (s1, s2, s3) =
  if Term.eq s2 s3 then s2
  else if is_one s2 ∧ is_zero s3 then s1
  else if is_one s1 then s2
  else if is_zero s1 then s3
  else match d_bitwise s1 with
    | Some(_, y, _, _) →
        let x = topvar y s2 s3 in
        let (pos1, neg1) = cofactors x s1 in
        let (pos2, neg2) = cofactors x s2 in
        let (pos3, neg3) = cofactors x s3 in
        let pos = build n (pos1, pos2, pos3) in
        let neg = build n (neg1, neg2, neg3) in
        if Term.eq pos neg then pos else Term.mk_app (Bv(Bitwise(n))) [x; pos; neg]
    | None →
        Term.mk_app (Bv(Bitwise(n))) [s1; s2; s3]
```

332. Term constructors.

```
let rec mk_sub n i j a =
  assert (0 ≤ i ∧ j < n ∧ n ≥ 0);
  if i = 0 ∧ j = n − 1 then
    a
  else if j < i then
    mk_eps
  else
    match d_interp a with
      | Some(Const(b), []) →
          mk_const(Bitv.sub b i (j − i + 1))
      | Some(Sub(m, k, l), [x]) →
          mk_sub m (k + i) (k + j) x
      | Some(Conc(n, m), [x; y]) →
```

```
        if j < n then
          mk_sub n i j x
        else if n ≤ i then
          mk_sub m (i − n) (j − n) y
        else
          (assert(i < n ∧ n ≤ j);
          let t = mk_sub n i (n − 1) x in
          let u = mk_sub m 0 (j − n) y in
          mk_conc (n − i) (j − n + 1) t u)
    | Some(Bitwise(n), [x; y; z]) →
        mk_bitwise (j − i + 1) (mk_sub n i j x) (mk_sub n i j y) (mk_sub n i j z)
    | None →
        Term.mk_app (Bv(Sub(n, i, j))) [a]
    | Some _ →
        failwith "Bv.mk_sub: ill-formed expression"

and mk_conc n m a b =
  assert (0 ≤ n ∧ 0 ≤ m);
  match n = 0, m = 0 with
    | true, true → mk_eps
    | true, false → b
    | false, true → a
    | false, false →
        (match merge n m a b with
          | None → Term.mk_app (Bv(Conc(n, m))) [a; b]
          | Some(c) → c)

and merge n m a b =
  match d_interp a, d_interp b with
    | _, Some(Conc(m1, m2), [b1; b2]) →
        Some(mk_conc (n + m1) m2 (mk_conc n m1 a b1) b2)
    | Some(Const(c), []), Some(Const(d), []) →
        Some(mk_const (Bitv.append c d))
    | Some(Sub(n, i, j), [x]), Some(Sub(m, j', k), [y])
        when j' = j + 1 ∧ Term.eq x y →
          assert(n = m);
          Some(mk_sub n i k x)
    | Some(Bitwise(n1), [x1; y1; z1]), Some(Bitwise(n2), [x2; y2; z2]) →
        (match merge n1 n2 x1 x2 with
          | None → None
          | Some(x) →
              (match merge n1 n2 y1 y2 with
                | None → None
                | Some(y) →
                    (match merge n1 n2 z1 z2 with
```

```
                              | None -> None
                              | Some(z) -> Some(mk_bitwise (n1 + n2) x y z))))
    | _ ->
        None

and mk_conc3 n m k a b c =
  mk_conc n (m + k) a (mk_conc m k b c)

and mk_bitwise n a b c =
  assert (n ≥ 0);
  if n = 0 then
    mk_eps
  else
    match d_const a, d_const b, d_const c with
      | Some(c1), Some(c2), Some(c3) ->
          mk_const (Bitv.bw_or (Bitv.bw_and c1 c2) (Bitv.bw_and (Bitv.bw_not c1) c3))
      | _ ->
          (match d_conc a, d_conc b, d_conc c with
             | Some(n1, n2, a1, a2), _, _ ->
                 assert(n = n1 + n2);
                 let b1, b2 = cut n n1 b in
                 let c1, c2 = cut n n1 c in
                 mk_conc n1 n2 (mk_bitwise n1 a1 b1 c1) (mk_bitwise n2 a2 b2 c2)
             | _, Some(n1, n2, b1, b2), _ ->
                 assert(n = n1 + n2);
                 let a1, a2 = cut n n1 a in
                 let c1, c2 = cut n n1 c in
                 mk_conc n1 n2 (mk_bitwise n1 a1 b1 c1) (mk_bitwise n2 a2 b2 c2)
             | _, _, Some(n1, n2, c1, c2) ->
                 assert(n = n1 + n2);
                 let a1, a2 = cut n n1 a in
                 let b1, b2 = cut n n1 b in
                 mk_conc n1 n2 (mk_bitwise n1 a1 b1 c1) (mk_bitwise n2 a2 b2 c2)
             | _ ->
                 drop (build n (lift n a, lift n b, lift n c)))

and lift n a =
  if is_bvbdd a then
    a
  else
    Term.mk_app (Bv(Bitwise(n))) [a; mk_one n; mk_zero n]

and drop a =
 match d_bitwise a with
   | Some(_, b1, b2, b3) when is_one b2 ∧ is_zero b3 -> b1
   | _ -> a
```

```
and cut n i a =
  (mk_sub n 0 (i - 1) a, mk_sub n i (n - 1) a)
```

333. Derived bitwise constructors.

```
let mk_bwconj n a b = mk_bitwise n a b (mk_zero n)
let mk_bwdisj n a b = mk_bitwise n a (mk_one n) b
let mk_bwneg n a = mk_bitwise n a (mk_zero n) (mk_one n)
let mk_bwimp n a1 a2 = mk_bitwise n a1 a2 (mk_one n)
let mk_bwiff n a1 a2 = mk_bitwise n a1 a2 (mk_bwneg n a2)
```

334. Mapping over bitvector terms.

```
let map f =
  let rec loop a =
    match d_interp a with
      | Some(Sym.Const(_), []) →
          a
      | Some(Sym.Sub(n, i, j), [x]) →
          mk_sub n i j (loop x)
      | Some(Sym.Conc(n, m), [x; y]) →
          mk_conc n m (loop x) (loop y)
      | Some(Sym.Bitwise(n), [x; y; z]) →
          mk_bitwise n (loop x) (loop y) (loop z)
      | None →
          f a
      | Some _ →
          assert false
  in
  loop
```

335. Does term a occur interpreted in b.

```
let rec occurs a b =
  let rec loop x =
    (Term.eq x a)
    ∨ (match d_interp x with
          | Some(op, l) →
              (match op, l with
                 | Sym.Const(_), [] → false
                 | Sym.Sub _, [x] → loop x
                 | Sym.Conc(n, m), [x; y] → (loop x) ∨ (loop y)
                 | Sym.Bitwise(n), [x; y; z] → (loop x) ∨ (loop y) ∨ (loop z)
                 | _ → failwith "Bv.map: ill-formed expression")
          | None → false)
  in
```

```
  loop b
```

336. Sigmatizing an expression.

```
let sigma op l =
  match op, l with
    | Sym.Const(c), [] → mk_const c
    | Sym.Sub(n, i, j), [x] → mk_sub n i j x
    | Sym.Conc(n, m), [x; y] → mk_conc n m x y
    | Sym.Bitwise(n), [x; y; z] → mk_bitwise n x y z
    | _ → failwith "Bv.sigma: ill-formed expression"
```

n-ary concatenation of some concatenation normal form

```
let rec mk_iterate n b = function
  | 0 → mk_eps
  | 1 → b
  | k → mk_conc n (n × (k − 1)) b (mk_iterate n b (k − 1))
```

Flattening out concatenations. The result is a list of equivalent equalities not containing any concatenations.

```
let decompose e =
  let rec loop acc = function
    | [] → acc
    | (a, b) :: el when Term.eq a b →
        loop acc el
    | (a, b) :: el →
        let (acc', el') =
          match d_conc a, d_conc b with
            | Some(n1, m1, x1, y1), Some(n2, m2, x2, y2) →
                if n1 = n2 then
                  (acc, ((x1, x2) :: (y1, y2) :: el))
                else if n1 < n2 then
                  let (x21, x22) = cut n2 n1 x2 in
                  let e1 = (x1, x21) in
                  let e2 = (y1, mk_conc (n2 − n1) m2 x22 y2) in
                  (acc, (e1 :: (e2 :: el)))
                else (* n1 > n2 *)
                  let (x11, x12) = cut n1 n2 x1 in
                  let e1 = (x11, x2) in
                  let e2 = (mk_conc (n1 − n2) m1 x12 y1, y2) in
                  (acc, (e1 :: e2 :: el))
            | Some(n, m, x, y), None →
                let e1 = (mk_sub (n + m) 0 (n − 1) b, x) in
                let e2 = (mk_sub (n + m) n (n + m − 1) b, y) in
                (acc, (e1 :: e2 :: el))
            | None, Some(n, m, x, y) →
```

```
            let e1 = (mk_sub (n + m) 0 (n − 1) a, x) in
            let e2 = (mk_sub (n + m) n (n + m − 1) a, y) in
            (acc, (e1 :: e2 :: el))
        | None, None →
            (((a, b) :: acc), el)
      in
      loop acc′ el′
  in
  loop [] [e]
```

337. Solving is based on the equation $ite(x, p, n) = (p \vee n)$ and $exists\ delta.\ x = (p$ and $(n\ => \ delta))$

```
and solve_bitwise n (a, b) =
  assert(n ≥ 0);
  let s = mk_bwiff n a b in
  let rec triangular_solve s e =
    match d_bitwise s with
      | Some (_, x, pos, neg) →
          if is_one pos ∧ is_zero neg then (* poslit *)
            (x, pos) :: e
          else if is_zero pos ∧ is_one neg then (* neglit *)
            (x, pos) :: e
          else
            let t′ = mk_bwconj n pos (mk_bwimp n neg (mk_fresh n)) in
            let e′ = (x, t′) :: e in
            let s′ = mk_bwdisj n pos neg in
            if is_zero s′ then
              raise Exc.Inconsistent
            else if is_one s′ then
              e′
            else if is_bitwise s′ then
              triangular_solve s′ e′
            else
              (s′, mk_one n) :: e′
      | None →
          (s, mk_one n) :: e
  in
  if is_zero s then
    raise Exc.Inconsistent
  else if is_one s then
    []
  else
    triangular_solve s []
```

338. Adding a solved pair $a\ |-> \ b$ to the list of solved forms sl, and propagating this new binding to the unsolved equalities *el* and the rhs of *sl*. It also makes sure that fresh variables *a* are never added to *sl* but only propagated.

```
let rec add a b (el, sl) =
  assert(¬(is_interp a));
  if Term.eq a b then
    (el, sl)
  else if inconsistent a b then
    raise Exc.Inconsistent
  else
    match is_fresh_bv_var a, is_fresh_bv_var b with
      | false, false →
          (inste el a b, (a, b) :: insts sl a b)
      | true, true →
          (inste el a b, insts sl a b)
      | true, false →
          if is_interp b then
            (inste el a b, insts sl a b)
          else
            (inste el b a, insts sl b a)
      | false, true →
          (inste el b a, insts sl b a)

and is_fresh_bv_var _ = false

and inste el a b =
  List.map (fun (x, y) → (apply1 x a b, apply1 y a b)) el

and insts sl a b =
  List.map (fun (x, y) → (x, apply1 y a b)) sl

and inconsistent a b =
  match width a, width b with
    | Some(n), Some(m) → n ≠ m
    | _ → false

and apply1 a x b = (* substitute x by b in a. *)
  map (fun y → if Term.eq x y then b else y) a
```

339. Toplevel solver.

```
let rec solve e =
  let (a, b, _) = Fact.d_equal e in
  let sl = solvel ([(a, b)], []) in
    List.map (fun (x, b) → Fact.mk_equal x b None) sl

and solvel (el, sl) =
  match el with
    | [] → sl
```

```
    | (a, b) :: el when Term.eq a b →
        solvel (el, sl)
    | (a, b) :: el →
        (match d_interp a, d_interp b with
            | None, Some _ when ¬(occurs a b) → (* Check if solved. *)
                solvel (add a b (el, sl))
            | Some _, None when ¬(occurs b a) →
                solvel (add b a (el, sl))
            | Some(Conc _, _), _ (* Decomposition of conc. *)
            | _, Some(Conc _, _) →
                solvel (decompose (a, b) @ el, sl)
            | Some(Bitwise(n), _), _ → (* Solve bitwise ops. *)
                solvel (solve_bitwise n (a, b) @ el, sl)
            | (None | Some(Const _, _)), Some(Bitwise(m), _) →
                solvel (solve_bitwise m (a, b) @ el, sl)
            | Some(Const(c), []), Some(Const(d), []) →
                if Pervasives.compare c d = 0 then
                  solvel (el, sl)
                else
                  raise Exc.Inconsistent
            | Some(Sub(n, i, j), [x]), Some(Sub(m, k, l), [y]) when Term.eq x y →
                assert(n = m);
                let (x, b) = solve_sub_sub x n i j k l in
                solvel (add x b (el, sl))
            | Some(Sub(n, i, j), [x]), _ →
                assert(n − j − 1 ≥ 0);
                assert(i ≥ 0);
                let b' = mk_conc3 i (j − i + 1) (n − j − 1)
                            (mk_fresh i) b (mk_fresh (n − j − 1)) in
                solvel (add x b' (el, sl))
            | _, Some(Sub(n, i, j), [x]) →
                assert(n − j − 1 ≥ 0);
                assert(i ≥ 0);
                let b' = mk_conc3 i (j − i + 1) (n − j − 1) (mk_fresh i)
                            b (mk_fresh (n − j − 1)) in
                solvel (add x b' (el, sl))
            | _ →
                let a, b = Term.orient(a, b) in
                solvel (add a b (el, sl)))

and solve_sub_sub x n i j k l =
  assert (n ≥ 0 ∧ i < k ∧ j − i = l − k);
  let lhs =
    mk_sub n i l x
  in
```

```
let rhs =
  if (l - i + 1) mod (k - i) = 0 then
    let a = mk_fresh (k - i) in
    mk_iterate (k - i) a ((l - i + 1)/(k - i))
  else
    let h = (l - i + 1) mod (k - i) in
    let h' = k - i - h in
    let a = mk_fresh h in
    let b = mk_fresh h' in
    let nc = (l - i - h + 1)/(k - i) in
    let c = mk_iterate (h + h') (mk_conc h' h b a) nc in
    mk_conc h (nc × (h' + h)) a c
in
(lhs, rhs)
```

6.27 Module boolean

Interface for module Boolean.mli

340. Module *Boolean*: Manipulating Boolean constants *mk_true* and *mk_false*.

```
val mk_true : unit → Term.t
val mk_false : unit → Term.t

val is_true : Term.t → bool
val is_false : Term.t → bool

val mk_conj : Term.t → Term.t → Term.t
val mk_disj : Term.t → Term.t → Term.t
val mk_xor : Term.t → Term.t → Term.t
val mk_neg : Term.t → Term.t
```

Module Boolean.ml

341. Boolean constants.

```
let mk_true () = Bitvector.mk_one 1
let mk_false () = Bitvector.mk_zero 1

let is_true a = (Term.eq a (mk_true()))
let is_false a = (Term.eq a (mk_false()))
```

342. Boolean connectives.

```
let mk_conj a b =
  Bitvector.mk_bitwise 1 a b (mk_false())
```

let $mk\_disj\ a\ b\ =$
  $Bitvector.mk\_bitwise\ 1\ a\ (mk\_true())\ b$

let $mk\_xor\ a\ b\ =$
  $Bitvector.mk\_bitwise\ 1\ a$
    $(Bitvector.mk\_bitwise\ 1\ b\ (mk\_false())\ (mk\_true()))$
    a

let $mk\_neg\ a\ =$
  $Bitvector.mk\_bitwise\ 1\ a\ (mk\_false())\ (mk\_true())$

6.28 Module pp

Interface for module Pp.mli

343. Module *Arr*: Nonlinear arithmetic.

val $mk\_one\ :\ Term.t$ (∗ Different form $Arith.mk\_one!$ ∗)

val $is\_one\ :\ Term.t\ \rightarrow\ bool$

val $mk\_mult\ :\ Term.t\ \rightarrow\ Term.t\ \rightarrow\ Term.t$

val $mk\_multl\ :\ Term.t\ list \rightarrow\ Term.t$

val $mk\_expt\ :\ int \rightarrow\ Term.t\ \rightarrow\ Term.t$

val $sigma\ :\ Sym.pprod\ \rightarrow\ Term.t\ list \rightarrow\ Term.t$

val $map : (Term.t\ \rightarrow\ Term.t)\ \rightarrow\ Term.t\ \rightarrow\ Term.t$

344. Abstract interpretation in the domain of constraints. Given a context f, which associates uninterpreted subterms of a with constraints, *cnstrnt f a* recurses over the interpreted structure of a and accumulates constraints by calling f at uninterpreted positions and abstractly interpreting the interpreted arithmetic operators in the domain of constraints.

val $tau\ :\ (Term.t\ \rightarrow\ Cnstrnt.t)\ \rightarrow\ Sym.pprod\ \rightarrow\ Term.t\ list \rightarrow\ Cnstrnt.t$

345. *gcd pp qq* computes the greatest common divisor of the power products pp and qq. It returns a triple of power products $(p,\ q,\ g)$ such that g divides both pp and qq, it is the largest such g, and $mk\_mult\ p\ pp$ and $mk\_mult\ q\ qq$ are equal to g.

val $gcd\ :\ Term.t\ \rightarrow\ Term.t\ \rightarrow\ Term.t\ \times\ Term.t\ \times\ Term.t$

346. Least common muliple *lcm pp qq*

val $lcm\ :\ Term.t\ \rightarrow\ Term.t\ \rightarrow\ Term.t$

347. *split pp* splits a power product pp into a pair (nn, dd) of a numerator nn and a denumerator dd, such that pp equals $mk\_div\ nn\ dd$.

val $split\ :\ Term.t\ \rightarrow\ Term.t\ \times\ Term.t$

val $numerator\ :\ Term.t\ \rightarrow\ Term.t$
val $denumerator\ :\ Term.t\ \rightarrow\ Term.t$

Module Pp.ml

348. Symbols.

```
let mult = Sym.Pp(Sym.Mult)
let expt n = Sym.Pp(Sym.Expt(n))
```

349. Constructors.

```
let mk_one =
  mk_app mult []

let is_one = function
  | App(Pp(Mult), []) → true
  | _ → false

let rec mk_expt n a =
  Trace.msg "pp" "Expt" (a, n) (Pretty.pair Term.pp Pretty.number);
  if n = 0 then (* a^0 = 1 *)
    mk_one
  else if n = 1 then (* a^1 = a *)
    a
  else
    match a with
      | App(Pp(Expt(m)), [x]) → (* x^m^n = x^(m × n) *)
          mk_expt (m × n) x
      | App(Pp(Mult), []) → (* 1^n = 1 *)
          mk_one
      | App(Pp(Mult), [x]) →
          mk_app (expt n) [x]
      | App(Pp(Mult), xl) → (* (x1 × ... × xk)^n = x1^n × ... × ...xk^n *)
          mk_multl (mapl (mk_expt n) xl)
      | _ →
          mk_app (expt n) [a]

and mk_multl al =
  List.fold_left mk_mult mk_one al

and mk_mult a b =
  Trace.msg "pp" "Mult" (a, b) (Pretty.pair Term.pp Term.pp);
  match a with
    | App(Pp(Expt(n)), [x]) →
        mk_mult_with_expt x n b
    | App(Pp(Mult), []) →
        b
    | App(Pp(Mult), xl) →
        mk_mult_with_pp xl b
    | _ →
```

```
        mk_mult_with_expt a 1 b
and mk_mult_with_expt x n b =
  match b with
    | App(Pp(Expt(m)), [y]) when Term.eq x y → (* x^n × x × m = x^(n + m) *)
        mk_expt (n + m) x
    | App(Pp(Mult), []) → (* x^n × 1 = x^n *)
        mk_expt n x
    | App(Pp(Mult), yl) → (* x^n × (y1 × ... × yk) = (y1 × ... x^n ... × yk) *)
        insert x n yl
    | _ →
        insert x n [b]
and mk_mult_with_pp xl b =
  match b with
    | App(Pp(Expt(m)), [y]) → (* (x1 × ... × xk) × y^m = (x1 × ...y^m × ... × xk)
*)
        insert y m xl
    | App(Pp(Mult), yl) →
        merge yl xl
    | _ →
        insert b 1 xl
and cmp (x, n) (y, m) =
  let res = Term.cmp x y in
    if res = 0 then Pervasives.compare n m else res
and destruct a =
  match a with
    | App(Pp(Expt(n)), [x]) → (x, n)
    | _ → (a, 1)
and insert x n bl =
  merge [mk_expt n x] bl
and insert1 x = insert x 1
and merge al bl =
  Trace.msg "pp" "Merge" (al, bl) (Pretty.pair (Pretty.list Term.pp) (Pretty.list Term.pp));
  let compare a b = cmp (destruct a) (destruct b) in
  let rec loop acc al bl =
    match al, bl with
      | [], [] → acc
      | _, [] → List.sort compare (acc @ al)
      | [], _ → List.sort compare (acc @ bl)
      | a :: al', b :: bl' →
          let (x, n) = destruct a
          and (y, m) = destruct b in
          let cmp = Term.cmp x y in
```

```
          if cmp = 0 then
            if n + m = 0 then
              loop acc al' bl'
            else
              loop (mk_expt (n + m) x :: acc) al' bl'
          else if cmp < 0 then
            loop (a :: acc) al' bl
          else
            loop (b :: acc) al bl'
  in
  match loop [] al bl with
    | [] → mk_one
    | [c] → c
    | cl → mk_app mult cl

let mk_inv a = mk_expt (−1) a
```

350. Sigma normal forms.

```
let sigma op l =
  match op, l with
    | Expt(n), [x] → mk_expt n x
    | Mult, xl → mk_multl xl
    | _ → assert false

let rec map f a =
  match a with
    | App(Pp(Expt(n)), [x]) →
        let x' = map f x in
          if x ≡ x' then a else
            mk_expt n x'
    | App(Pp(Mult), xl) →
        let xl' = mapl (map f) xl in
          if xl' ≡ xl then a else
            mk_multl xl'
    | _ →
        f a
```

351. Constraint.

```
let tau ctxt op l =
  Trace.msg "pp" "tau" l (Pretty.list Term.pp);
  try
    match op, l with
      | Expt(n), [x] →
          Cnstrnt.expt n (ctxt x)
      | Mult, [] →
```

```
        Cnstrnt.mk_one
    | Mult, _ →
        Cnstrnt.multl (List.map ctxt l)
    | _ →
        assert false
  with
      Not_found → Cnstrnt.mk_real
```

352. Normalize a power product to a list.

```
let to_list a =
  match a with
    | App(Pp(Mult), xl) → xl
    | _ → [a]
```

353. Greatest common divisor of two power products. For example, *gcd* 'x^2 × y' 'x × y^2 returns the triple (y, x, x × y). x × y is the gcd of these power products.

```
let gcd a b =
  let rec gcdloop ((pl, ql, gcd) as acc) (al, bl) =
    match al, bl with
      | [], [] →
          acc
      | [], bl →
          (bl @ pl , ql, gcd)
      | al, [] →
          (pl, al @ ql, gcd)
      | a :: al', b :: bl' →
          let (x, n) = destruct a
          and (y, m) = destruct b in
          let res = Term.cmp x y in
            if res = 0 then
              let acc' =
                if n = m then
                  (pl, ql, mk_expt n x :: gcd)
                else if n < m then
                  (mk_expt (m − n) x :: pl, ql, mk_expt n x :: gcd)
                else (* n > m *)
                  (pl, mk_expt (n − m) x :: ql, mk_expt m x :: gcd)
              in
                gcdloop acc' (al', bl')
            else if res > 0 then
              let x_pow_n = mk_expt n x in
                gcdloop (x_pow_n :: pl, ql, x_pow_n :: gcd) (al', bl)
            else (* res < 0 *)
              let y_pow_m = mk_expt m y in
```

```
          gcdloop (y_pow_m :: pl, ql, y_pow_m :: gcd) (al, bl')
  in
  let (pl, ql, gcd) = gcdloop ([], [], []) (to_list a, to_list b) in
    (mk_multl pl, mk_multl ql, mk_multl gcd)

let split a =
  let (numerator, denumerator) =
    List.partition
      (fun m →
         let (_, n) = destruct m in
           n ≥ 0)
      (to_list a)
  in
    (mk_multl numerator, mk_inv (mk_multl denumerator))

let numerator a = fst(split a)
let denumerator a = snd(split a)
```

354. Least common multiple

```
let lcm qq pp =
  let rec loop acc al bl =
    match al, bl with
      | [], [] →
          acc
      | [], bl →
          mk_mult acc (mk_multl bl)
      | al, [] →
          mk_mult acc (mk_multl al)
      | a :: al', b :: bl' →
          let (x, n) = destruct a
          and (y, m) = destruct b in
          let acc' =
            if Term.eq x y then
              mk_mult acc (mk_expt (Pervasives.min n m) x)
            else
              mk_mult a (mk_mult b acc)
          in
            loop acc' al' bl'
  in
    loop mk_one (to_list pp) (to_list qq)
```

6.29 Module arr

Interface for module Arr.mli

355. Module *Arr*: Array expressions.

val mk_select : $Term.t \rightarrow Term.t \rightarrow Term.t$

val mk_update : $Term.t \rightarrow Term.t \rightarrow Term.t \rightarrow Term.t$

val $sigma$: $Sym.arrays \rightarrow Term.t\ list \rightarrow Term.t$

val map : $(Term.t \rightarrow Term.t) \rightarrow Term.t \rightarrow Term.t$

Module Arr.ml

356. Reducing patterns of the form $select(update(a, i, x),\ j)$ according to the equations $i\ =\ j\ =>\ select(update(a, i, x),\ j)\ =\ x\ i\ \neq\ j\ =>\ select(update(a, i, x), j)\ =\ select(a, j)$

```
let mk_select =
  let select = mk_app (Arrays(Select)) in
    fun b j →
      match b with
        | App(Arrays(Update), [a; i; x]) →
            (match Term.is_equal i j with
                | Three.Yes →
                    x
                | Three.No →
                    select [a; j]
                | Three.X →
                    select [b; j])
        | _ →
            select [b; j]

let mk_update =
  let update = mk_app (Arrays(Update)) in
    fun a j y →
      match a with
        | App(Arrays(Update), [b; i; x]) when Term.eq i j →
            update [b; i; y]
        | _ →
            update [a; j; y]

let sigma op l =
  match op, l with
    | Update, [a; i; x] →
```

```
      mk_update a i x
    | Select, [a; j] →
      mk_select a j
    | _ →
      assert false
```

```
let rec map f a =
  match a with
    | App(Arrays(Update), [a; i; x]) →
        let a' = map f a and i' = map f i and x' = map f x in
          if a ≡ a' ∧ i ≡ i' ∧ x ≡ x' then a else
            mk_update a' i' x'
    | App(Arrays(Select), [a; j]) →
        let a' = map f a and j' = map f j in
          if a ≡ a' ∧ j ≡ j' then a else
            mk_select a' j'
    | _ →
        f a
```

6.30 Module bvarith

Interface for module Bvarith.mli

357. Module *Arr*: Array expressions.

val *mk_unsigned* : *Term.t* → *Term.t*

val *sigma* : *Sym.bvarith* → *Term.t list* → *Term.t*

val *map* : (*Term.t* → *Term.t*) → *Term.t* → *Term.t*

val *tau* : (*Term.t* → *Cnstrnt.t*) → *Sym.bvarith* → *Term.t list* → *Cnstrnt.t*

Module Bvarith.ml

```
let rec mk_unsigned =
  let unsigned = mk_app (Bvarith(Unsigned)) in
    function
      | App(Bv(Const(b)), []) →
          Arith.mk_num (Q.of_int (unsigned_interp_of b))
      | App(Bv(Conc(n, m)), [x; y]) →
          let ux = mk_unsigned x
          and uy = mk_unsigned y in
          let two_expt_m = Q.of_z (Z.expt (Z.of_int 2) m) in
            Arith.mk_add (Arith.mk_multq two_expt_m ux) uy
```

```
      | a ->
          unsigned [a]

and unsigned_interp_of b =
  Bitv.fold_right
    (fun x acc ->
       if x then 2 × acc + 1 else 2 × acc)
    b 0

let sigma op l =
  match op, l with
    | Unsigned, [x] -> mk_unsigned x
    | _ -> assert false

let rec map f a =
  match a with
    | App(Bvarith(Unsigned), [x]) ->
        let x' = map f x in
          if x ≡ x' then a else
            mk_unsigned x'
    | _ ->
        f a

let tau _ _ _ = Cnstrnt.mk_nat
```

6.31 Module apply

Interface for module Apply.mli

358. Module *Apply*: Theoruy of function application and abstraction

```
val mk_apply : (Sym.t -> Term.t list -> Term.t)
                 -> Cnstrnt.t option -> Term.t -> Term.t list
                   -> Term.t

val mk_abs : Term.t -> Term.t

val sigma : Sym.apply -> Term.t list -> Term.t

val map : (Term.t -> Term.t) -> Term.t -> Term.t
```

Module Apply.ml

```
let mk_abs a =
  mk_app (Fun(Abs)) [a]

let rec mk_apply sigma r a al =
  match a, al with
```

```
      | App(Fun(Abs), [x]), [y] →
          subst sigma x y 0
      | _ →
          mk_app (Fun(Apply(r))) (a :: al)
and subst sigma a s k =
   match a with
      | Var(x) →
          if Var.is_free x then
             let i = Var.d_free x in
                if k < i then
                   Var(Var.mk_free(i − 1))
                else if i = k then
                   s
                else
                   Var(Var.mk_free i)
          else
             a
      | App(Fun(Abs), [x]) →
          mk_abs (subst sigma x (lift s 0) (k + 1))
      | App(f, xl) →
          sigma f (substl sigma xl s k)
and substl sigma al s k =
   mapl (fun x → subst sigma x s k) al
and lift a k =
   match a with
      | Var(x) →
          if Var.is_free x then
             let i = Var.d_free x in
                if i < k then a else Var(Var.mk_free(i + 1))
          else
             a
      | App(Fun(Abs), [x]) →
          mk_abs (lift x (k + 1))
      | App(f, xl) →
          mk_app f (liftl xl k)
and liftl al k =
   mapl (fun a → lift a k) al
let sigma op al =
   match op, al with
      | Apply(r), x :: xl →
          mk_apply mk_app r x xl (* no simplifications *)
      | Abs, [x] →
```

```
        mk_abs x
    | _ → assert false

let rec map f a =
  match a with
    | App(Fun(Apply(r)), x :: xl) →
        let x' = map f x in
        let xl' = mapl (map f) xl in
          if x ≡ x' ∧ xl ≡ xl' then a else
            mk_apply mk_app r x' xl'
    | App(Fun(Abs), [x]) →
        let x' = map f x in
          if x ≡ x' then a else
            mk_abs x'
    | _ →
        f a
```

6.32 Module app

Interface for module App.mli

359. Module *App*: Operations on uninterpreted terms.

```
val sigma : Sym.t → Term.t list → Term.t
```

```
val lazy_sigma : Term.t → Sym.t → Term.t list → Term.t
```

360. *is_uninterp* a holds iff the function symbol of a is uninterpreted (see module *Sym*).

```
val is_uninterp : Term.t → bool
```

361. For a term a such that *is_uninterp* a holds, *d_uninterp* a returns the uninterpreted function symbol and the argument list of a.

```
val d_uninterp : Term.t → Sym.t × Term.t list
```

362. Maps.

```
val map : (Term.t → Term.t) → Term.t → Term.t
```

Module App.ml

363.

```
let sigma f l =
  Term.mk_app f l
```

```
let lazy_sigma a f l =
```

```
  assert(¬(is_var a) ∧ Sym.eq (sym_of a) f);
  let m = args_of a in
  if try List.for_all2 eq l m with Invalid_argument _ → false then
    a
  else
    sigma f l

let is_uninterp = function
  | App(Sym.Uninterp _, _) → true
  | _ → false

let d_uninterp a =
  assert(is_uninterp a);
  let f, l = Term.destruct a in
  (f, l)
```

364. Theory-specific normalization.

```
let map ctxt a =
  match a with
    | Var _ → ctxt(a)
    | App(f, l) →
        let l' = mapl ctxt l in
          if l ≡ l' then a else
            mk_app f l'
```

6.33 Module th

Interface for module Th.mli

365. Classification of function symbols.

```
type t

val eq : t → t → bool

val u : t (* Theory of uninterpreted function symbols. *)
val la : t (* Linear arithmetic theory. *)
val p : t (* Product theory. *)
val bv : t (* Bitvector theory. *)
val cop : t (* Coproducts. *)
val pprod : t (* Power products. *)
val app : t (* Theory of function abstraction and application. *)
val arr : t (* Array theory. *)
val bvarith : t (* Theory of bitvector interpretation(s). *)

val to_int : t → int
```

val of_int : $int \rightarrow t$

366. Only u is fully uninterpreted.

val $is\_fully\_uninterp$: $t \rightarrow bool$

367. la, p, bv, cop are fully interpreted.

val is_fully_interp : $t \rightarrow bool$

val to_string : $t \rightarrow string$

val pp : $t\ Pretty.printer$

368. Classification of function symbols.

val of_sym : $Sym.t \rightarrow t$

369. Theory-specific map function.

val map : $t \rightarrow (Term.t \rightarrow Term.t) \rightarrow Term.t \rightarrow Term.t$

370. Theory-specific solver

val $solve$: $t \rightarrow Fact.equal \rightarrow Fact.equal\ list$

371. Arrays of index theory.

module $Array$: sig

  type $\alpha\ arr$

  val $create$: $\alpha \rightarrow \alpha\ arr$

  val $copy$: $\alpha\ arr \rightarrow \alpha\ arr$

  val get : $\alpha\ arr \rightarrow t \rightarrow \alpha$

  val set : $\alpha\ arr \rightarrow t \rightarrow \alpha \rightarrow unit$

  val $reset$: $\alpha\ arr \rightarrow \alpha \rightarrow unit$

  val $iter$: $(t \rightarrow \alpha \rightarrow unit) \rightarrow \alpha\ arr \rightarrow unit$

  val for_all : $(\alpha \rightarrow bool) \rightarrow \alpha\ arr \rightarrow bool$

  val for_all2 : $(\alpha \rightarrow \beta \rightarrow bool) \rightarrow \alpha\ arr \rightarrow \beta\ arr \rightarrow bool$

  val to_list : $\alpha\ arr \rightarrow (t \times \alpha)\ list$

  val pp : $\alpha\ Pretty.printer \rightarrow \alpha\ arr\ Pretty.printer$

end

Module Th.ml

372. Classification of function symbols.

```
type t = int

let eq i j = (i = j)

let of_int i = i

let to_int i = i

let names = ["u"; "la"; "p"; "bv"; "cop"; "nl"; "app"; "arr"; "bva"]

let num_of_theories = List.length names

let u = 0

let la = 1
let p = 2
let bv = 3
let cop = 4

let pprod = 5
let app = 6
let arr = 7
let bvarith = 8

let to_string = List.nth names

exception Found of t

let of_string str =
  try
    let i = ref 0 in
      while !i < num_of_theories do
        if to_string !i = str then
          raise(Found(!i));
        i := !i + 1
      done;
      raise(Invalid_argument str)
  with
      Found(i) → i

let pp fmt th =
  Format.fprintf fmt "%s" (to_string th)

let is_fully_uninterp i = (i = u)

let is_fully_interp i =
  (la ≤ i ∧ i ≤ cop)

let of_sym = function
```

```
  | Sym.Uninterp _ → u
  | Sym.Arith _ → la
  | Sym.Product _ → p
  | Sym.Bv _ → bv
  | Sym.Coproduct _ → cop
  | Sym.Arrays _ → arr
  | Sym.Pp _ → pprod
  | Sym.Fun _ → app
  | Sym.Bvarith _ → bvarith

module Array = struct

  type α arr = α array

  let create x = Array.create num_of_theories x

  let copy = Array.copy

  let get = Array.unsafe_get
  let set = Array.unsafe_set

  let reset a x =
    for i = 0 to num_of_theories − 1 do
      set a i x
    done

  let iter f a =
    for i = 0 to num_of_theories − 1 do
      f i (get a i)
    done

  let fold_left = Array.fold_left
  let fold_right = Array.fold_right

  let to_list a = failwith "to do"

  let of_list = Array.of_list

  exception No

  let for_all p a =
    try
      for i = 0 to num_of_theories − 1 do
        if ¬ (p (get a i)) then
          raise No
      done;
      true
    with
        No → false

  let for_all2 p a b =
    try
```

```
      for i = 0 to num_of_theories - 1 do
        if ¬(p (get a i) (get b i)) then
          raise No
      done;
      true
    with
        No → false

  let pp p fmt a =
    Pretty.map pp p fmt (to_list a)

end
```

373. Theory-specific normalization.

```
let maps =
  let a = Array.create (fun _ a → a) in
    List.iter
      (fun (i, m) → Array.set a i m)
      [u, App.map;
       la, Arith.map;
       p, Tuple.map;
       bv, Bitvector.map;
       cop, Coproduct.map;
       pprod, Pp.map;
       app, Apply.map;
       arr, Arr.map;
       bvarith, Bvarith.map];
    a

let map = Array.get maps
```

374. Theory-specific solver

```
let solvers =
  let a = Array.create (fun e → [e]) in
    List.iter
      (fun (i, m) → Array.set a i m)
      [la, (fun e →
                match Arith.solve e with
                  | None → []
                  | Some(e') → [e']);
       p, Tuple.solve;
       bv, Bitvector.solve;
       cop, Coproduct.solve];
    a

let solve = Array.get solvers
```

6.34 Module atom

Interface for module Atom.mli

375. The module *Atom* implements constructors for atomic predicates.

type t =
  | *True*
  | *Equal* of *Fact.equal*
  | *Diseq* of *Fact.diseq*
  | *In* of *Fact.cnstrnt*
  | *False*

376. Constructors.

val *mk_true* : *unit* → *t*
val *mk_false* : *unit* → *t*

val *mk_equal* : *Fact.equal* → *t*

val *mk_diseq* : *Fact.diseq* → *t*

val *mk_in* : *Fact.cnstrnt* → *t*

377. Pretty-printing.

val *pp* : *t Pretty.printer*

378. The less-then constructor *lt* (a, b) builds a constraint corresponding to the fact that the difference $sub(a, b)$ is negative. Similarly, the less-or-equal constructor *le* associates a non-positive constraint to $sub(a, b)$. In addition, $sub(a, b)$ is normalized so that the coefficient of its least power product, with respect to $\ll <$, is one.

val *mk_lt* : *Term.t* → *Term.t* → *t*
val *mk_le* : *Term.t* → *Term.t* → *t*

379. Set of atoms.

module *Set* : (*Set.S* with type *elt* = *t*)

Module Atom.ml

380. type t =

  | *True*
  | *Equal* of *Fact.equal*
  | *Diseq* of *Fact.diseq*
  | *In* of *Fact.cnstrnt*

```
  | False
```

381. Constructors.

```
let mk_true () = True

let mk_false () = False

let mk_equal e =
  let (a, b, _) = Fact.d_equal e in
    if Term.eq a b then
      mk_true()
    else if Term.is_interp_const a ∧ Term.is_interp_const b then
      mk_false()
    else
      Equal(e) (* Larger Term on rhs *)

let rec mk_in c =
  let (a, c, j) = Fact.d_cnstrnt c in
  if Cnstrnt.is_empty c then
    False
  else
    match Cnstrnt.d_singleton c with
    | Some(q) →
        mk_equal (Fact.mk_equal a (Arith.mk_num q) j)
    | None →
        (match a with
            | Term.App(Sym.Arith(Sym.Num(q)), []) →
                if Cnstrnt.mem q c then True else False
            | _ →
                let (a', c') = normalize (a, c) in
                  In(Fact.mk_cnstrnt a' c' j))

and normalize (a, c) =
  match a with
    | Term.App(Arith(Multq(q)), [x]) when ¬(Mpa.Q.is_zero q) →
        (x, Cnstrnt.multq (Mpa.Q.inv q) c)
    | Term.App(Arith(Add), m1 :: m2 :: ml) →
        (match m1, m2 with
            | Term.App(Arith(Sym.Num(q)), []),
              Term.App(Arith(Multq(p)), [x])
                when ¬(Mpa.Q.is_zero p) → (* q + p × x + ml in c iff *)
                let pinv = Mpa.Q.inv p in (* x + 1/p × ml in 1/ p × (c − q) *)
                let c' = Cnstrnt.multq pinv (Cnstrnt.addq (Q.minus q) c) in
                let a' = Arith.mk_add x (Arith.mk_multq pinv (Arith.mk_addl ml)) in
                  (a', c')
            | _ → (a, c))
    | _ → (a, c)
```

```
let rec mk_diseq d =
  let (a, b, j) = Fact.d_diseq d in
    if Term.eq a b then
      mk_false()
    else if Term.is_interp_const a ∧ Term.is_interp_const b then
      mk_true()
    else if Term.eq a (Boolean.mk_true()) then
      mk_equal(Fact.mk_equal b (Boolean.mk_false()) j)
    else if Term.eq a (Boolean.mk_false()) then
      mk_equal(Fact.mk_equal b (Boolean.mk_true()) j)
    else if Term.eq b (Boolean.mk_true()) then
      mk_equal(Fact.mk_equal a (Boolean.mk_false()) j)
    else if Term.eq b (Boolean.mk_false()) then
      mk_equal(Fact.mk_equal a (Boolean.mk_true()) j)
    else
      match Arith.d_num a, Arith.d_num b with
        | Some(q), _ →
            mk_in(Fact.mk_cnstrnt b (Cnstrnt.mk_diseq q) j)
        | _, Some(p) →
            mk_in(Fact.mk_cnstrnt a (Cnstrnt.mk_diseq p) j)
        | None, None →
            Diseq(Fact.mk_diseq a b j)
```

382. Constructing inequalities.

```
let rec mk_lt a b =
  lower (Q.lt, Cnstrnt.mk_lt Dom.Real, Cnstrnt.mk_gt Dom.Real) (a, b)

and mk_le a b =
  lower (Q.le, Cnstrnt.mk_le Dom.Real, Cnstrnt.mk_ge Dom.Real) (a, b)

and lower (f, less, greater) (a, b) =
  let (q, ml) = Arith.poly_of (Arith.mk_sub a b) in
  match ml with
    | [] →
        if f q Q.zero then mk_true() else mk_false()
    | m :: ml → (*s case p × x + ml < q' *)
        let (p, x) = Arith.mono_of m in (*s case q + p × x + ml < 0 *)
        assert(¬(Q.is_zero p));
        let rel = (if Q.gt p Q.zero then less else greater) in
        let c = rel (Q.minus (Q.div q p)) in
        let ml' = List.map (Arith.mk_multq (Q.inv p)) ml in
        let a = Arith.of_poly Q.zero ml' in
          mk_in (Fact.mk_cnstrnt (Arith.mk_add x a) c None)
```

383. Pretty-printing.

```
let pp fmt = function
  | True -> Pretty.string fmt "True"
  | False -> Pretty.string fmt "False"
  | Equal(e) -> Fact.pp_equal fmt e
  | Diseq(d) -> Fact.pp_diseq fmt d
  | In(c) -> Fact.pp_cnstrnt fmt c
```

384. Set of atoms.

```
type atom = t

module Set = Set.Make(
  struct
    type t = atom
    let compare a b =
      if a = b then 0 else Pervasives.compare a b
  end)
```

6.35 Module use

Interface for module Use.mli

385. `type t`

```
val mem : Term.t -> t -> bool

val apply : t -> Term.t -> Term.Set.t

val find : t -> Term.t -> Term.Set.t

val set : Term.t -> Term.Set.t -> t -> t
```

386. *empty* use lists.

```
val empty : t
```

387. *add x a use* adds x to the use of y for each variable in a.

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

388. *remove x a s* deletes x from the use of y for each variable in a.

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

Module Use.ml

389. `type t = Set.t Map.t`

```
let mem = Map.mem

let apply u a = Map.find a u

let find u a =
  try Map.find a u with Not_found → Set.empty

let set = Map.add
```

390. empty use list.

```
let empty = Map.empty
```

391. *add x a use* adds x to the use of y for each toplevel uninterpreted term in a.

```
let add x =
  Term.fold
    (fun y acc →
       try
         let uy = Map.find y acc in
         let uy' = Set.add x uy in
         if uy ≡ uy' then acc else Map.add y uy' acc
       with
           Not_found →
             Map.add y (Set.singleton x) acc)
```

392. *remove x a s* deletes x from the use of y for each toplevel uninterpreted term in a.

```
let remove x =
  Term.fold
    (fun y acc →
       try
         let uy = Map.find y acc in
         let uy' = Set.remove x uy in
         if Set.is_empty uy' then
           Map.remove y acc
         else if uy ≡ uy' then
           acc
         else
           Map.add y uy' acc
       with
           Not_found → acc)
```

393. Pretty-printing.

```
let rec pp fmt u =
  Pretty.map Term.pp (Pretty.set Term.pp) fmt (to_list u)

and to_list u =
  Map.fold (fun x ys acc → (x, Set.elements ys) :: acc) u []
```

6.36 Module d

Interface for module D.mli

394. Module D: Context for handling disequalities.

type t

val pp : $Format.formatter \rightarrow t \rightarrow unit$

val $changed$: $Term.Set.t\ ref$

395. Return disequalities as bindings of the form $x \mid -> \{y1, ..., yn\}$. The interpretation of such a binding is the conjunction $x \neq y1 \wedge ... \wedge x \neq yn$ of all known disequalities for x. The bindings returned by deq are closed in that forall x, y such that $x \mid -> \{..., y, ...\}$ then also $y \mid -> \{...., x, ...\}$

val deq_of : $t \rightarrow Term.Set.t\ Term.Map.t$

396. $deq\ s\ a$ is just returns the binding for a in $deq\_of\ s$.

val deq : $t \rightarrow Term.t \rightarrow Term.Set.t$

397. check if two terms are known to be disequal.

val is_diseq : $t \rightarrow Term.t \rightarrow Term.t \rightarrow bool$

398. The empty disequality context.

val $empty$: t

399. Is state unchanged.

val eq : $t \rightarrow t \rightarrow bool$

400. $merge\ e\ s$ merges a variable equality $'x = y'$

val $merge$: $Fact.equal \rightarrow t \rightarrow t$

401. $add\ (a, b)\ s$ disequality $a \neq b$ to the disequality context s.

val add : $Fact.diseq \rightarrow t \rightarrow t$

402. Return disequalites for x.

val $disequalities$: $t \rightarrow Term.t \rightarrow Fact.diseq\ list$

Module D.ml

403. Known disequalities; $x \mid -> \{y, z\}$ is interpreted as $x \neq y \wedge x \neq z$. The function is closed in that forall x, y such that $x \mid -> \{..., y, ...\}$ then also $y \mid -> \{...., x, ...\}$

```
type t = Term.Set.t Term.Map.t

let changed = ref Term.Set.empty

let empty = Term.Map.empty

let eq s t = (s ≡ t)

let deq_of s = s

let to_list s =
  let eq (x1, y1) (x2, y2) =
    (Term.eq x1 x2 ∧ Term.eq y1 y2) ∨
    (Term.eq x1 y2 ∧ Term.eq y1 x2)
  in
  let mem (x, y) = List.exists (eq (x, y)) in
  Term.Map.fold
    (fun x ys acc →
       Term.Set.fold
         (fun y acc →
            if mem (x, y) acc then acc else (x, y) :: acc)
         ys acc)
    s []

let pp fmt s =
  let l = to_list s in
  if l ≠ [] then
    begin
      Format.fprintf fmt "\nd:";
      Pretty.list Term.pp_diseq fmt l
    end
```

404. All terms known to be disequal to a.

```
let deq s a =
  try Term.Map.find a s with Not_found → Term.Set.empty
```

405. Check if two terms are known to be disequal.

```
let is_diseq s a b =
  Term.Set.mem b (deq s a)
```

406. Adding a disequality over variables

```
let rec add d s =
```

```
let (x, y, _) = Fact.d_diseq d in
let xd = deq s x in
let yd = deq s y in
let xd' = Term.Set.add y xd in
let yd' = Term.Set.add x yd in
match xd ≡ xd', yd ≡ yd' with
  | true, true →
      s
  | true, false →
      Trace.msg "d" "Update" (x, y) Term.pp_diseq;
      changed := Term.Set.add y !changed;
      Term.Map.add y yd' s
  | false, true →
      Trace.msg "d" "Update" (x, y) Term.pp_diseq;
      changed := Term.Set.add x !changed;
      Term.Map.add x xd' s
  | false, false →
      Trace.msg "d" "Update" (x, y) Term.pp_diseq;
      changed := Term.Set.add x (Term.Set.add y !changed);
      Term.Map.add x xd' (Term.Map.add y yd' s)
```

407. Propagating an equality between variables.

```
let merge e s =
  let (a, b, _) = Fact.d_equal e in
  let da = deq s a and db = deq s b in
  if Term.Set.mem a db ∨ Term.Set.mem b da then
    raise Exc.Inconsistent
  else
    let dab = Term.Set.union da db in
    if db ≡ dab then
      Term.Map.remove a s
    else
      begin
        changed := Term.Set.add b !changed;
        Term.Map.add b dab (Term.Map.remove a s)
      end
```

408. Return disequalities for x.

```
let disequalities s x =
  Term.Set.fold
    (fun y acc →
       (Fact.mk_diseq x y None) :: acc)
    (deq s x)
    []
```

6.37 Module v

Interface for module V.mli

409. Module V: Representation of equivalence classes of variables.

type t

410. *partition* s returns a partitioning of the set of variables in the form of a map with a domain consisting of canonical representatives and the corresponding equivalence class in the codomain. It does only list non-singleton equivalence classes.

val $partition\ :\ t\ \rightarrow\ Term.Set.t\ Term.Map.t$

411. *find* s x returns the canonical representative of x with respect ot the partitioning of the variables in s. In addition, *find′* performs dynamic path compression as a side effect.

val $find\ :\ t\ \rightarrow\ Term.t\ \rightarrow\ Term.t$

val $find'\ :\ t\ \rightarrow\ Term.t\ \rightarrow\ t\ \times\ Term.t$

val $justification\ :\ t\ \rightarrow\ Term.t\ \rightarrow\ Term.t\ \times\ Fact.justification\ option$

val $equality\ :\ t\ \rightarrow\ Term.t\ \rightarrow\ Fact.equal$

412. The set of changed x in domain.

val $changed\ :\ Term.Set.t\ ref$

413. Set of removable variables.

val $removable\ :\ Term.Set.t\ ref$

414. Equality test.

val $eq\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$

415. *is_equal* s x y holds if and only if x and y are in the same equivalence class modulo s.

val $is\_equal\ :\ t\ \rightarrow\ Term.t\ \rightarrow\ Term.t\ \rightarrow\ bool$

416. The empty context.

val $empty\ :\ t$

417. Adding a variable equality $x\ =\ y$ to a context s.

val $merge\ :\ Fact.equal\ \rightarrow\ t\ \rightarrow\ t$

418. *restrict* x s removes occurrences of x from s. Should only be called for x in *removable* s.

val *restrict* : $Term.t \rightarrow t \rightarrow t$

419. Pretty-printing.

val *pp* : *t Pretty.printer*

420. Folding over the members of a specific equivalence class.

val *fold* : $t \rightarrow (Term.t \rightarrow \alpha \rightarrow \alpha) \rightarrow Term.t \rightarrow \alpha \rightarrow \alpha$

421. Iterate over the extension of an equivalence class.

val *iter* : $t \rightarrow (Term.t \rightarrow unit) \rightarrow Term.t \rightarrow unit$

422. *exists* $s\ p\ x$ holds if $p\ y$ holds for some y congruent to x modulo s.

val *exists* : $t \rightarrow (Term.t \rightarrow bool) \rightarrow Term.t \rightarrow bool$

423. *for_all* $s\ p\ x$ holds if $p\ y$ holds for all y congruent to x modulo s.

val *for_all* : $t \rightarrow (Term.t \rightarrow bool) \rightarrow Term.t \rightarrow bool$

424. *choose* $s\ p\ x$ chooses a y which is congruent to x modulo s which satisfies p. If there is no such y, the exception *Not_found* is raised.

val *choose* : $t \rightarrow (Term.t \rightarrow \alpha\ option) \rightarrow Term.t \rightarrow \alpha$

Module V.ml

425. type t = {

```
  find : (Term.t × Fact.justification option) Map.t;
  inv : Set.t Map.t
}
let changed = ref Term.Set.empty
let removable = ref Term.Set.empty
let eq s t =
  s.find ≡ t.find
let apply s = function
  | App _ → raise Not_found (* only variables in domain. *)
  | x → fst(Map.find x s.find)
let find s x =
  let rec loop x =
    try
      let y = apply s x in
        if Term.eq x y then y else loop y
    with
```

```
        Not_found -> x
  in
    loop x

let justification s x =
  (find s x, None)

let equality s x =
  let (y, prf) = justification s x in
    Fact.mk_equal x y prf

let inv s x =
  match x with
    | App _ -> raise Not_found
    | _ -> Map.find x s.inv

let union x y prf s =
  Trace.msg "v" "Union" (x, y) Term.pp_equal;
    changed := Set.add x !changed;
    if is_fresh_var x then
      removable := Set.add x !removable;
    let invy =
      Set.add x
        (try Map.find y s.inv with Not_found -> Set.empty)
    in
      {find = Map.add x (y, prf) s.find;
       inv = Map.add y invy s.inv}

let restrict x s =
  try
    let y = apply s x in (* prf |- x = y. *)
      Trace.msg "v" "Restrict" x pp;
      let y = find s y in (* now get canonical y. *)
      let find' =
        let newfind = Map.remove x s.find in (* remove x |-> y. *)
          try
            let invx = inv s x in (* for all z |-> x, set z |-> y. *)
              Set.fold
                (fun z -> Map.add z (y, None))
                invx
                newfind
          with
              Not_found -> newfind
      in
      let inv' =
        let newinv = Map.remove x s.inv in (*s remove the inverse of x. *)
          try
```

```
          let invy = inv s y in (* remove x from the inverse of y. *)
          let invy' = Set.remove x invy in
            if Set.is_empty invy' then
              Map.remove y newinv
            else
              Map.add y invy' newinv
        with
            Not_found -> newinv
    in
      changed := Set.remove x !changed;
      removable := Set.remove x !removable;
      {find = find'; inv = inv'}
  with
      Not_found -> s
```

426. Canonical representative with dynamic path compression.

```
let find' s x =
  let rec loop acc x =
    try
      let y = fst(Map.find x s.find) in
      if Term.eq x y then
        (acc, y)
      else
        loop (x :: acc) y
    with
        Not_found -> (acc, x)
  in
  let (xl, y) = loop [] x in
  let s' = List.fold_right (fun x -> union x y None) xl s in
  (s', y)
```

427. Variable equality modulo s.

```
let is_equal s x y =
  let x' = find s x
  and y' = find s y in
    Term.eq x' y'
```

428. The empty context.

```
let empty = {
  find = Map.empty;
  inv = Map.empty
}
```

```
let is_empty s = (s.find ≡ Map.empty)
```

429. Starting from the canonical representative $x' = find\ s\ x$, the function f is applied to each y in $inv\ s\ x'$ and the results are accumulated.

```
let fold s f x =
  let rec loop y acc =
    let acc' = f y acc in
    try
      Set.fold loop (Map.find y s.inv) acc'
    with
        Not_found → acc'
  in
  let y = find s x in
    loop y
```

430. Adding a binding $a\ |-> \ b$ to a context s.

```
let merge e s =
  let (x, y, prf) = Fact.d_equal e in (* prf |- x = y *)
  let (x', prf1) = justification s x in (* prf1 |- x = x'. *)
  let (y', prf2) = justification s y in (* prf2 |- y = y'. *)
  let (x', y') = Term.orient (x', y') in
    if Term.eq x' y' then
      s
    else
      let prf' = Fact.mk_rule "trans" [prf; prf1; prf2] in
        union x' y' prf' s
```

431. Extension of the equivalence class for x.

```
let ext s x = fold s Set.add x Set.empty
```

432. Iteration.

```
let iter s f x =
  let rec loop y =
    f y;
    try
      Set.iter loop (Map.find y s.inv)
    with
        Not_found → ()
  in
  let y = find s x in
    loop y
```

433. Exists/Forall

```
let exists s p x =
```

```
  let rec loop y =
    p y ∨
    try
      Set.exists loop (Map.find y s.inv)
    with
        Not_found → false
  in
  let y = find s x in
    loop y

let for_all s p x =
  let rec loop y =
    p y ∧
    try
      Set.for_all loop (Map.find y s.inv)
    with
        Not_found → true
  in
  let y = find s x in
    loop y
```

434. Choose an element satisfying some property.

```
exception Found

let choose s p x =
  let result = ref (Obj.magic 1) in
  try
    iter s
      (fun y →
          match p y with
            | Some(z) →
                result := z;
                raise Found
            | None → ())
      x;
    raise Not_found
  with
      Found → !result
```

435. Set of canonical representatives with non-trivial equivalence classes. These are the variables occurring in the codomain of *find* which are not themselves in the domain of *find*.

```
let canrepr s =
  Map.fold
    (fun _ (y, _) acc →
        if Map.mem y s.find then
```

```
      acc
    else
      Set.add y acc)
  s.find
  Set.empty
```

436. Representation of the equivalence classes as a map with the canonical representatives as domain and the corresponding extensions in the codomain. The following is not terribly efficient.

```
let partition s =
  Set.fold
    (fun x →
       Map.add x (ext s x))
    (canrepr s)
    Map.empty
```

437. Pretty-printing.

```
let pp fmt s =
  if ¬(is_empty s) then
    let m = partition s in
    let l = Map.fold (fun x ys acc → (x, Set.elements ys) :: acc) m [] in
    Pretty.string fmt "\nv:";
    Pretty.map Term.pp (Pretty.set Term.pp) fmt l
```

6.38 Module c

Interface for module C.mli

438. type t

val *cnstrnts* : $t \rightarrow$ (*Cnstrnt.t* $\times$ *Fact.justification option*) *Var.Map.t*

val *to_list* : $t \rightarrow$ (*Var.t* $\times$ *Cnstrnt.t*) *list*

val *apply* : $t \rightarrow$ *Term.t* $\rightarrow$ *Cnstrnt.t*

val *to_fact* : $t \rightarrow$ *Term.t* $\rightarrow$ *Fact.cnstrnt*

val *mem* : *Term.t* $\rightarrow t \rightarrow$ *bool*

439. Empty constraint map.

val *empty* : t

val *eq* : $t \rightarrow t \rightarrow$ *bool*

440. Add a new constraint.

val add : $Fact.cnstrnt \rightarrow t \rightarrow t$

441. Changed.

val $changed$: $Set.t$ ref

442. Merge a variable equality $x = y$ in the constraint map by adding x in ij for the canonical variable x, where x in i, y in j are in the constraint map and ij is the intersection of i and j, and by removing the constraint for y. In case, ij is a singleton constraint with element q, an equality $x = q$ is generated. Singleton constraints are always retained in the constraint map in order to keep the invariant that the best available constraint are always associated with canonical variables.

val $merge$: $Fact.equal \rightarrow t \rightarrow t$

443. Propagate disequalities to the constraint part. The following is not complete and should be extended to all finite constraints, but the disequality sets might become rather large then.

val $diseq$: $Fact.diseq \rightarrow t \rightarrow t$

444. Split.

val $split$: $t \rightarrow Atom.Set.t$

445. Pretty-printing.

val pp : t $Pretty.printer$

Module C.ml

446. type $t = (Cnstrnt.t \times Fact.justification\ option)\ Var.Map.t$

let $empty = Var.Map.empty$

let $eq\ s\ t = (s \equiv t)$

let $cnstrnts\ s = s$

let $to\_list\ s =$
  $Var.Map.fold$ (fun $x\ (i,\ \_)\ acc \rightarrow (x,\ i) :: acc)\ s\ [\,]$

let $changed = ref\ Set.empty$

let $apply\ s =$ function
  | $Var(x) \rightarrow fst(Var.Map.find\ x\ s)$
  | $App\ \_ \rightarrow raise\ Not\_found$

let $justification\ s =$ function
  | $Var(x) \rightarrow Var.Map.find\ x\ s$
  | $\_ \rightarrow raise\ Not\_found$

```
let to_fact s x =
  let (i, prf) = justification s x in
    Fact.mk_cnstrnt x i prf

let mem a s =
  match a with
    | Var(x) -> Var.Map.mem x s
    | App _ -> false
```

447. *update x i s* updates the constraint map with the constraint x in i.

```
let update a i prf s =
  match a with
    | Var(x) ->
        Trace.msg "c" "Update" (a, i) Term.pp_in;
        changed := Term.Set.add a !changed;
        Var.Map.add x (i, prf) s
    | _ -> s
```

448. Restrict the map.

```
let restrict a s =
  match a with
    | Var(x) when Var.Map.mem x s ->
        Trace.msg "c" "Restrict" a Term.pp;
        changed := Term.Set.remove a !changed;
        Var.Map.remove x s
    | _ -> s
```

449. Adding a new constraint.

```
let rec add c s =
  let (x, i, prf1) = Fact.d_cnstrnt c in (* prf1 |- x in i. *)
  try
    let (j, prf2) = justification s x in (* prf2 |- x in j. *)
      (match Cnstrnt.cmp i j with
         | Binrel.Disjoint ->
             raise Exc.Inconsistent
         | (Binrel.Same | Binrel.Super) ->
             s
         | Binrel.Sub ->
             update x i prf1 s
         | Binrel.Singleton(q) ->
             let prf = Fact.mk_rule "inter" [prf1; prf2] in
               update x (Cnstrnt.mk_singleton q) prf s
         | Binrel.Overlap(ij) ->
             let prf = Fact.mk_rule "inter" [prf1; prf2] in
```

```
              update x ij prf s)
  with
      Not_found ->
        update x i prf1 s
```

450. Merge a variable equality $x = y$ in the constraint map by adding x in ij for the canonical variable x, where x in i, y in j are in the constraint map and ij is the intersection of i and j, and by removing the constraint for y. In case, ij is a singleton constraint with element q, an equality $x = q$ is generated. Singleton constraints are always retained in the constraint map in order to keep the invariant that the best available constraint are always associated with canonical variables.

```
let merge e s =
  Trace.msg "c1" "Equal" e Fact.pp_equal;
  let (x, y, prf) = Fact.d_equal e in (* prf |- x = y *)
  try
    let (i, prf1) = justification s x in (* prf1 |- x in i *)
    let s' = restrict x s in
    try
      let (j, prf2) = justification s y in (* prf2 |- y in j. *)
      match Cnstrnt.cmp i j with
        | Binrel.Disjoint ->
            raise Exc.Inconsistent
        | (Binrel.Same | Binrel.Super) ->
            s'
        | Binrel.Sub ->
            let prf' = Fact.mk_rule "equal_sub" [prf; prf1; prf2] in
              update y i None s'
        | Binrel.Singleton(q) ->
            let prf' = Fact.mk_rule "equal_overlap" [prf; prf1; prf2] in
              update y (Cnstrnt.mk_singleton q) prf' s'
        | Binrel.Overlap(ij) ->
            let prf' = Fact.mk_rule "equal_overlap" [prf; prf1; prf2] in
              update y ij prf' s'
    with
        Not_found ->
          let prf' = Fact.mk_rule "equal_cnstrnt" [prf; prf1] in
            update y i prf' s'
  with
      Not_found ->
        s
```

451. Propagate disequalities to the constraint part. The following is not complete and should be extended to all finite constraints, but the disequality sets might become rather large then.

```
let diseq d s =
  Trace.msg "c1" "Diseq" d Fact.pp_diseq;
  let (x, y, prf) = Fact.d_diseq d in (* prf |- x ≠ y *)
  try
    let (i, prf1) = justification s x in (* prf1 |- x in i *)
    let (j, prf2) = justification s y in
    match Cnstrnt.d_singleton i, Cnstrnt.d_singleton j with
      | Some(q), Some(p) →
          if Mpa.Q.equal q p then
            raise Exc.Inconsistent
          else
            s
      | Some(q), None →
          let j' = Cnstrnt.inter j (Cnstrnt.mk_diseq q) in
            add (Fact.mk_cnstrnt y j' None) s
      | None, Some(q) →
          let i' = Cnstrnt.inter i (Cnstrnt.mk_diseq q) in
            add (Fact.mk_cnstrnt x i' None) s
      | None, None →
          s
  with
      Not_found → s
```

452. Split.

```
let split s =
  Var.Map.fold
    (fun x (i, prf) acc →
       if Cnstrnt.is_finite i then
         Atom.Set.add (Atom.mk_in (Fact.mk_cnstrnt (Var(x)) i prf)) acc
       else
         acc)
    s Atom.Set.empty
```

453. Pretty-printing.

```
let pp fmt s =
  let l = to_list s in
    if l ≠ [] then
      begin
        Format.fprintf fmt "\nc:";
        Pretty.map Var.pp Cnstrnt.pp fmt l
      end
```

6.39 Module sig

Interface for module Sig.mli

454. Module *Sig*: builtin simplification

```
val mk_mult : Term.t → Term.t → Term.t
val mk_multl : Term.t list → Term.t
val mk_expt : int → Term.t → Term.t
val mk_div : Term.t → Term.t → Term.t
val mk_inv : Term.t → Term.t
```

Module Sig.ml

```
let rec mk_mult a b =
  Trace.msg "nonlin" "Mult" (a, b) (Pretty.pair Term.pp Term.pp);
  if Term.eq a b then (* a × a --> a^2 *)
    mk_expt 2 a
  else if Pp.is_one a then
    b
  else if Pp.is_one b then
    a
  else
    match a, b with
      | App(Arith(op), xl), _ →
          mk_mult_arith op xl b
      | _, App(Arith(op), yl) →
          mk_mult_arith op yl a
      | _ →
          mk_inj (Pp.mk_mult a b)

and mk_inj pp =
  if Pp.is_one pp then
    Arith.mk_one
  else
    pp

and mk_mult_arith op yl b =
  match op, yl with
    | Num(q), [] →
        Arith.mk_multq q b
    | Multq(q), [x] →
        if Pp.is_one x then
```

```
        Arith.mk_multq q b
      else
        Arith.mk_multq q (mk_mult x b)
    | Add, _ →
        mk_mult_add b yl
    | _ →
        assert false

and mk_mult_add a yl =
  Arith.mk_addl (mapl (mk_mult a) yl)

and mk_multl al =
  List.fold_left mk_mult Arith.mk_one al

and mk_expt n a =
  Trace.msg "nonlin" "Expt" (a, n) (Pretty.pair Term.pp Pretty.number);
  if n = 0 then
    Arith.mk_one
  else if n = 1 then
    a
  else match a with
    | App(Arith(op), xl) →
        mk_expt_arith n op xl
    | _ →
        mk_inj (Pp.mk_expt n a)

and mk_expt_arith n op xl =
  match op, xl with
    | Num(q), [] → (* case q^n *)
        if n ≥ 0 then
          Arith.mk_num (Mpa.Q.expt q n)
        else if n = −1 ∧ ¬(Q.is_zero q) then
          Arith.mk_num (Mpa.Q.inv q)
        else
          mk_inj (Pp.mk_expt (−1) (Arith.mk_num q))
    | Multq(q), [x] → (* (q × x)^n = q^n × x^n *)
        if n ≥ 0 then
          Arith.mk_multq (Mpa.Q.expt q n) (mk_expt n x)
        else if n = −1 ∧ ¬(Q.is_zero q) then
          Arith.mk_multq (Mpa.Q.inv q) (mk_inj (Pp.mk_expt (−1) x))
        else
          mk_inj (Pp.mk_expt n (Pp.mk_mult (Arith.mk_num q) x))
    | Add, _ → (* case (x1 + ... + xk)^n *)
        mk_expt_add n xl
    | _ →
        assert false
```

```
and mk_expt_add n xl =
  Trace.msg "nonlin" "Expt_add" (Arith.mk_addl xl, n) (Pretty.pair Term.pp Pretty.number);
  if n = 0 then
    Arith.mk_one
  else if n = 1 then
    Arith.mk_addl xl
  else if n > 1 then
    mk_mult_add (mk_expt_add (n - 1) xl) xl
  else
    mk_inj (Pp.mk_expt n (Arith.mk_addl xl))

let mk_div a b =
  Trace.msg "nonlin" "Div" (a, b) (Pretty.pair Term.pp Term.pp);
  mk_mult a (mk_expt (-1) b)

let mk_inv a =
  Trace.msg "nonlin" "Inv" a Term.pp;
  mk_expt (-1) a
```

6.40 Module partition

Interface for module Partition.mli

455. Module *Partition*: A partition consists of a

- set of variable equalities $x = y$,
- a set of variable disequalities $x \neq y$,
- and a set of variable constraints x in i,

where i is an arithmetic constraint of type *Cnstrnt.t*.

```
type t
```

456. Accessors.

```
val v_of : t → V.t
val d_of : t → D.t
val c_of : t → C.t
```

457. The *empty* partition.

```
val empty : t
```

458. $v\ s\ x$ returns the canonical representative of the equivalence class in the partitioning s containing the variable x.

$\mathsf{val}\ v\ :\ t\ \rightarrow\ Term.t\ \rightarrow\ Term.t$

459. $c\ s\ x$ returns, for a canonical variable x, an associatiatede arithmetic constraint, if there is one. Otherwise, *Not_found* is raised.

$\mathsf{val}\ c\ :\ t\ \rightarrow\ Term.t\ \rightarrow\ Cnstrnt.t$

460. $update\_v\ s\ v$ updates the v part of the partitioning s if it is different from $s.v$. Similarly, *update_d* and *update_c* update the disequality part and the constraint part, respectively.

$\mathsf{val}\ update\_v\ :\ t\ \rightarrow\ V.t\ \rightarrow\ t$
$\mathsf{val}\ update\_d\ :\ t\ \rightarrow\ D.t\ \rightarrow\ t$
$\mathsf{val}\ update\_c\ :\ t\ \rightarrow\ C.t\ \rightarrow\ t$

461. $copy\ p$ does a shallow copying of p.

$\mathsf{val}\ copy\ :\ t\ \rightarrow\ t$

462. $is\_int\ s\ a$ tests if the constraint $cnstrnt\ s\ a$ is included in $Cnstrnt.mk\_int$.

$\mathsf{val}\ is\_int\ :\ t\ \rightarrow\ Term.t\ \rightarrow\ bool$

463. $deq\ s\ x$ returns the set of all variable y disequal to x as stored in the variable disequality part d of the partitioning s. Disequalities as obtained from the constraint part c are not necessarily included.

$\mathsf{val}\ deq\ :\ t\ \rightarrow\ Term.t\ \rightarrow\ Term.Set.t$

464. Pretty-printing of a partitioning.

$\mathsf{val}\ pp\ :\ t\ Pretty.printer$

465. $is\_equal\ s\ x\ y$ for variables x, y returns *Three.Yes* if x and y belong to the same equivalence class modulo s, that is, if $v\ s\ x$ and $v\ s\ y$ are equal. The result is *Three.No* if x is in $deq\ y$, y is in $deq\ x$, or x in i and y in j are constraints in s and i, j are disjoint. Otherwise, *Three.X* is returned.

$\mathsf{val}\ is\_equal\ :\ t\ \rightarrow\ Term.t\ \rightarrow\ Term.t\ \rightarrow\ Three.t$

$merge\ e\ s$ adds a new variable equality e of the form $x\ =\ y$ into the partition s. If x is already equal to y modulo s, then s is unchanged; if x and y are disequal, then the exception *Exc.Inconsistent* is raised; otherwise, the equality $x\ =\ y$ is added to s to obtain s' such that $v\ s'\ x$ is identical to $v\ s'\ y$.

$\mathsf{val}\ merge\ :\ Fact.equal\ \rightarrow\ t\ \rightarrow\ t$

466. $remove\ s$ removes all internal variables which are not canonical.

$\mathsf{val}\ restrict\ :\ Term.Set.t\ \rightarrow\ t\ \rightarrow\ t$

467. $add\ c\ s$ adds a constraint of the form x in i to the constraint part c of the partition s. May raise *Exc.Inconsistent* if the resulting constraint for x is the empty constraint (see $C.add$).

val add : $Fact.cnstrnt \rightarrow t \rightarrow t$

468. $diseq\ d\ s$ adds a disequality of the form $x \neq y$ to s. If $x = y$ is already known in s, that is, if $is\_equal\ s\ x\ y$ yields $Three.Yes$, then an exception $Exc.Inconsistent$ is raised; if $is\_equal\ s\ x\ y$ equals $Three.No$ the result is unchanged; otherwise, $x \neq y$ is added using $D.add$.

val $diseq$: $Fact.diseq \rightarrow t \rightarrow t$

469. Return stored facts.

val $equality$: $t \rightarrow Term.t \rightarrow Fact.equal$
val $disequalities$: $t \rightarrow Term.t \rightarrow Fact.diseq\ list$
val $cnstrnt$: $t \rightarrow Term.t \rightarrow Fact.cnstrnt$

470. $eq\ s\ t$ holds if the respective equality, disequality, and constraint parts are identical, that is, stored in the same memory location.

val eq : $t \rightarrow t \rightarrow bool$

471. Management of changed variables.

module $Changed$: sig

  val $reset$: $unit \rightarrow unit$
  val $save$: $unit \rightarrow Term.Set.t \times Term.Set.t \times Term.Set.t$
  val $restore$: $Term.Set.t \times Term.Set.t \times Term.Set.t \rightarrow unit$
  val $stable$: $unit \rightarrow bool$

end

Module Partition.ml

472. Equalities and disequalities over variables and constraints on variables

type t = {
  mutable v : $V.t$; (* Variable equalities. *)
  mutable d : $D.t$; (* Variables disequalities. *)
  mutable c : $C.t$ (* Constraints. *)
}

let $empty$ = {
  $v = V.empty$;
  $d = D.empty$;
  $c = C.empty$
}

let $copy\ p$ = {$v = p.v$; $d = p.d$; $c = p.c$}

473. Accessors.

```
let v_of s = s.v
let d_of s = s.d
let c_of s = s.c
```

474. Destructive Updates.

```
let update_v p v = (p.v ← v; p)
let update_d p d = (p.d ← d; p)
let update_c p c = (p.c ← c; p)
```

475. Canonical variables module s.

```
let v s = V.find s.v
```

476. Constraint for a variable.

```
let c s = C.apply s.c
```

477. All disequalities of some variable x.

```
let deq s = D.deq s.d
```

478. Pretty-printing.

```
let pp fmt s =
  V.pp fmt s.v;
  D.pp fmt s.d;
  C.pp fmt s.c
```

479. Test if states are unchanged.

```
let eq s t =
  V.eq s.v t.v ∧
  D.eq s.d t.d ∧
  C.eq s.c t.c
```

480. Equality test.

```
let is_equal s x y =
  let x' = v s x in
  let y' = v s y in
  if Term.eq x' y' then
    Three.Yes
  else if D.is_diseq s.d x' y' then
    Three.No
  else
    try
      let i = C.apply s.c x in
      let j = C.apply s.c y in
      if Cnstrnt.is_disjoint i j then
```

```
        Three.No
      else
        Three.X
    with
        Not_found → Three.X
```

481. Test for integerness.

```
let is_int s x =
  try
    let i = C.apply s.c x in
    Cnstrnt.dom_of i = Dom.Int
  with
      Not_found → false
```

Merge a variable equality.

```
let merge e s =
  let (x, y, _) = Fact.d_equal e in
  match is_equal s x y with
    | Three.Yes → s
    | Three.No → raise Exc.Inconsistent
    | Three.X →
        Trace.msg "p" "Merge" e Fact.pp_equal;
        let v' = V.merge e s.v in
        let d' = D.merge e s.d in
        let c' = C.merge e s.c in
          update_v (update_d (update_c s c') d') v'
```

482. Add a constraint.

```
let add c s =
  let c' = C.add c s.c in
    if C.eq s.c c' then s else
      begin
        Trace.msg "p" "Add" c Fact.pp_cnstrnt;
          update_c s c'
      end
```

483. Add a disequality.

```
let diseq d s =
  let (x, y, _) = Fact.d_diseq d in
  match is_equal s x y with
    | Three.Yes →
        raise Exc.Inconsistent
    | Three.No →
        s
```

```
| Three.X →
    let d' = D.add d s.d in
    let c' = C.diseq d s.c in
      Trace.msg "p" "Diseq" d Fact.pp_diseq;
      update_d (update_c s c') d'

let restrict xs s =
  let v' = Set.fold V.restrict xs s.v in
    update_v s v'
```

484. Stored facts.

```
let equality p = V.equality p.v
let disequalities p = D.disequalities p.d
let cnstrnt p = C.to_fact p.c
```

485. Management of changed sets.

```
module Changed = struct

  let reset () =
    V.changed := Set.empty;
    D.changed := Set.empty;
    C.changed := Set.empty

  let save () = (!V.changed, !D.changed, !C.changed)

  let restore (v, d, c) =
    V.changed := v;
    D.changed := d;
    C.changed := c

  let stable () =
    !V.changed = Set.empty ∧
    !D.changed = Set.empty ∧
    !C.changed = Set.empty

end
```

6.41 Module solution

Interface for module Solution.mli

486. Module *Solution*: abstract datatype for representing and manipulating conjunctions of equations $x = a$, where x is a variable and a is a non-variable term. As an invariant, solution sets s are kept in functional form, that is, if $x = a$ and $x = b$ in s, then a is identical with b. In addition, solution sets are injective, that is, $x = a$ and $y = a$ are not in a solution set for $x \neq y$.

type t

487. *to_list* s returns a list of pairs (x, a), where x is a variable and a a term for the equalities $x = a$ in the solution set s.

val $to\_list : t \to (Term.t \times Term.t)\ list$

488. Pretty-printing of solution sets.

val $pp : Th.t \to t\ Pretty.printer$

489. *fold* $f\ s\ e$ applies $f\ x\ a$ to all $x = a$ in the solution set s in an unspecified order and accumulates the result.

val $fold : (Term.t \to Term.t \times Fact.justification\ option \to \alpha \to \alpha) \to t \to \alpha \to \alpha$

490. *apply* $s\ x$ returns b if $x = b$ is in s, and raises *Not_found* otherwise.

val $apply : t \to Term.t \to Term.t$

find $s\ x$ returns b if $x = b$ is in s, and x otherwise. For non-variable argument a, *find* $s\ a$ always returns a.

val $find : t \to Term.t \to Term.t$

491. *justification* $s\ x$ returns (b, j) if $x = b$ is in s with justification j. Raises *Not_found* if there is no such justification.

val $justification : t \to Term.t \to Term.t \times Fact.justification\ option$

val $equality : t \to Term.t \to Fact.equal$

492. *inv* $s\ b$ returns x if $x = b$ is in s; otherwise *Not_found* is raised.

val $inv : t \to Term.t \to Term.t$

493. *mem* $s\ x$ holds iff $x = b$ is in s.

val $mem : t \to Term.t \to bool$

494. *occurs* $s\ x$ holds if either *mem* $s\ x$ or if x is a variable in some b where $y = b$ is in s.

val $occurs : t \to Term.t \to bool$

495. *use* $s\ x$ returns all y such that $y = a$ in s and x is a variable in $Term.vars\ a$.

val $use : t \to Term.t \to Term.Set.t$

496. The *empty* solution set, which does not contain any equality.

val $empty : t$

497. *is_empty* s holds iff s does not contain any equalities.

val $is\_empty : t \to bool$

498. $eq\ s\ t$ tests if the solution sets s and t are identical in the sense that the sets of equalities are stored at the same memory location.

val $eq : t \to t \to bool$

499. $restrict\ s\ x$ removes equalities $x = a$ from s.

val $restrict : Th.t \to Term.t \to t \to t$

500. $union\ (x,\ b)\ s$ adds an equality $x = b$ to s, possibly removing an equality $x = b'$ in s.

val $union : Th.t \to Fact.equal \to t \to t$

501. $name\ s\ a$ returns the variable x if there is an equation $x = a$ in s. Otherwise, it creates a fresh variiable x' and installs a solution $x' = a$ in s.

val $name : Th.t \to Term.t \times t \to Term.t \times t$

502. $fuse\ norm\ (p,\ s)\ r$ propagates the equalities in r on the right-hand side of equalities in s. The return value $(p',\ s')$ consists vi of an extension of the partition p with newly generated variable equalities and a modified solution set s', which is obtained by transforming every $x = b$ in s to $x = norm(r)(b)$. Here, $norm(r)(b)$ replaces occurrences of z in b with a if $z = a$ is in r and normalizes the result according to the $norm$ argument function. If $norm(r)(b)$ reduces to a variable, say y, then the variable equality $x = y$ is added to the partition p. This may trigger an exception Not_found, if the disequality $x \neq y$ can be deduced from the partition p. This equality is also propagated in the resulting s' in that every occurrence of x is replaced by y. In case $norm(r)(b)$ results in a non-variable b', the equality $x = b'$ is added if there is no $y = b'$ already in the solution set. If there is such a y, then the equality $x = y$ is added to the partitioning p and only one of $x = b'$, $y = b'$ is retained in the resulting solution set.

val $fuse : Th.t \to Partition.t \times t \to Fact.equal\ list \to Partition.t \times t$

503. $compose\ norm\ (p, s)\ r$ is a $fuse$ step followed by extending (and possibly overwriting $x = ...$) the resulting s' with all $x = b$, for b a non-variable term, in sl. If b is a variable, then it is added to v' and ch' is extended accordingly.

val $compose : Th.t \to Partition.t \times t \to Fact.equal\ list \to Partition.t \times t$

504. Every modification or addition of an equality $x = a$ to a solution set s— using $union$, $fuse$, or $compose$—has the side-effect of adding x to the set of changed variables in s. This set can be obtained by $changed\ s$. $reset\ s$ resets the set of changed variables in s to the empty set.

module $Changed$: sig

  type $t = Term.Set.t\ Th.Array.arr$

```
  val reset : unit → unit
  val save : unit → t
  val restore : t → unit
  val stable : unit → bool
end
```

Module Solution.ml

505. type t = {

```
    find : (Term.t × Fact.justification option) Map.t;
    inv : Term.t Map.t;
    use : Use.t
}
let empty = {
  find = Map.empty;
  inv = Map.empty;
  use = Use.empty
}
let is_empty s =
  s.find ≡ Map.empty
let eq s t =
  s.find ≡ t.find
```

506. Changed sets.

```
let changed = Array.create Set.empty
module Changed = struct
  type t = Set.t Th.Array.arr
  let reset () = Array.reset changed Set.empty
  let restore =
    Array.iter (Array.set changed)
  let save () = Array.copy changed
  let stable () =
    Array.for_all Set.is_empty changed
end
```

507. Fold over the *find* structure.

```
let fold f s = Term.Map.fold f s.find
```

508. Solution set

```
let to_list s =
  fold (fun x (b,_) acc -> (x, b) :: acc) s []
```

509. Pretty-printer.

```
let pp i fmt s =
  Pretty.string fmt "\n";
  Th.pp fmt i;
  Pretty.string fmt ":";
  Pretty.list (Pretty.eqn Term.pp) fmt (to_list s)

let apply s x =
  match x with
    | App _ -> raise Not_found (* Invariant: only vars in domain of s. *)
    | _ -> fst(Map.find x s.find)

let find s x =
  match x with
    | App _ -> x
    | _ -> (try fst(Map.find x s.find) with Not_found -> x)

let justification s x =
  match x with
    | App _ -> raise Not_found
    | _ -> Map.find x s.find

let equality s x =
  let (a, prf) = justification s x in
    Fact.mk_equal x a prf

let inv s a = Map.find a s.inv

let mem s x =
  Map.mem x s.find

let use s = Use.find s.use
```

510. Does a variable occur in s.

```
let occurs s x =
  mem s x ∨ ¬ (Term.Set.is_empty (use s x))
```

511. *union x b s* adds new equality $x = b$ to s, possibly overwriting an equality $x = ...$

```
let union i e s =
  let (x, b, j) = Fact.d_equal e in
  assert(is_var x);
  Trace.msg (to_string i) "Update" (x, b) Term.pp_equal;
```

```
if Term.eq x b then
  s
else
  let use' =
    try
      Use.remove x (fst(Map.find x s.find)) s.use
    with
        Not_found -> s.use
  in
    Array.set changed i (Set.add x (Array.get changed i));
    {find = Map.add x (b, j) s.find;
     inv = Map.add b x s.inv;
     use = Use.add x b use'}
```

512. Extend with binding $x = b$, where x is fresh

```
let extend i b s =
  let x = Term.mk_fresh_var (Name.of_string "s") None in
  let e = Fact.mk_equal x b (Fact.mk_rule "extend" []) in
    Trace.msg (to_string i) "Extend" e Fact.pp_equal;
    (x, union i e s)
```

513. Restrict domain.

```
let restrict i x s =
  try
    let b = fst(Map.find x s.find) in
      Trace.msg (to_string i) "Restrict" x Term.pp;
      Array.set changed i (Set.remove x (Array.get changed i));
      {find = Map.remove x s.find;
       inv = Map.remove b s.inv;
       use = Use.remove x b s.use}
  with
      Not_found -> s
```

514. *name e a* returns a variable x if there is a solution $x = a$. Otherwise, it creates a new name x' and installs a solution $x' = a$ in e.

```
let name i (b, s) =
  try
    (inv s b, s)
  with
      Not_found -> extend i b s

let rec norm i r a =
  let eqs = ref [] in
  let assoc x = function
```

```
    | [] → x
    | e :: el →
        let (a, b, prf) = Fact.d_equal e in
          if Term.eq x a then
            begin
              eqs := prf :: !eqs;
              b
            end
          else
            assoc x el
  in
  let b = Th.map i (fun x → assoc x r) a in
    (b, !eqs)

and assoc x = function
  | [] → x
  | e :: el →
      let (a, b, prf) = Fact.d_equal e in
        if Term.eq x a then
          begin
            eqs := prf :: !eqs;
            b
          end
        else
          assoc x el

and eqs = ref []
```

515. Fuse.

```
let rec fuse i (p, s) r =
  Trace.msg (Th.to_string i) "Fuse" r (Pretty.list Fact.pp_equal);
  Set.fold
    (fun x acc →
       try
         let (b, prf) = justification s x in (* prf |- x = b. *)
         let (b', prfs) = norm i r b in (* prfs |- b = b'. *)
         let e' = Fact.mk_equal x b' (Fact.mk_rule "trans" (prf :: prfs)) in
           update i e' acc
       with
           Not_found → acc)
    (dom s r)
    (p, s)

and dom s r =
  List.fold_right
    (fun e →
```

```
        let (x, _, _) = Fact.d_equal e in
          Set.union (use s x))
    r Set.empty

and update i e (p, s) =
  let (x, b, prf1) = Fact.d_equal e in (* prf1 |- x = b. *)
  assert(is_var x);
  if Term.eq x b then
    (p, restrict i x s)
  else if is_var b then
    vareq i e (p, s)
  else
    try
      let y = inv s b in
        if Term.eq x y then (p, s) else
          let e' = Fact.mk_equal x y None in
          let p' = Partition.merge e' p in
          let s' =
            if y <<< x then
              restrict i x s
            else
              let s' = restrict i y s in
                union i e s'
          in
            (p', s')
    with
        Not_found ->
          let s' = union i e s in
            (p, s')

and vareq i e (p, s) =
  let (x, y, prf1) = Fact.d_equal e in (* prf1 |- x = y. *)
  let p' = Partition.merge e p in
  let s' =
    try
      let (a, prf2) = justification s y in (* prf2 |- y = a. *)
        if y <<< x then
          restrict i x s
        else
          let e' = Fact.mk_equal x a (Fact.mk_rule "trans" [prf1; prf2]) in
          let s' = restrict i y s in
            union i e' s'
    with
        Not_found ->
          restrict i x s
```

```
  in
    (p', s')
```

516. Composition.

```
let compose i (p, s) r =
  Trace.call (Th.to_string i) "Compose" r (Pretty.list Fact.pp_equal);
  let (p', s') = fuse i (p, s) r in
  let (p'', s'') = List.fold_right (update i) r (p', s') in
    Trace.exit (Th.to_string i) "Compose" () Pretty.unit;
    (p'', s'')
```

6.42 Module context

Interface for module Context.mli

517. Module *Context*: The logical context of the decision procedures.

518. A logical context is given by a tuple $(ctxt, u, i, d)$ where

- *ctxt* is the set of atoms used for building up this logical context.
- u consists of a set of equalities $x = y$ over terms x, y which are either uninterpreted or interpreted constants (see module U).
- i is the context of interpreted solution sets (see module Th).
- d stores the known disequalities over uninterpreted terms (see module D).

```
type t

val ctxt_of : t → Atom.Set.t
val p_of : t → Partition.t
val eqs_of : t → Th.t → Solution.t
val upper_of : t → int

val v_of : t → V.t
val d_of : t → D.t
val c_of : t → C.t

val empty : t
```

519. Pretty-printing the context of a state.

```
val pp : t Pretty.printer
```

520. Identity test on contexts.

val $eq : t \rightarrow t \rightarrow bool$

521. For an term a which is either uninterpreted or an interpreted constant, $type\_of\ s\ a$ returns the most refined type as obtained from both static information encoded in the arity of uninterpreted function symbols (see module Sym) and the constraint context $c\_of\ s$.

val $cnstrnt : t \rightarrow Term.t \rightarrow Cnstrnt.t$

val $is\_equal : t \rightarrow Term.t \rightarrow Term.t \rightarrow Three.t$

522. Variable partitioning.

val $v : t \rightarrow Term.t \rightarrow Term.t$
val $d : t \rightarrow Term.t \rightarrow Term.Set.t$
val $c : t \rightarrow Term.t \rightarrow Cnstrnt.t$

523. Fold over an equivalence class.

val $fold : t \rightarrow (Term.t \rightarrow \alpha \rightarrow \alpha) \rightarrow Term.t \rightarrow \alpha \rightarrow \alpha$

524. $lookup\ s\ a$ returns, if possible, a canonical variable equal to a.

val $lookup : t \rightarrow Term.t \rightarrow Term.t$

525. $choose\ s\ p\ x$ returns z if for some y in the equivalence class of x $p\ y$ yields $Some(z)$. If there is not such y, Not_found is raised.

val $choose : t \rightarrow (Term.t \rightarrow \alpha\ option) \rightarrow Term.t \rightarrow \alpha$

526. Parameterized operators

val $mem : Th.t \rightarrow t \rightarrow Term.t \rightarrow bool$
val $apply : Th.t \rightarrow t \rightarrow Term.t \rightarrow Term.t$
val $find : Th.t \rightarrow t \rightarrow Term.t \rightarrow Term.t$
val $inv : Th.t \rightarrow t \rightarrow Term.t \rightarrow Term.t$
val $use : Th.t \rightarrow t \rightarrow Term.t \rightarrow Term.Set.t$
val $equality : Th.t \rightarrow t \rightarrow Term.t \rightarrow Fact.equal$

527. $sigma$ normal form.

val $sigma : t \rightarrow Sym.t \rightarrow Term.t\ list \rightarrow Term.t$

528. $solve\ i\ s\ (a,\ b)$ applies solver for theory i on equality $a\ =\ b$.

val $solve : Th.t \rightarrow t \rightarrow Fact.equal \rightarrow Fact.equal\ list$

529. Updates.

val $extend : Atom.t \rightarrow t \rightarrow t$

val $union : Th.t \rightarrow Fact.equal \rightarrow t \rightarrow t$

val $restrict : Th.t \rightarrow Term.t \rightarrow t \rightarrow t$

val *fuse* : *Th.t* → *Fact.equal* → *t* → *t*

val *compose* : *Th.t* → *Fact.equal* → *t* → *t*

val *update* : *Partition.t* → *t* → *t*

val *name* : *Th.t* → *t* × *Term.t* → *t* × *Term.t*

530. List all constraints with finite extension.

val *split* : *t* → *Atom.Set.t*

module *Changed* : sig

  type *t*

  val *in_v* : *t* → *Term.Set.t*
  val *in_d* : *t* → *Term.Set.t*
  val *in_c* : *t* → *Term.Set.t*

  val *in_eqs* : *Th.t* → *t* → *Term.Set.t*

  val *reset* : *unit* → *unit*

  val *save* : *unit* → *t*

  val *restore* : *t* → *unit*

  val *stable* : *unit* → *bool*

  val *pp* : *t Pretty.printer*

end

531. Update rules work on the following global variables together with the index for creating new variables. Within a *protect* environment, updates are performed destructively. Global variables are not protected!

val *protect* : (*t* → *t*) → *t* → *t*

Module Context.ml

532. Decision procedure state.

type *t* = {
  mutable *ctxt* : *Atom.Set.t*; (∗ Current context. ∗)
  mutable *p* : *Partition.t*; (∗ Variable partitioning. ∗)
  eqs : *Solution.t Th.Array.arr*; (∗ Theory-specific solution sets. ∗)
  mutable *upper* : *int*; (∗ Upper bound on fresh variable index. ∗)
}

let *empty* = {
  ctxt = *Atom.Set.empty*;

```
  p = Partition.empty;
  eqs = Array.create Solution.empty;
  upper = 0
}
```

533. Accessors for components of partitioning.

```
let ctxt_of s = s.ctxt
let p_of s = s.p
let v_of s = Partition.v_of s.p
let d_of s = Partition.d_of s.p
let c_of s = Partition.c_of s.p
let eqs_of s = Array.get s.eqs
let upper_of s = s.upper
```

534. Equality test. Do not take upper bounds into account.

```
let eq s t =
 Partition.eq s.p t.p ∧
 Array.for_all2
   (fun eqs1 eqs2 →
      Solution.eq eqs1 eqs2)
   s.eqs t.eqs
```

535. Destructive updates.

```
let extend a s =
  (s.ctxt ← Atom.Set.add a s.ctxt; s)

let update s i eqs =
  (Array.set s.eqs i eqs; s)

let install s i (p, eqs) =
  s.p ← p;
  Array.set s.eqs i eqs;
  s

let union i e s =
  update s i (Solution.union i e (eqs_of s i))

let restrict i x s =
  update s i (Solution.restrict i x (eqs_of s i))

let name i (s, b) =
  let (x', ei') = Solution.name i (b, eqs_of s i) in
    (update s i ei', x')
```

536. Shallow copying.

```
let copy s = {
  ctxt = s.ctxt;
```

```
  p = Partition.copy s.p;
  eqs = Array.copy s.eqs;
  upper = s.upper}
```

537. Canonical variables module s.

```
let v s = V.find (v_of s)

let c s = C.apply (c_of s)

let d s = D.deq (d_of s)

let fold s f x = V.fold (v_of s) f (v s x)
```

538. Constraint of a in s.

```
let cnstrnt s =
  let rec of_term a =
    match a with
    | Var _ →
        c s (v s a)
    | App(Arith(op), xl) →
        Arith.tau of_term op xl
    | App(Pp(op), xl) →
        Pp.tau of_term op xl
    | App(Bvarith(op), xl) →
        Bvarith.tau of_term op xl
    | _ →
        raise Not_found
  in
    Trace.func "context" "Cnstrnt" Term.pp Cnstrnt.pp
      of_term
```

539. Choosing a variable.

```
let choose s = V.choose (v_of s)
```

540. Pretty-printing.

```
let pp fmt s =
  let pps i sl =
    if ¬(Solution.is_empty sl) then
      Solution.pp i fmt sl
  in
  Partition.pp fmt s.p;
  Array.iter (fun i eqs → pps i eqs) s.eqs
```

541. Parameterized operations on solution sets.

```
let mem i s = Solution.mem (eqs_of s i)
```

```
let inv i s = Solution.inv (eqs_of s i)

let apply i s = Solution.apply (eqs_of s i)

let find i s = Solution.find (eqs_of s i)

let use i s = Solution.use (eqs_of s i)

let equality i s = Solution.equality (eqs_of s i)
```

542. Variable partitioning.

```
let rec is_equal s x y =
  match Term.is_equal x y with
    | Three.X → Partition.is_equal s.p x y
    | res → res
```

543. *sigma*-normal forms.

```
let sigma s f =
  match f with
    | Arith(op) → Arith.sigma op
    | Product(op) → Tuple.sigma op
    | Bv(op) → Bitvector.sigma op
    | Coproduct(op) → Coproduct.sigma op
    | Fun(op) → Apply.sigma op
    | Pp(op) → Pp.sigma op
    | Arrays(op) → Arr.sigma op
    | Bvarith(op) → Bvarith.sigma op
    | Uninterp _ → mk_app f
```

Component-wise solver. Only defined for fully interpreted theories.

```
let solve i _ =
  Trace.func "context" "solve"
    Fact.pp_equal
    (Pretty.list Fact.pp_equal)
    (Th.solve i)

let fuse i e s =
  install s i (Solution.fuse i (s.p, eqs_of s i) [e])

let rec compose i e s =
  try
    let sl' = solve i s e in
      install s i (Solution.compose i (s.p, eqs_of s i) sl')
  with
      Exc.Unsolved →
        Format.eprintf "Warning:␣Incomplete␣Solver@.";
        ignore i e s
```

```
and ignore i e s =
  let (a, b, prf) = Fact.d_equal e in
  let (x', ei') = Solution.name i (a, eqs_of s i) in
  let (y', ei'') = Solution.name i (b, ei') in
  let e' = Fact.mk_equal x' y' None in
    union i e' s

let update p s = (s.p ← p; s)
```

Lookup terms on rhs of solution sets.

```
let lookup s a =
  match a with
    | Var _ →
        v s a
    | App(f, _) →
        let i = Th.of_sym f in
          try
            let x = inv i s a in
              v s x
          with
              Not_found → a
```

544. List all constraints with finite extension.

```
let split s = C.split (c_of s)
```

545. Administration of changed sets. For each of component v, d, c of the partition there is such a set stored in respective global variables $V.changed$, $D.changed$, and $C.changed$. Here, we define the change sets for the theory-specific solution sets. In addition, functions for saving, resetting, and restoring are provided.

```
module Changed = struct

  type t = Term.Set.t × Term.Set.t × Term.Set.t × Term.Set.t Array.arr

  let reset () =
    Partition.Changed.reset ();
    Solution.Changed.reset ()

  let save () =
    let (v, d, c) = Partition.Changed.save () in
    let e = Solution.Changed.save () in
      (v, d, c, e)

  let restore (v, d, c, e) =
    Partition.Changed.restore (v, d, c);
    Solution.Changed.restore e

  let stable () =
    Partition.Changed.stable () ∧
```

```
    Solution.Changed.stable ()

  let in_v (v, _, _, _) = v
  let in_d (_, d, _, _) = d
  let in_c (_, _, c, _) = c
  let in_eqs i (_, _, _, e) = Array.get e i

  let pp fmt (v, d, c, e) =
    let ppset str xs =
      if ¬(Set.is_empty xs) then
        begin
          Format.fprintf fmt "\n%s: " str;
          Pretty.set Term.pp fmt (Set.elements xs)
        end
    in
      ppset "v" v; ppset "d" d; ppset "c" c;
      Array.iter (fun i → ppset (Th.to_string i)) e

end
```

546. Update rules work on the following global variables together with the index for creating new variables. Within a *protect* environment, updates are performed destructively. Global variables are protected!

```
let protect f s =
  let k' = !Var.k in
  let r' = !V.removable in
  let ch' = Changed.save () in
    try
      Var.k := s.upper;
      Changed.reset ();
      V.removable := Term.Set.empty;
      let s' = f (copy s) in
        s'.upper ← !Var.k;
        Var.k := k';
        V.removable := r';
        Changed.restore ch';
        s'
    with
      | exc →
          Var.k := k';
          V.removable := r';
          Changed.restore ch';
          raise exc
```

6.43 Module can

Interface for module Can.mli

547. Module *Can*: Various canonizers and normalizers.

548. Canonization of terms.

```
val term : Context.t → Term.t → Term.t
val arith : Context.t → Sym.arith → Term.t list → Term.t
val eq : Context.t → Term.t → Term.t → bool
```

549. Normalization of atoms using canonization.

```
val atom : Context.t → Atom.t → Atom.t
```

Module Can.ml

550. Miscellaneous.

```
type sign = Pos of Term.t | Neg of Term.t
exception Found of sign
```

551. Don't use *find* for uninterpreted theory.

```
let rec find th s a =
  if Th.eq th Th.u ∨ Th.eq th Th.app then
    a
  else if Th.is_fully_interp th then
    let b = Context.find th s a in
      b
  else
    findequiv th s a

and findequiv th s a =
  try
    choose s
      (fun x →
         try
           let b = apply th s x in
             Some(b)
         with Not_found → None)
      a
  with
```

```
      Not_found → a
```

552. Canonization of terms.

```
let rec term s =
  Trace.func "canon" "Term" Term.pp Term.pp
    (can s)

and can s a =
  match a with
    | Var _ →
        v s a
    | App(Sym.Arith(op), al) →
        arith s op al
    | App(f, al) →
        let th = Th.of_sym f in
        let interp x = find th s (can s x) in
        let al′ = mapl interp al in
        let a′ = if al ≡ al′ then a else sigma s f al′ in
          lookup s a′

and arith s op l = (* special treatment for arithmetic *)
  match op, l with (* for optimizing memory usage. *)
    | Sym.Num(q), [] →
        lookup s (Arith.mk_num q)
    | Sym.Multq(q), [x] →
        lookup s (Arith.mk_multq q (find Th.la s (can s x)))
    | Sym.Add, [x; y] →
        let a′ = find Th.la s (can s x)
        and b′ = find Th.la s (can s y) in
          lookup s (Arith.mk_add a′ b′)
    | Sym.Add, _ :: _ :: _ →
        let interp a = find Th.la s (can s a) in
        let l′ = mapl interp l in
          lookup s (Arith.mk_addl l′)
    | _ →
        let str = "Ill-formed␣term␣" ^
                    (Pretty.to_string Sym.pp (Sym.Arith(op))) ^
                    (Pretty.to_string (Pretty.list Term.pp) l)
        in
          failwith str
```

553. Canonical Term Equality.

```
let eq s a b =
  Term.eq (term s a) (term s b)
```

554. Canonization of atoms.

```
let rec atom s =
  Trace.func "canon" "Atom" Atom.pp Atom.pp
    (function
       | Atom.True -> Atom.True
       | Atom.Equal(e) -> equal s e
       | Atom.Diseq(d) -> diseq s d
       | Atom.In(c) -> cnstrnt s c
       | Atom.False -> Atom.False)

and equal s e =
  let (a, b, _) = Fact.d_equal e in
  let x' = can s a
  and y' = can s b in
  match Context.is_equal s x' y' with
    | Three.Yes ->
        Atom.mk_true()
    | Three.No ->
        Atom.mk_false()
    | Three.X ->
        let (x'', y'') = crossmultiply s (x', y') in
        Atom.mk_equal (Fact.mk_equal x'' y'' None)

and diseq s d =
  let (a, b, _) = Fact.d_diseq d in
  let x' = can s a
  and y' = can s b in
    match Context.is_equal s x' y' with
      | Three.Yes -> Atom.mk_false()
      | Three.No -> Atom.mk_true()
      | Three.X ->
          let (x'', y'') = crossmultiply s (x', y') in
            Atom.mk_diseq (Fact.mk_diseq x'' y'' None)

and cnstrnt s c =
  let (a, c, _) = Fact.d_cnstrnt c in
  let mk_in x i =
    let (x', i') = normalize s (x, i) in
      Atom.mk_in (Fact.mk_cnstrnt x i' None)
  in
  let a' = can s a in
  try
    let d = Context.cnstrnt s a' in
    match Cnstrnt.cmp c d with
      | Binrel.Sub ->
          mk_in a' c
      | (Binrel.Super | Binrel.Same) ->
```

```
          Atom.mk_true()
        | Binrel.Disjoint →
          Atom.mk_false()
        | Binrel.Singleton(q) →
          let n = lookup s (Arith.mk_num q) in
          Atom.mk_equal (Fact.mk_equal n a' None)
        | Binrel.Overlap(cd) →
          mk_in a' cd
  with
      Not_found → mk_in a' c

and crossmultiply s (a, b) =
  let (a', b') = crossmultiply1 s (a, b) in
    if Term.eq a a' ∧ Term.eq b b' then
      (a, b)
    else
      let (a'', b'') = crossmultiply s (a', b') in
        (can s a'', can s b'')

and crossmultiply1 s =
  Trace.func "shostak" "Crossmultiply"
    (Pretty.pair Term.pp Term.pp)
    (Pretty.pair Term.pp Term.pp)
    (fun (a, b) →
       let da = Pp.denumerator a in
       let db = Pp.denumerator b in
       let d = Pp.lcm da db in
         if Pp.is_one d then (a, b) else
           (Sig.mk_mult a d, Sig.mk_mult b d))

and normalize s (a, c) = (* following still suspicious. *)
  let (a', c') = normalize1 s (a, c) in
    if Term.eq a a' ∧ Cnstrnt.eq c c' then
      (a, c)
    else
      let (a'', c'') = normalize1 s (a', c') in
        (can s a'', c'')

and normalize1 s =
  Trace.func "shostak" "Normalize" Term.pp_in Term.pp_in
    (fun (a, c) →
       let arrange a q ds = (* compute a × ds − q × ds. *)
         Arith.mk_sub (Sig.mk_mult a ds) (Arith.mk_multq q ds)
       in
       let lower dom alpha = Cnstrnt.mk_lower dom (Mpa.Q.zero, alpha) in
       let upper dom alpha = Cnstrnt.mk_upper dom (alpha, Mpa.Q.zero) in
         try
```

```
          match Cnstrnt.d_upper c with (* a < q or a ≤ q. *)
            | Some(dom, q, beta) →
                (match signed_denum s a with
                   | Pos(ds) →
                       (arrange a q ds, lower dom beta)
                   | Neg(ds) →
                       (arrange a q ds, upper dom beta))
            | _ →
                (match Cnstrnt.d_lower c with (*s a > q or a ≥ q. *)
                   | Some(dom, alpha, q) →
                       (match signed_denum s a with
                          | Pos(ds) →
                              (arrange a q ds, upper dom alpha)
                          | Neg(ds) →
                              (arrange a q ds, lower dom alpha))
                   | _ → (a, c))
          with
              Not_found → (a, c))

and signed_denum s a =
  try
    List.iter
      (fun m →
         let (_, x) = Arith.mono_of m in
         let d = Pp.denumerator x in
           if Pp.is_one d then () else
             let i = Context.cnstrnt s d in
               if Cnstrnt.is_pos i then
                 raise (Found(Pos(d)))
               else if Cnstrnt.is_neg i then
                 raise (Found(Neg(d)))
               else
                 ())
      (Arith.monomials a);
    raise Not_found
  with
      Found(res) → res
```

6.44 Module arrays

Interface for module Arrays.mli

555. *propagate* $e\ (p,\ u)$ possibly deduces new equalities from the variable equality e of the form $x\ =\ y$. If both $z1\ =\ select(u,\ j1)$ and $z2\ =\ update(a2, i2, k2)$ are in the solution set u, and if and $u = z2$, $x = i2$, $y = j1$ are known to be true in the partitioning p, then the equality $z1 = k2$ is added to p.

val $propagate : Fact.equal \rightarrow Context.t \rightarrow Context.t$

556. $diseq\ d\ (p,\ u)$ propagates a disequality d of the form $i \neq j$. If $z1 = select(upd,\ j')$, $z2 = update(a, i', x)$ are equalities in u and if $i = i'$, $j = j'$, $upd = z2$ are equalities in the partitioning p, then the variable equality $z1 = z3$ is added to p. Either $z3 = select(a, j)$ is already in u or $z3$ is generated and the equality $z3 = select(a, j)$ is added to u.

val $diseq : Fact.diseq \rightarrow Context.t \rightarrow Context.t$

Module Arrays.ml

557. From the equality $x = y$ and the facts $z1 = select(upd1, j1)$, $z2 = update(a2, i2, k2)$ in s and $upd1 \equiv z2$ mod s, $x \equiv i2$ mod s, $y \equiv j1$ mod s deduce that $z1 = k2$.

```
let rec propagate e s =
  Trace.msg "rule" "Array(=)" e Fact.pp_equal;
  let (x, y, prf) = Fact.d_equal e in
    propagate1 (x, y, prf)
      (propagate1 (y, x, prf) s)

and propagate1 (x, y, prf) s =
    Set.fold
      (fun z1 s1 →
         match apply Th.arr s1 z1 with
           | App(Arrays(Select), [upd1; j1])
               when is_equal s1 y j1 = Three.Yes →
               fold s1
                 (fun z2 s2 →
                    match apply Th.arr s2 z2 with
                      | App(Arrays(Update), [a2; i2; k2])
                          when is_equal s2 x i2 = Three.Yes →
                          let e' = Fact.mk_equal (v s2 z1) (v s2 k2) None in
                            update (Partition.merge e' (p_of s2)) s2
                      | _ → s2)
                 upd1 s1
           | _ → s1)
      (use Th.arr s x)
      s
```

558. Propagating a disequalities. From the disequality $i \neq j$ and the facts $z1 = select(upd,\ j')$, $z2 = update(a, i', x)$, $i = i'$, $j = j'$, $upd = z2$, it follows that $z1 = z3$, where $z3 = select(a, j)$.

```
let rec diseq d s =
```

```
  Trace.msg "rule" "Array(<>)" d Fact.pp_diseq;
  let (i, j, prf) = Fact.d_diseq d in
    diseq1 (i, j, prf)
      (diseq1 (j, i, prf) s)

and diseq1 (i, j, prf) s =
  Set.fold
   (fun z1 s1 →
      match apply Th.arr s1 z1 with
        | App(Arrays(Select), [upd; j'])
            when is_equal s1 j j' = Three.Yes →
            fold s
              (fun z2 s2 →
                  match apply Th.arr s2 z2 with
                    | App(Arrays(Update), [a; i'; _])
                        when is_equal s2 i i' = Three.Yes →
                        let (s', z3) = name Th.arr (s, Arr.mk_select a j) in
                        let e' = Fact.mk_equal (v s2 z1) (v s2 z3) None in
                        let p' = Partition.merge e' (p_of s2) in
                          update p' s'
                    | _ → s2)
              upd s1
        | _ → s)
    (use Th.arr s j)
    s
```

6.45 Module deduce

Interface for module Deduce.mli

559. Module *Deduce*. Deducing new facts from a variable equality.

```
val deduce : Fact.equal → Context.t → Context.t
val of_pprod : Term.t → Sym.pprod × Term.t list → Context.t → Context.t
val of_linarith : Term.t → Sym.arith × Term.t list → Context.t → Context.t
```

Module Deduce.ml

560. Fold over all partitions of a list l.

```
let rec partitions_fold l f e =
  match l with
    | [] →
```

```
      f ([], []) e
  | x :: xl ->
      List.fold_left
        (fun acc (l1, l2) ->
            f ((x :: l1), l2) (f (l1, (x :: l2)) acc))
      []
      (partitions_fold xl f e)
```

561. Deduce new constraints from an equality.

```
let rec deduce e s =
  Trace.msg "deduce" "Deduce" e Fact.pp_equal;
  let (x, b, _) = Fact.d_equal e in
    assert(is_var x);
    match b with
      | Var _ -> of_var x b s
      | App(Pp(pp), yl) -> of_pprod x (pp, yl) s
      | App(Arith(op), yl) -> of_linarith x (op, yl) s
      | _ -> s

and infer x i s =
  let c' = Fact.mk_cnstrnt (v s x) i None in
  let p' = Partition.add c' (p_of s) in
    update p' s
```

562. Deduce new constraints from an equality of the form $x = b$, where x is a variable.

```
and of_var x y s =
  try
    let i = c s x and j = c s y in
      infer x j (infer y i s)
  with
      Not_found -> s

and of_pprod x (op, yl) s =
  Trace.msg "deduce" "Deduce(u)" x Term.pp;
  let j = Pp.tau (c s) op yl in
  try
    let i = c s x in
      match Cnstrnt.cmp i j with
        | Binrel.Same ->
            s
        | Binrel.Disjoint ->
            raise Exc.Inconsistent
        | Binrel.Sub ->
            s
        | Binrel.Super ->
```

```
            infer x i s
        | Binrel.Singleton(q) →
            infer x (Cnstrnt.mk_singleton q) s
        | Binrel.Overlap(ij) →
            infer x ij s
      with
          Not_found →
            infer x j s

and of_linarith x (op, yl) s =
  Trace.msg "deduce" "Deduce(a)" (x, Arith.sigma op yl) Term.pp_equal;
  match op, yl with
    | Num(q), [] →
        of_num x q s
    | Multq(q), [y] →
        of_multq x (q, y) s
    | Add, ml →
        of_add x ml s
    | _ →
        assert false

and of_num x q s = (* case x = q. *)
  let j' = Cnstrnt.mk_singleton q in
    infer x j' s

and of_multq x (q, y) s = (* case x = q × y. *)
  assert(¬(Mpa.Q.is_zero q));
  try
    let i' = Cnstrnt.multq q (c s y) in (* 1. x in q ** C(y) *)
    let j' = Cnstrnt.multq (Mpa.Q.inv q) (c s x) in (* 2. y in 1/q ** C(x) *)
      infer x i' (infer y j' s)
  with
      Not_found → s

and is_unbounded s = function
  | App(Arith(Num _), []) →
      false
  | App(Arith(Multq(_)), [x]) →
      (try Cnstrnt.is_unbounded (c s x) with Not_found → true)
  | x →
      (try Cnstrnt.is_unbounded (c s x) with Not_found → true)

and cnstrnt_of_monomial s =
  Trace.func "deduce" "Cnstrnt_of_monomial" Term.pp Cnstrnt.pp
    (function
       | App(Arith(Num(q)), []) →
           Cnstrnt.mk_singleton q
```

```
        | App(Arith(Multq(q)), [x]) ->
            (try Cnstrnt.multq q (c s x) with Not_found -> Cnstrnt.mk_real)
        | x ->
            (try c s x with Not_found -> Cnstrnt.mk_real))

and cnstrnt_of_monomials s = function
  | [] ->
      Cnstrnt.mk_zero
  | [m] ->
      cnstrnt_of_monomial s m
  | [m1; m2] ->
      let i1 = cnstrnt_of_monomial s m1 in
      let i2 = cnstrnt_of_monomial s m2 in
      Cnstrnt.add i1 i2
  | m :: ml ->
      let i = cnstrnt_of_monomial s m in
      Cnstrnt.add i (cnstrnt_of_monomials s ml)

and of_add x ml s = (* case x = q1 × y1 + ... + qn × yn. *)
  of_add_bounds x ml
    (of_add_int x ml s)

and of_add_bounds x ml s =
  let ml' = Arith.mk_neg x :: ml in (* ml' not necessarily orderd. *)
    Trace.msg "deduce" "Add_bounds" ml' (Pretty.infixl Term.pp "++");
    match partition_unbounded s ml' with
      | yl, [] -> (* subcase: all monomials unbound. *)
          let i = cnstrnt_of_monomials s yl in
            infer x i s
      | [], zl -> (* subcase: all monomials bound. *)
          propagate_zero zl s
      | [(Var _ as y)], zl -> (* subcase: only one variable is unbound. *)
          let i = Cnstrnt.multq Mpa.Q.negone (cnstrnt_of_monomials s zl) in
            infer y i s
      | [App(Arith(Multq(q)), [y])], zl -> (* subcase: q × y + zl = 0 with y bound, zl
unbound *)
          let i = cnstrnt_of_monomials s zl in (* thus: y in -1/q ** C(zl) *)
          let i' = Cnstrnt.multq (Mpa.Q.minus (Mpa.Q.inv q)) i in
            infer y i' s
      | yl, zl ->
          extend (yl, zl) s

and partition_unbounded s =
  Trace.func "deduce" "Partition"
    (Pretty.list Term.pp)
    (Pretty.pair (Pretty.list Term.pp) (Pretty.list Term.pp))
    (List.partition (is_unbounded s))
```

```
and extend (yl, zl) s = (* yl + zl = 0. *)
  let a = Can.term s (Arith.mk_addl yl) in
    if Arith.is_num a then
      s
    else
      let i' = Cnstrnt.multq Mpa.Q.negone (cnstrnt_of_monomials s zl) in
        if is_var a then
          infer a i' s
        else
          let x = Var(Var.mk_slack None) in
          let e = Fact.mk_equal x a None in
            Trace.msg "deduce" "Extend" e Fact.pp_equal;
            compose Th.la e (infer x i' s)
```

563. Propagate constraints for $ml = 0$ for each variable x in b. Suppose b is of the form $pre + q \times x + post'$, then x in -1/$q \times (j + k)$ is derived, where pre in j and $post'$ in k. Following should be optimized.
this needs to be done for every subterm.

```
and propagate_zero ml =
  Trace.msg "deduce" "Propagate" ml (Pretty.list Term.pp);
  let rec loop j post s =
    match post with
      | [] → s
      | m :: post' →
          let (q, x) = Arith.mono_of m in
          let qinv = Mpa.Q.inv q in
          let k = cnstrnt_of_monomials s post' in
          let j' =
            try Cnstrnt.add (Cnstrnt.multq qinv (c s x)) j
            with Not_found → Cnstrnt.mk_real in
          let i' = Cnstrnt.multq (Mpa.Q.minus qinv) (Cnstrnt.add j k) in
          let s' = infer x i' s in
            loop j' post' s'
  in
    loop Cnstrnt.mk_zero ml
```

564. And now also propagate the integer information.

```
and of_add_int x ml s =
  let ml' = Arith.mk_neg x :: ml in (* ml' not necessarily orderd. *)
  Trace.msg "deduce" "Int" ml' (Pretty.list Term.pp);
  match List.filter (fun m → ¬ (is_int s m)) ml' with
    | [] →
        s
    | [Var _ as x] →
```

```
          infer x Cnstrnt.mk_int s
      | [App(Arith(Num(q)), [])] →
          if Mpa.Q.is_integer q then s else raise Exc.Inconsistent
      | [App(Arith(Multq(q)), [x])]
          when Mpa.Q.is_integer q →
          infer x Cnstrnt.mk_int s
      | ql →
          let a = Can.term s (Arith.mk_addl ql) in
            match Arith.d_num a with
              | Some(q) →
                  if Mpa.Q.is_integer q then s else raise Exc.Inconsistent
              | _ →
                  if is_var a then
                    infer a Cnstrnt.mk_int s
                  else
                    s (* do not introduce new names for now. *)

and is_int s m =
  let is_int_var x =
    try
      Cnstrnt.dom_of (c s x) = Dom.Int
    with
        Not_found → false
  in
    match m with
      | App(Arith(Num(q)), []) →
          Mpa.Q.is_integer q
      | App(Arith(Multq(q)), [x]) →
          Mpa.Q.is_integer q ∧
          is_int_var x
      | _ →
          is_int_var m
```

6.46 Module rule

Interface for module Rule.mli

565. Module *Rule*: encoding of logical rules for manipulating contexts.

type $rule$ = $Context.t \rightarrow Context.t$

type α $transform$ = $Context.t \times \alpha \rightarrow Context.t \times \alpha$

566. Merge variable equality.

val $merge$: $Fact.equal \rightarrow rule$

567. Add a constraint.

val add : $Fact.cnstrnt \rightarrow rule$

568. Add a disequality.

val $diseq$: $Fact.diseq \rightarrow rule$

569. Abstract.

val $abstract\_equal$: $Fact.equal\ transform$
val $abstract\_diseq$: $Fact.diseq\ transform$
val $abstract\_cnstrnt$: $Fact.cnstrnt\ transform$

570. Tactics.

val $prop$: $Fact.equal \rightarrow rule$

val $groebner\_completion$: $bool\ ref$

val $groebner$: $Fact.equal \rightarrow rule$

571. Close.

val $close$: $rule$

val $maxclose$: $int\ ref$

Module Rule.ml

572. type $rule = Context.t \rightarrow Context.t$

type $\alpha\ transform = Context.t \times \alpha \rightarrow Context.t \times \alpha$

573. Extend with fresh variable equality.

```
let extend =
  let v = Name.of_string "v" in
    (fun (s, a) →
       assert(¬(is_var a));
       let x = Var(Var.mk_fresh v None) in
       let i = Th.of_sym (sym_of a) in
       let e = Fact.mk_equal x a None in
         (union i e s, x))
```

574. Variable abstract an atom

575. Abstraction.

```
let rec abstract_equal (s, e) =
  Trace.msg "rule" "Abstract" e Fact.pp_equal;
  let (a, b, _) = Fact.d_equal e in
  let (s', x') = abstract_toplevel_term (s, a) in
  let (s'', y') = abstract_toplevel_term (s', b) in
  let e' = Fact.mk_equal x' y' None in
    (s'', e')

and abstract_diseq (s, d) =
  Trace.msg "rule" "Abstract" d Fact.pp_diseq;
  let (a, b, _) = Fact.d_diseq d in
  let (s', x') = abstract_toplevel_term (s, a) in
  let (s'', y') = abstract_toplevel_term (s', b) in
  let d' = Fact.mk_diseq x' y' None in
    (s'', d')

and abstract_cnstrnt (s, c) =
  Trace.msg "rule" "Abstract" c Fact.pp_cnstrnt;
  let (a, i, _) = Fact.d_cnstrnt c in
  let (s', a') = abstract_term la (s, a) in (* not necessarily a variable. *)
  let c' = Fact.mk_cnstrnt a' i None in
    (s', c')

and abstract_toplevel_term (s, a) =
  abstract_term u (s, a)

and abstract_term i (s, a) =
  match a with
    | Var _ →
        (s, a)
    | App(f, al) →
        let j = Th.of_sym f in
        let (s', al') = abstract_args j (s, al) in
        let a' = if Term.eql al al' then a else sigma s f al' in
          if i = u ∨ i = arr ∨ i ≠ j then
            try
              let x' = inv j s' a' in
              (s', v s' x')
            with
                Not_found → extend (s', a')
          else
            (s', a')

and abstract_args i (s, al) =
  match al with
    | [] →
        (s, [])
```

```
  | b :: bl →
      let (s', bl') = abstract_args i (s, bl) in
      let (s'', b') = abstract_term i (s', b) in
        if Term.eq b b' ∧ bl ≡ bl' then
          (s'', al)
        else
          (s'', b' :: bl')
```

576. Adding equalities/disequalities/constraints to partition.

```
let merge e s =
  Trace.msg "rule" "Merge" e Fact.pp_equal;
  update (Partition.merge e (p_of s)) s

let diseq d s =
  Trace.msg "rule" "Diseq" d Fact.pp_diseq;
  update (Partition.diseq d (p_of s)) s

let add c s =
  Trace.msg "rule" "Add" c Fact.pp_cnstrnt;
  let (a, i, _) = Fact.d_cnstrnt c in
    if is_var a then
      update (Partition.add c (p_of s)) s
    else
      let x' = Var(Var.mk_slack None) in
      let c' = Fact.mk_cnstrnt x' i None in
      let e' = Fact.mk_equal x' a None in
        Trace.msg "foo" "Slackify" e' Fact.pp_equal;
        let s' = update (Partition.add c' (p_of s)) s in
          compose la e' s'
```

577. Sequential composition

```
let (&&&) f g x = g (f x)
```

578. Applying a rule f for all equalities $x = a$ in theory-specific solution set s of index i such that x is equivalent to some y in the changed set ch.

```
let fold i ch f =
  Set.fold
    (fun x s →
       try
         f (equality i s (v s x)) s
       with
           Not_found → s)
    ch
```

579. Applyfing rule f e for all equalities $x = y$ such that x is in the changed set ch and accumulating the results.

```
let foldv ch f =
  Set.fold
    (fun x s ->
       try
         f (V.equality (v_of s) x) s
       with
           Not_found -> s)
    ch
```

580. Applying rule $f\ d$ for all disequalities d in s of the form $x \neq y$ where x is in ch, and accumulating the results.

```
let foldd ch f =
  Set.fold
    (fun x s ->
       try
         let dl = D.disequalities (d_of s) x in
           List.fold_right f dl s
       with
           Not_found -> s)
    ch

let foldc ch f =
  Set.fold
   (fun x s ->
      try
        f (Partition.cnstrnt (p_of s) x) s
      with
          Not_found -> s)
    ch
```

581. Propagating merged equalities into theory-specific solution sets.

```
let rec prop_star ch = foldv ch prop

and prop e =
  (compose la e &&&
   compose p e &&&
   compose bv e &&&
   compose cop e &&&
   fuse u e &&&
   fuse pprod e &&&
   fuse app e &&&
   fuse arr e &&&
   fuse bvarith e)

and compose i e s =
  Trace.msg "rule" "Compose" e Fact.pp_equal;
```

```
let (x, y, _) = Fact.d_equal e in
  if ¬(Set.is_empty (use i s x)) then (* x occurs on rhs. *)
    let b = find i s y in
    let e' = Fact.mk_equal x b None in
      fuse i e' s
  else
    try
      let a = apply i s x in
        try
          let b = apply i s y in
            if Term.eq a b then s else
              let e' = Fact.mk_equal a b None in
                Context.compose i e' s
        with
            Not_found →
              let e' = Fact.mk_equal y a None in
                Context.compose i e' s (* (Context.restrict i x s) *)
    with
        Not_found → s (* x occurs neither on rhs nor on lhs. *)
```

582. Deduce new facts from changes in the linear arithmetic part.

```
let rec arith_star ch =
  fold la ch arith_deduce

and arith_deduce e s =
  Trace.msg "rule" "Arith_deduce" e Fact.pp_equal;
  let (x, b, _) = Fact.d_equal e in
    match b with
      | App(Arith(op), xl) →
          Deduce.of_linarith x (op, xl) s
      | _ → s
```

583. Groebner basis completion for power products.

```
let groebner_completion = ref false

let rec pprod_star (che, chc) s =
  (groebner_star che &&&
   fold pprod che pproduct_deduce &&&
   foldc chc pproduct_cnstrnt) s

and groebner_star ch s =
  if ¬(!groebner_completion) then s else
    fold pprod ch groebner s

and groebner e s =
  if ¬(!groebner_completion) then s else
```

```
    let (x, pp, _) = Fact.d_equal e in
      Fact.Equalset.fold
        (fun e' s ->
           let (y, qq, _) = Fact.d_equal e' in
           let (p, q, gcd) = Pp.gcd pp qq in (* now x × p = gcd and y × q = gcd *)
             if Pp.is_one gcd ∨ Pp.is_one p ∨ Pp.is_one q then
               s
             else
               let xp = Pp.mk_mult (v s x) p
               and yq = Pp.mk_mult (v s y) q in
                 try
                   let u1 = Context.inv pprod s xp in
                   let u2 = Context.inv pprod s yq in
                     match Context.is_equal s u1 u2 with
                       | Three.Yes ->
                           s
                       | Three.No ->
                           raise Exc.Inconsistent
                       | Three.X ->
                           let e'' = Fact.mk_equal (v s u1) (v s u2) None in
                             Trace.msg "rule" "Groebner" e'' Fact.pp_equal;
                             merge e'' s
                 with
                     Not_found -> s)
        (cuts e s)
        s

and cuts e s =
  let (_, pp, _) = Fact.d_equal e in
  let es =
    Set.fold
      (fun y acc1 ->
         (Set.fold
            (fun u acc2 ->
               try
                 let e' = equality pprod s u in
                   Fact.Equalset.add e' acc2
               with
                   Not_found -> acc2)
            (use pprod s y)
            acc1))
      (vars_of pp)
      Fact.Equalset.empty
  in
    Fact.Equalset.remove e es
```

```
and pproduct_deduce e s =
  Trace.msg "rule" "Nonlin_deduce" e Fact.pp_equal;
  let (x, b, _) = Fact.d_equal e in
    match b with
      | App(Pp(op), xl) →
          Deduce.of_pprod x (op, xl) s
      | _ → s

and pproduct_cnstrnt c s =
  Trace.msg "rule" "Nonlin_cnstrnt" c Fact.pp_cnstrnt;
  s
```

584. Propagate variable equalities and disequalities into array equalities.

```
let arrays_star (che, chd) =
  fold arr che Arrays.propagate &&&
  foldd chd Arrays.diseq
```

585. Deduce new facts from changes in the constraint part.

```
let rec cnstrnt_star ch =
  foldc ch cnstrnt

and cnstrnt c =
  Trace.msg "rule" "Cnstrnt" c Fact.pp_cnstrnt;
  (cnstrnt_singleton c &&&
   cnstrnt_diseq c &&&
   cnstrnt_equal la c &&&
   cnstrnt_equal pprod c)

and cnstrnt_equal th c s =
  Trace.msg "rule" "Cnstrnt(=)" c Fact.pp_cnstrnt;
  let (x, i, _) = Fact.d_cnstrnt c in
  try
    let b = apply th s x in
    let e = Fact.mk_equal x b None in
      Deduce.deduce e s
  with
      Not_found →
        (Set.fold
           (fun y s →
              try
                let e = equality th s y in
                  Deduce.deduce e s
              with
                  Not_found → s)
           (use th s x)
           s)
```

```
and cnstrnt_diseq c s =
  Trace.msg "rule" "Cnstrnt(<>)" c Fact.pp_cnstrnt;
  let (x, i, _) = Fact.d_cnstrnt c in
  match Cnstrnt.d_singleton i with
    | None -> s
    | Some(q) ->
        let j = Cnstrnt.mk_diseq q in
          Set.fold
            (fun y s ->
               let c' = Fact.mk_cnstrnt y j None in
                 update (Partition.add c' (p_of s)) s)
            (d s x) s

and cnstrnt_singleton c s =
  Trace.msg "rule" "Cnstrnt(single)" c Fact.pp_cnstrnt;
  let (x, i, _) = Fact.d_cnstrnt c in
  match Cnstrnt.d_singleton i with
    | None -> s
    | Some(q) ->
        let e = Fact.mk_equal x (Arith.mk_num q) None in
          compose la e s
```

586. Normalization step. Remove all variables x which are are scheduled for removal in the partitioning. Check also that this variable x does not occur in any of the solution sets. Since x is noncanonical, this check only needs to be done for the u part, since all other solution sets are kept in canonical form.

```
let compactify = ref true

let normalize s =
  let filter =
    Set.filter
      (fun x ->
         ¬ (mem u s x) ∧ (* left-hand sides of these solution sets. *)
         ¬ (mem pprod s x) ∧ (* are not kept in canonical form. *)
         ¬ (mem app s x) ∧
         ¬ (mem arr s x) ∧
         ¬ (mem bvarith s x))
  in
  if !compactify then
    let xs = !V.removable in
    let xs' = filter xs in
      Trace.msg "rule" "GC" (Term.Set.elements xs') (Pretty.set Term.pp);
      let p' = Partition.restrict xs' (p_of s) in
        update p' s
  else
```

```
s
```

587. *close s* applies the rules above until the resulting state is unchanged.

```
let maxclose = ref 20 (* value -1 is unbounded *)

exception Maxclose

let rec close s =
  let n = ref 0 in
  let s = ref s in
    try
      while ¬(Changed.stable()) do
        let ch = Changed.save () in
        Changed.reset ();
        s := close1 ch !s;
        n := !n + 1;
        if !n > !maxclose then
          raise Maxclose
      done;
      normalize !s
    with
        Maxclose →
          Format.eprintf "\nPossible incompleteness: Upper bound %d reached.@." !maxclose;
          !s

and close1 ch =
  Trace.msg "close" "Close" ch Changed.pp;
  let chv = Changed.in_v ch in
  let chd = Changed.in_d ch in
  let chc = Changed.in_c ch in
  let ch i = Changed.in_eqs i ch in
    prop_star chv &&&
    arith_star (ch la) &&&
    cnstrnt_star chc &&&
    pprod_star (ch pprod, chc) &&& (* Propagate into Power products. *)
    arrays_star (chv, chd) (* Propagate into arrays. *)
```

6.47 Module process

Interface for module Process.mli

588. Module *Shostak*: extending a logical context using a version of Shostak's algorithm.

589. *process s a* extends the logical context s with an atom a. The return value is *Valid* if a can be shown to be valid in s, *Inconsistent* if s conjoined with a can be shown to be inconsistent, and *Satisfiable*(s') otherwise. In the latter case, s' is a logical state equivalent to s conjoined with a.

```
type α status =
  | Valid
  | Inconsistent
  | Ok of α

val pp : α Pretty.printer → α status Pretty.printer

val atom : Context.t → Atom.t → Context.t status
```

Module Process.ml

590. Processing an atom

```
type α status =
  | Valid
  | Inconsistent
  | Ok of α

let pp pp fmt = function
  | Valid → Format.fprintf fmt ":valid"
  | Inconsistent → Format.fprintf fmt ":unsat"
  | Ok(x) → Format.fprintf fmt ":ok␣"; pp fmt x

let rec atom s =
  Trace.func "shostak" "Process" Atom.pp (pp Context.pp)
    (fun a →
       try
         match Can.atom s a with
           | Atom.True →
               Valid
           | Atom.False →
               Inconsistent
           | Atom.Equal(e) →
               Ok(merge a e s)
           | Atom.Diseq(d) →
               Ok(diseq a d s)
           | Atom.In(c) →
               Ok(add a c s)
         with
             Exc.Inconsistent → Inconsistent)

and merge a e =
  Context.protect
    (fun s →
```

```
        let s = Context.extend a s in
        let (s', e') = Rule.abstract_equal (s, e) in
        let s'' = Rule.merge e' s' in
          Rule.close s'')

and add a c =
  Context.protect
    (fun s ->
        let s = Context.extend a s in
        let (s', c') = Rule.abstract_cnstrnt (s, c) in
        let s'' = Rule.add c' s' in
          Rule.close s'')

and diseq a d =
  Context.protect
    (fun s ->
        let s = Context.extend a s in
        let (s', d') = Rule.abstract_diseq (s, d) in
        let s'' = Rule.diseq d' s' in
          Rule.close s'')
```

6.48 Module symtab

Interface for module Symtab.mli

591. Module *Symtab*: Symbol table functions.

```
type entry =
  | Def of Term.t
  | Arity of int
  | Type of Cnstrnt.t
  | State of Context.t

and t
```

592. *lookup n s* lookup value *e* if binding n |− > e is in the table s; otherwise exception *Not_found* is raised.

```
val lookup : Name.t → t → entry
```

593. Empty symbol table.

```
val empty : t
```

594. Adding a binding n |− > e to a symbol table. Throws *Invalid_argument*, if n is already in the domain of the table.

```
val add : Name.t → entry → t → t
```

595. Removing an entry n |− > ... from the symbol table.

```
val remove : Name.t → t → t
```

596. *filter p s* builds a subtable of s with all bindings n |− > e satisfying predicate $p\ n\ e$.

```
val filter : (Name.t → entry → bool) → t → t
```

597. Projections.

```
val def : t → t
val arity : t → t
val typ : t → t
val state : t → t
```

598. Pretty printing of a symbol table.

```
val pp : t Pretty.printer

val pp_entry : entry Pretty.printer
```

Module Symtab.ml

```
type entry =
  | Def of Term.t
  | Arity of int
  | Type of Cnstrnt.t
  | State of Context.t

and t = entry Name.Map.t

let lookup = Name.Map.find

let empty_name = Name.of_string "empty"

let empty =
  Name.Map.add
    empty_name
    (State (Context.empty))
    Name.Map.empty

let add n e s =
  if Name.Map.mem n s then
    raise (Invalid_argument "Name already in table")
  else
    Name.Map.add n e s

let remove n s =
  if Name.eq n empty_name then s else Name.Map.remove n s
```

```
let filter p s =
  Name.Map.fold
    (fun n e acc →
       if p n e then Name.Map.add n e acc else acc)
    s
    Name.Map.empty

let state = filter (fun _ e → match e with State _ → true | _ → false)
let def = filter (fun _ e → match e with Def _ → true | _ → false)
let arity = filter (fun _ e → match e with Arity _ → true | _ → false)
let typ = filter (fun _ e → match e with Type _ → true | _ → false)

let rec pp fmt s =
  Name.pp_map pp_entry fmt s

and pp_entry fmt e =
  let pr = Format.fprintf fmt in
  match e with
    | Def(x) → pr "@[def("; Term.pp fmt x; pr ")@]"
    | Arity(a) → pr "@[sig("; Format.fprintf fmt "%d" a; pr ")@]"
    | Type(c) → pr "@[type("; Cnstrnt.pp fmt c; pr ")@]"
    | State(s) →
        pr "@[state(";
        Pretty.set Atom.pp fmt (Atom.Set.elements (Context.ctxt_of s));
        pr ")@]"
```

6.49 Module istate

Interface for module Istate.mli

599. type *t*

```
val current : unit → Context.t
val get_context : Name.t option → Context.t
val symtab : unit → Symtab.t
val eot : unit → string
val inchannel : unit → in_channel
val outchannel : unit → Format.formatter
```

600. Initialize.

```
val initialize : bool → string → in_channel → Format.formatter → unit
```

601. Resetting the state.

```
val reset : unit → unit
```

602. Setting input and output channels.

val $set\_inchannel$: $in\_channel \rightarrow unit$
val $set\_outchannel$: $Format.formatter \rightarrow unit$

val $flush$: $unit \rightarrow unit$
val nl : $unit \rightarrow unit$

603. Adding definitions to state

val def : $Name.t \rightarrow Term.t \rightarrow unit$
val sgn : $Name.t \rightarrow int \rightarrow unit$
val typ : $Name.t \rightarrow Cnstrnt.t \rightarrow unit$

604. Symbol table entry.

val $entry\_of$: $Name.t \rightarrow Symtab.entry\ option$

605. Type from symbol table.

val $type\_of$: $Name.t \rightarrow Cnstrnt.t\ option$

606. Getting the width of bitvector terms from the signature.

val $width\_of$: $Term.t \rightarrow int\ option$

607. State-dependent destructors.

608. Context of.

val $ctxt\_of$: $Name.t\ option \rightarrow Atom.Set.t$

609. Canonization w.r.t current state.

val can : $Atom.t \rightarrow Atom.t$

val $cant$: $Term.t \rightarrow Term.t$

val $sigma$: $Sym.t \rightarrow Term.t\ list \rightarrow Term.t$

610. Adding a new fact

val $process$: $Name.t\ option \rightarrow Atom.t \rightarrow Name.t\ Process.status$

611. Checking for validity/unsatisfiability.

val $valid$: $Name.t\ option \rightarrow Atom.t \rightarrow bool$

val $unsat$: $Name.t\ option \rightarrow Atom.t \rightarrow bool$

612. Change current state.

val $save$: $Name.t\ option \rightarrow Name.t$
val $restore$: $Name.t \rightarrow unit$
val $remove$: $Name.t \rightarrow unit$

val $forget$: $unit \rightarrow unit$

613. Applying maps.

val $find$: $Name.t$ $option \rightarrow Th.t \rightarrow Term.t \rightarrow Term.t$
val inv : $Name.t$ $option \rightarrow Th.t \rightarrow Term.t \rightarrow Term.t$
val use : $Name.t$ $option \rightarrow Th.t \rightarrow Term.t \rightarrow Term.Set.t$

614. Solution set for equality theories.

val $solution$: $Name.t$ $option \rightarrow Th.t \rightarrow (Term.t \times Term.t)$ $list$

615. Solver.

val $solve$: $Th.t \rightarrow (Term.t \times Term.t) \rightarrow (Term.t \times Term.t)$ $list$

616. Variable partitioning.

val $partition$: $unit \rightarrow (Term.t \times Term.t$ $list)$ $list$

617. Disequalities.

val $diseq$: $Name.t$ $option \rightarrow Term.t \rightarrow Term.Set.t$

618. Constraint.

val $cnstrnt$: $Name.t$ $option \rightarrow Term.t \rightarrow Cnstrnt.t$ $option$

619. Equality/disequality test.

val is_equal : $Term.t \rightarrow Term.t \rightarrow bool$

620. Splitting.

val $split$: $unit \rightarrow Atom.Set.t$

Module Istate.ml

621. Global state.

```
type t = {
  mutable current : Context.t;
  mutable symtab : Symtab.t;
  mutable inchannel : in_channel;
  mutable outchannel : Format.formatter;
  mutable eot : string;
  mutable counter : int
}
let init () = {
  current = Context.empty;
```

```
  symtab = Symtab.empty;
  inchannel = Pervasives.stdin;
  outchannel = Format.std_formatter;
  eot = "";
  counter = 0
}
let s = init ()
```

622. Initialize.

```
let initialize pp eot inch outch =
  Term.pretty := pp;
  s.eot ← eot;
  s.inchannel ← inch;
  s.outchannel ← outch
```

623. Accessors to components of global state.

```
let current () = s.current
let symtab () = s.symtab
let eot () = s.eot
let inchannel () = s.inchannel
let outchannel () = s.outchannel
```

624. Adding to symbol table

```
let def n a =
  let e = Symtab.Def(a) in
  s.symtab ← Symtab.add n e s.symtab

let sgn n a =
  let e = Symtab.Arity(a) in
  s.symtab ← Symtab.add n e s.symtab

let typ n c =
  let e = Symtab.Type(c) in
  s.symtab ← Symtab.add n e s.symtab

let entry_of n =
  try
    Some(Symtab.lookup n s.symtab)
  with
      Not_found → None
```

625. Type from the symbol table.

```
let type_of n =
  match Symtab.lookup n s.symtab with
    | Symtab.Type(c) → Some(c)
```

```
    | _ → None
```

626. Get context for name in symbol table

```
let context_of n =
  match Symtab.lookup n s.symtab with
    | Symtab.State(c) → c
    | _ → raise (Invalid_argument("No context of name " ^ (Name.to_string n)))
```

627. Getting the width of bitvector terms from the signature.

```
let width_of a =
  if Term.is_var a then
    let n = Term.name_of a in
    try
      match Symtab.lookup n s.symtab with
        | Symtab.Arity(i) → Some(i)
        | _ → None
    with
        Not_found → None
  else
    Bitvector.width a
```

628. Resetting all of the global state.

```
let reset () =
  Tools.do_at_reset ();
  s.current ← Context.empty;
  s.symtab ← Symtab.empty;
  s.counter ← 0
```

629. Getting either current context or explicitly specified context.

```
let get_context = function
  | None → s.current
  | Some(n) → context_of n
```

630. Set input and output channels.

```
let set_inchannel ch =
  s.inchannel ← ch

let set_outchannel fmt =
  s.outchannel ← fmt

let flush () = Format.fprintf s.outchannel "@?"
let nl () = Format.fprintf s.outchannel "\n"
```

631. Context.

```
let ctxt_of = function
```

```
  | None  →  Context.ctxt_of s.current
  | Some(n) → Context.ctxt_of (context_of n)
```

632. Canonization w.r.t current state.

```
let can p =
  Can.atom s.current p

let cant a =
  Can.term s.current a

let sigma f l =
  Context.sigma s.current f l
```

633. Create a fresh name for a state.

```
let rec fresh_state_name () =
  s.counter ← s.counter + 1;
  let n = Name.of_string ("s" ^ (string_of_int s.counter)) in
  try
    let _ = Symtab.lookup n s.symtab in (* make sure state name is really fresh. *)
    fresh_state_name ()
  with
      Not_found →
        n
```

634. Change current state.

```
let save arg =
  let n = match arg with
    | None → fresh_state_name ()
    | Some(n) → n
  in
  let e = Symtab.State s.current in
  s.symtab ← Symtab.add n e s.symtab;
  n

let restore n =
  try
    match Symtab.lookup n s.symtab with
      | Symtab.State(t) → s.current ← t
      | _ → raise Not_found
  with
      Not_found → raise (Invalid_argument "Not␣a␣state␣name")

let remove n =
  s.symtab ← Symtab.remove n s.symtab

let forget () =
```

```
  s.current ← Context.empty
```

635. Adding a new fact

```
let process n a =
  let t = (get_context n) in
  let status = Process.atom t a in
  match status with (* Update state and install new name in symbol table *)
    | Process.Ok(t') →
        s.current ← t';
        let n = save None in
        Process.Ok(n)
    | Process.Valid →
        Process.Valid
    | Process.Inconsistent →
        Process.Inconsistent

let valid n a =
  match Process.atom (get_context n) a with
    | Process.Valid → true
    | _ → false

let unsat n a =
  match Process.atom (get_context n) a with
    | Process.Inconsistent → true
    | _ → false
```

636. Accessors.

```
let diseq n a =
  let s = get_context n in
  let a' = Can.term s a in
  try
    Context.d s a'
  with
      Not_found → Term.Set.empty

let cnstrnt n a =
  let s = get_context n in
  let a' = Can.term s a in
  try
    Some(Context.cnstrnt s a')
  with
      Not_found → None
```

637. Applying maps.

```
let find n i x = Context.find i (get_context n) x
let inv n i b = Context.inv i (get_context n) b
```

```
let use n i = Context.use i (get_context n)
```

638. Solution sets.

```
let solution n i =
  Solution.fold
    (fun x (a, _) acc → (x, a) :: acc)
    (Context.eqs_of (get_context n) i)
    []
```

639. Variable partitioning.

```
let partition () =
  Term.Map.fold
    (fun x ys acc →
        (x, Term.Set.elements ys) :: acc)
    (V.partition (Context.v_of s.current))
    []
```

640. Solver.

```
let solve i (a, b) =
  try
    let e = Fact.mk_equal a b None in
      List.map (fun e' →
                    let (x, b, _) = Fact.d_equal e' in
                      (x, b))
        (Th.solve i e)
  with
    | Exc.Inconsistent → raise(Invalid_argument("Unsat"))
    | Exc.Unsolved → raise(Invalid_argument("Unsolvable"))
```

641. Equality/disequality test.

```
let is_equal a b =
  Can.eq s.current a b

let is_int a =
  try
    let c = Context.cnstrnt s.current a in
     Cnstrnt.dom_of c = Dom.Int
  with
      Not_found → false
```

642. Splitting.

```
let split () = Context.split s.current
```

6.50 Module help

Interface for module Help.mli

```
val on_help : unit → unit
val syntax : unit → unit
val commands : unit → unit
```

Module Help.ml

```
let on_help () =
  Format.eprintf "@[";
  Format.eprintf "help.␣␣␣␣␣␣␣␣␣␣␣Displays␣this␣mesage.@\n";
  Format.eprintf "help␣commands.␣␣Lists␣all␣commands.@\n";
  Format.eprintf "help␣syntax.␣␣␣␣Outlines␣syntactic␣categories.@]@."

let commands () =
  Format.eprintf "@[";
  Format.eprintf "assert␣<atom>.␣␣␣␣␣␣␣␣␣␣␣Assert␣atom␣in␣current␣context.@\n";
  Format.eprintf "can␣(<term>␣|␣<atom>).␣␣␣Canonical␣forms.@\n";
  Format.eprintf "cnstrnt␣<term>.␣␣␣␣␣␣␣␣␣␣Constraint␣associated␣with␣term.@\n";
  Format.eprintf "ctxt.␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Current␣logical␣context.@\n";
  Format.eprintf "def␣<var>␣:=␣<term>.␣␣␣␣␣Term␣definition.@\n";
  Format.eprintf "diseq␣<term>.␣␣␣␣␣␣␣␣␣␣␣␣Variables␣known␣to␣be␣disequal.@\n";
  Format.eprintf "exit.␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Exiting␣ICS␣interpreter␣(or:␣Ctrl-D)@\n";
  Format.eprintf "forget.␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Clearing␣current␣logical␣context.@\n";
  Format.eprintf "find␣<th>␣<var>.␣␣␣␣␣␣␣␣␣Interpretation␣of␣variable␣in␣theory.@\n";
  Format.eprintf "gc.␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Garbage␣collection␣in␣current␣context.@\n";
  Format.eprintf "help␣[commands␣|␣syntax].␣Help␣texts.@\n";
  Format.eprintf "inv␣<th>␣<term>.␣␣␣␣␣␣␣␣␣External␣variable␣for␣term␣in␣theory@\n";
  Format.eprintf "partition.␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Current␣variable␣partitioning.@\n";
  Format.eprintf "reset.␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Reinitializing␣ICS.@\n";
  Format.eprintf "restore␣<name>.␣␣␣␣␣␣␣␣␣␣Restore␣current␣context.@\n";
  Format.eprintf "remove␣<name>.␣␣␣␣␣␣␣␣␣␣␣Remove␣a␣saved␣current␣context.@\n";
  Format.eprintf "save␣<name>.␣␣␣␣␣␣␣␣␣␣␣␣␣Save␣current␣context.@\n";
  Format.eprintf "sig␣<var>␣:␣bv[<int>].␣␣␣Signature␣extension.@\n";
  Format.eprintf "sigma␣(<term>␣|␣<atom>).␣Normal␣forms.@\n";
  Format.eprintf "show.␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Solution␣sets␣and␣variable␣partitioning.@\n";
  Format.eprintf "solution␣<th>.␣␣␣␣␣␣␣␣␣␣␣Solution␣sets␣for␣individual␣theory.@\n";
  Format.eprintf "solve␣<ith>␣<term>␣=␣<term>.␣Solve␣equation␣in␣interpreted␣theory.@\n";
  Format.eprintf "symtab␣[<name>].␣␣␣␣␣␣␣␣␣Symbol␣table␣entries.@\n";
  Format.eprintf "trace␣<ident>,...<ident>.␣Tracing.@\n";
  Format.eprintf "type␣<var>␣:=␣<cnstrnt>.␣Type␣definition.@\n";
```

```
  Format.eprintf "untrace.␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Stop␣tracing.@\n";
  Format.eprintf "<term>␣«␣<term>.␣␣␣␣␣␣␣␣Comparing␣two␣terms.@\n\n";
  Format.eprintf "An␣interpreted␣theory␣<ith>␣is␣either␣'a'␣for␣arithmetic,␣'t'␣for@\n";
  Format.eprintf "tuples,␣or␣'bv'␣for␣bitvectors.␣In␣addition,␣<th>␣includes␣'u'␣the@\n";
  Format.eprintf "theory␣of␣unintepreted␣terms";
  Format.eprintf "@]@."

let syntax () =
  let pr = Format.eprintf in
  pr "@[";
  pr "<term>␣::=␣<var>␣@\n";
  pr "␣␣␣␣␣␣␣␣␣␣|␣<name>␣'('␣<term>␣{','␣<term>}*␣')'␣>@\n";
  pr "␣␣␣␣␣␣␣␣␣␣|␣<arith>␣|␣<tuple>␣|␣<array>␣|␣<sexpr>␣|␣<bv>␣|␣<boolean>@\n";
  pr "␣␣␣␣␣␣␣␣␣␣|␣'('␣<term>␣')'@\n\n";
  pr "<var>␣::=␣<ident>␣|␣<ident>␣'!'␣<int>@\n\n";
  pr "<arith>␣::=␣<rational>␣|␣<term>␣{'+'␣|␣'-'␣|␣'*'}␣<term>@\n\n";
  pr "␣␣␣␣␣␣␣␣␣␣␣|␣'-'␣<term>␣|␣<term>␣'^'␣<int>@\n";
  pr "<tuple>␣::=␣'('␣<term>␣{','␣<term>}*␣')'@\n";
  pr "␣␣␣␣␣␣␣␣␣␣␣|␣'proj['␣<int>␣','␣<int>␣']'␣'('␣<term>␣')'@\n\n";
  pr "<sexpr>␣::=␣'nil'␣|␣'cons'␣'('␣<term>␣','␣<term>␣')'@\n";
  pr "␣␣␣␣␣␣␣␣␣␣␣|␣{'car'␣|␣'cdr'}␣'('␣<term>␣')'@\n\n";
  pr "<array>␣::=␣<term>␣'['␣<term>␣':='␣<term>␣']'@\n";
  pr "␣␣␣␣␣␣␣␣␣␣␣|␣<term>␣'['␣<term>␣']'@\n\n";
  pr "<boolean>␣::=␣'true'␣|␣'false'@\n\n";
  pr "<bv>␣::=␣'0b'␣{'0'␣|␣'1'}*␣|␣<term>␣'++'␣<term>@\n";
  pr "␣␣␣␣␣␣␣|␣<term>␣'['<int>':'<int>']'␣'('<term>')'@\n";
  pr "␣␣␣␣␣␣␣|␣<term>␣{'&&'␣|␣'||'␣|␣'##'}␣<term>@\n\n";
  pr "<atom>␣::=␣<term>␣{'='␣|␣'<>'␣|␣'<'␣|␣'>'␣|␣'<='␣|␣'>='}␣<term>@\n";
  pr "␣␣␣␣␣␣␣␣␣␣|␣<term>␣'in'␣<cnstrnt>@\n\n";
  pr "<cnstrnt>␣::=␣<ident>@\n";
  pr "␣␣␣␣␣␣␣␣␣␣␣␣␣|␣('int'|'real')␣{<left>␣'..'␣<right>}␣␣@\n";
  pr "<left>␣::=␣'(-inf'␣|␣'('␣<rat>␣|␣'['␣<rat>@\n";
  pr "<right>␣::=␣'inf)'␣|␣<rat>␣')'␣|␣<rat>␣']'@\n\n";
  pr "<ident>␣::=␣␣('A'|..|'Z'|'a'|..|'z')␣{'A'|..|'Z'|'a'|..|'z'|'\''|'0'|..|'9'}*@\n";
  pr "<int>␣::=␣{'0'|..|'9'}+@\n";
  pr "<rat>␣::=␣<int>␣{'/'␣<int>}@\n\n";
  pr "Comments␣start␣with␣percent␣and␣end␣with␣a␣newline.@\n";
  pr "Identifiers␣do␣not␣include␣any␣keywords␣and␣command␣names.@\n";
  pr "Operator␣precedence␣in␣increasing␣order:@\n";
  pr "'-',␣'+',␣'*',␣'/',␣'^',␣'++',␣'||',␣'##',␣'&&'.@\n";
  pr "Conventions:@\n";
  pr "nonterminals:␣<..>@\n";
  pr "terminals:␣␣'..'@\n";
  pr "choice:␣..␣|␣..@\n";
  pr "optional:␣{␣..␣}@\n";
```

```
pr "zero␣or␣more␣repetions:␣{..}*@\n";
pr "one␣or␣more␣repetions:␣{..}+@\n";
pr "@]@."
```

6.51 Module result

Interface for module Result.mli

643. Module *Result*: result type for commands.

```
type t =
  | Term of Term.t
  | Atom of Atom.t
  | Cnstrnt of Cnstrnt.t option
  | Optterm of Term.t option
  | Name of Name.t
  | Terms of Term.Set.t
  | Atoms of Atom.Set.t
  | Unit of unit
  | Bool of bool
  | Solution of (Term.t × Term.t) list
  | Context of Context.t
  | Process of Name.t Process.status
  | Symtab of Symtab.t
  | Entry of Symtab.entry
  | Int of int
  | String of string
val output : Format.formatter → t → unit
```

Module Result.ml

```
type t =
  | Term of Term.t
  | Atom of Atom.t
  | Cnstrnt of Cnstrnt.t option
  | Optterm of Term.t option
  | Name of Name.t
  | Terms of Term.Set.t
  | Atoms of Atom.Set.t
  | Unit of unit
  | Bool of bool
  | Solution of (Term.t × Term.t) list
```

```
  | Context of Context.t
  | Process of Name.t Process.status
  | Symtab of Symtab.t
  | Entry of Symtab.entry
  | Int of int
  | String of string

let output fmt = function
  | Term(t) → Term.pp fmt t
  | Atom(a) → Atom.pp fmt a
  | Cnstrnt(Some(c)) → Cnstrnt.pp fmt c
  | Cnstrnt(None) → Format.fprintf fmt "None"
  | Optterm(Some(t)) → Term.pp fmt t
  | Optterm(None) → Format.fprintf fmt "None"
  | Name(n) → Name.pp fmt n
  | Terms(ts) → Pretty.set Term.pp fmt (Term.Set.elements ts)
  | Atoms(al) → Pretty.set Atom.pp fmt (Atom.Set.elements al)
  | Unit() → Format.fprintf fmt ""
  | Bool(x) → Format.fprintf fmt "%s" (if x then "true" else "false")
  | Solution(sl) → Pretty.list (Pretty.eqn Term.pp) fmt sl
  | Context(c) → Context.pp fmt c
  | Process(status) → Process.pp Name.pp fmt status
  | Symtab(sym) → Symtab.pp fmt sym
  | Entry(e) → Symtab.pp_entry fmt e
  | Int(i) → Format.fprintf fmt "%d" i
  | String(s) → Format.fprintf fmt "%s" s
```

6.52 Module ics

Interface for module Ics.mli

644. Module *Ics*: the application programming interface to ICS for asserting formulas to a logical context, switching between different logical contexts, and functions for manipulating and normalizing terms.

There are two sets of interface functions. The functional interface provides functions for building up the main syntactic categories of ICS such as terms and atoms, and for extending logical contexts using *process*, which is side-effect free.

In contrast to this functional interface, the command interface manipulates a global state consisting, among others, of symbol tables and the current logical context. The *cmd_rep* procedure, which reads commands from the current input channel and manipulates the global structures accordingly, is used to implement the ICS interactor.

Besides functions for manipulating ICS datatypes, this interface also contains a number of standard datatypes such as channels, multiprecision arithmetic, tuples, and lists.

645. Controls. *reset* clears all the global tables. This does not only include the current context but also internal tables used for hash-consing and memoization purposes. *gc* triggers a full major collection of ocaml's garbage collector. *do_at_exit* clears out internal data structures.

val *reset* : *unit* → *unit*

val *gc* : *unit* → *unit*

val *do_at_exit* : *unit* → *unit*

646. *set_maxloops n* determines an upper number of loops in the main ICS loop. $n < 0$ determines that there is no such bound; this is also the default.

val *set_maxloops* : *int* → *unit*

647. Rudimentary control on trace messages, which are sent to *stderr*. These functions are mainly included for debugging purposes, and are usually not being used by the application programmer. *trace_add str* enables tracing of functions associated with trace level *str*. *trace_add* `"all"` enables all tracing. *trace_remove str* removes *str* from the set of active trace levels, and *trace_reset*() disables all tracing. *trace_get*() returns the set of active trace levels.

val *trace_reset* : *unit* → *unit*
val *trace_add* : *string* → *unit*
val *trace_remove* : *string* → *unit*
val *trace_get* : *unit* → *string list*

648. Channels. *inchannel* is the type of input channels. A channel of name *str* is opened with *in_of_string str*. This function raises *Sys_error* in case such a channel can not be opened. *outchannel* is the type of formatting output channels, and channels of this type are opened with *out_of_string*. *stdin*, *stdout*, and *stderr* are predefined channels for standard input, standard output, and standard error. *flush* flushes the *stdout* channel.

type *inchannel* = *in_channel*
type *outchannel* = *Format.formatter*

val *channel_stdin* : *unit* → *inchannel*
val *channel_stdout* : *unit* → *outchannel*
val *channel_stderr* : *unit* → *outchannel*
val *inchannel_of_string* : *string* → *inchannel*
val *outchannel_of_string* : *string* → *outchannel*

val *flush* : *unit* → *unit*

649. Multi-precision rational numbers. *num_of_int n* injects an integer into this type, *num_of_ints n m*, for $m \neq 0$, constructs a normalized representation of the rational n/m in q, *string_of_num q* constructs a string (usually for printout) of a rational number, and

$num\_of\_string\ s$ constructs a rational, whenever s is of the form "n/m" where n and m are naturals.

type q

val $num\_of\_int\ :\ int \rightarrow\ q$
val $num\_of\_ints\ :\ int \rightarrow\ int \rightarrow\ q$
val $ints\_of\_num\ :\ q\ \rightarrow\ string \times string$
val $string\_of\_num\ :\ q\ \rightarrow\ string$
val $num\_of\_string\ :\ string \rightarrow\ q$

650. Names. $name\_of\_string$ and $name\_to\_string$ coerce between the datatypes of strings and names. These coercions are inverse to each other. $name\_eq$ tests for equality of names in constant time.

type $name$

val $name\_of\_string\ :\ string \rightarrow\ name$
val $name\_to\_string\ :\ name\ \rightarrow\ string$
val $name\_eq\ :\ name\ \rightarrow\ name\ \rightarrow\ bool$

651. Arithmetic constraints. A constraint consists of a domain restriction Int or $Real$, a real interval, and a set of disequality numbers. A real number satisfies such a constraint if, first, it satisfies the domain restriction, second, it is a member of the interval, and, third, it is none of the numbers in the disequality set.

$cnstrnt\_of\_string\ str$ parses the string str according to the nonterminal $cnstrnteof$ in module $Parser$ (see its specification in file parser.mly) and produces the corresponding constraint representation. In contrast, $cnstrnt\_input$ in parses the concrete syntax of constraints from the input channel in. Constraints c are printed to the output channel out using $cnstrnt\_output\ out\ c$ and to the standard output using $cnstrnt\_pp\ c$.

For the definition of constraint constructors see Module $Cnstrnt$. $cnstrnt\_mk\_int()$ constructs an integer constraint, $cnstrnt\_mk\_nat()$ a constraint for the natural numbers, $cnstrnt\_mk\_singleton\ q$ is the constraint which holds only of q, $cnstrnt\_mk\_diseq\ q$ holds for all reals except for q, $cnstrnt\_mk\_oo\ l\ h$ constructs an open interval with lower bound l and upper bound h, $cnstrnt\_mk\_oc\ l\ h$ is the left-open, right-closed interval with lower bound l and upper bound h, $cnstrnt\_mk\_co\ l\ h$ is the left-closed, right-open interval with lower bound l and upper bound h, $cnstrnt\_mk\_cc\ l\ h$ is the closed interval with lower bound l and upper bound h.

The intersection of two constraints c, d is computed by $cnstrnt\_inter\ c\ d$, that is, a real q is in both c and d iff it is in $cnstrnt\_inter\ c\ d$.

type $cnstrnt$

val $cnstrnt\_of\_string\ :\ string \rightarrow\ cnstrnt$
val $cnstrnt\_input\ :\ inchannel\ \rightarrow\ cnstrnt$
val $cnstrnt\_output\ :\ outchannel\ \rightarrow\ cnstrnt\ \rightarrow\ unit$
val $cnstrnt\_pp\ :\ cnstrnt\ \rightarrow\ unit$

val $cnstrnt\_mk\_int\ :\ unit \rightarrow\ cnstrnt$

val $cnstrnt\_mk\_nonint$: $unit \rightarrow cnstrnt$
val $cnstrnt\_mk\_nat$: $unit \rightarrow cnstrnt$
val $cnstrnt\_mk\_singleton$: $q \rightarrow cnstrnt$
val $cnstrnt\_mk\_diseq$: $q \rightarrow cnstrnt$
val $cnstrnt\_mk\_oo$: $q \rightarrow q \rightarrow cnstrnt$
val $cnstrnt\_mk\_oc$: $q \rightarrow q \rightarrow cnstrnt$
val $cnstrnt\_mk\_co$: $q \rightarrow q \rightarrow cnstrnt$
val $cnstrnt\_mk\_cc$: $q \rightarrow q \rightarrow cnstrnt$
val $cnstrnt\_mk\_lt$: $q \rightarrow cnstrnt$
val $cnstrnt\_mk\_le$: $q \rightarrow cnstrnt$
val $cnstrnt\_mk\_gt$: $q \rightarrow cnstrnt$
val $cnstrnt\_mk\_ge$: $q \rightarrow cnstrnt$

val $cnstrnt\_inter$: $cnstrnt \rightarrow cnstrnt \rightarrow cnstrnt$

652. Abstract interval interpretation. A real number x is in $cnstrnt\_add\ c\ d$ iff there are real numbers y in c and z in d such that $x = y + x$. Likewise, x is in $cnstrnt\_multq\ q\ c$ iff there exists y in c such that $x = q \times y$. In constrast to these exact abstract operators, $cnstrnt\_mult$ and $cnstrnt\_div$ compute overaprroximations, that is, if x in c and y in d then there exists a z in $cnstrnt\_mult\ c\ d$ ($cnstrnt\_div\ c\ d$) such that $z = x \times y$ ($z = x/y$).

val $cnstrnt\_add$: $cnstrnt \rightarrow cnstrnt \rightarrow cnstrnt$
val $cnstrnt\_multq$: $q \rightarrow cnstrnt \rightarrow cnstrnt$
val $cnstrnt\_mult$: $cnstrnt \rightarrow cnstrnt \rightarrow cnstrnt$
val $cnstrnt\_div$: $cnstrnt \rightarrow cnstrnt \rightarrow cnstrnt$

653. Theories. A theory is associated with each function symbol. These theories are indexed by naturals between 0 and 8 according to the following table

0 Theory of uninterpreted function symbols. 1 Linear arithmetic theory. 2 Product theory. 3 Bitvector theory. 4 Coproducts. 5 Power products. 6 Theory of function abstraction and application. 7 Array theory. 8 Theory of bitvector interpretation(s).

type $th = int$

val th_to_string : $th \rightarrow string$

654. Function symbols. These are partitioned into uninterpreted function symbols and function symbols interpreted in one of the builtin theories. For each interpreted function symbol there is a recognizer function is_xxx. Some values of type sym represent families of function symbols. The corresponding indices can be obtained using the destructor d_xxx functions (only after checking that is_xxx holds.

type sym

val $sym\_is\_uninterp$: $sym \rightarrow bool$
val sym_is_interp : $th \rightarrow sym \rightarrow bool$

655. sym_eq tests for equality of two function symbols.

val sym_eq : $sym \rightarrow sym \rightarrow bool$

656. sym_cmp provides a total ordering on function symbols. It returns a negative integer if $s < t$, 0 if s is equal to t, and a positive number if $s > t$.

val sym_cmp : $sym \rightarrow sym \rightarrow int$

657. Arithmetic function symbols are either numerals, addition, or linear multiplication.

val sym_is_num : $sym \rightarrow bool$
val sym_d_num : $sym \rightarrow q$

val sym_is_add : $sym \rightarrow bool$

val sym_is_multq : $sym \rightarrow bool$
val sym_d_multq : $sym \rightarrow q$

Symbols interpreted in the theory of products.

val sym_is_tuple : $sym \rightarrow bool$

val sym_is_proj : $sym \rightarrow bool$
val sym_d_proj : $sym \rightarrow int \times int$

658. Symbols interpreted in the theory of coproducts.

val sym_is_inl : $sym \rightarrow bool$
val sym_is_inr : $sym \rightarrow bool$
val sym_is_outl : $sym \rightarrow bool$
val sym_is_outr : $sym \rightarrow bool$

659. Symbols in the bitvector theory.

val $sym\_is\_bv\_const$: $sym \rightarrow bool$

val $sym\_is\_bv\_conc$: $sym \rightarrow bool$
val $sym\_d\_bv\_conc$: $sym \rightarrow int \times int$

val $sym\_is\_bv\_sub$: $sym \rightarrow bool$
val $sym\_d\_bv\_sub$: $sym \rightarrow int \times int \times int$

val $sym\_is\_bv\_bitwise$: $sym \rightarrow bool$
val $sym\_d\_bv\_bitwise$: $sym \rightarrow int$

660. Symbols from the theory of power products.

val sym_is_mult : $sym \rightarrow bool$

val sym_is_expt : $sym \rightarrow bool$

661. Symbols from the theory of function abstraction and application.

val sym_is_apply : $sym \rightarrow bool$
val sym_d_apply : $sym \rightarrow cnstrnt\ option$

val sym_is_abs : $sym \rightarrow bool$

662. Symbols from the theory of arrays.

val sym_is_select : $sym \rightarrow bool$

val sym_is_update : $sym \rightarrow bool$

663. Symbols from the theory of arithmetic interpretations of bitvectors.

val $sym\_is\_unsigned$: $sym \rightarrow bool$

664. Terms. Terms are either variables, application of uninterpreted functions, or interpreted constants and operators drawn from a combination of theories..

term_of_string parses a string according to the grammar for the nonterminal *termeof* in module *Parser* (see its specification in file parser.*mly*) and builds a corresponding term. Similary, *term_input* builds a term by reading from an input channel.

term_output out a prints term *a* on the output channel *out*, and *term_pp a* is equivalent to *term_output stdout a*.

Terms are build using constructors, whose names are all of the form *mk_xxx*. For each constructor *mk_xxx* there is a corresponding recognizer *is_xxx* which reduces to true if its argument term has been built with the constructor *mk_xxx*. Moreover, for each constructor *mk_xxx*} above there is a corresponding desctructor *d_xxx* for analyzing the components of such a constructor term.

type *term*

val $term\_of\_string$: $string \rightarrow term$
val $term\_to\_string$: $term \rightarrow string$
val $term\_input$: $inchannel \rightarrow term$
val $term\_output$: $outchannel \rightarrow term \rightarrow unit$
val $term\_pp$: $term \rightarrow unit$

665. Equality and Comparison. Comparison *cmp a b* returns either -1, 0, or 1 depending on whether *a* is less than *b*, the arguments are equal, or *a* is greater than *b* according to the builtin term ordering (see *Term*.(<<<)). *term_eq a b* is true iff if *term_cmp a b* returns 0.

val $term\_eq$: $term \rightarrow term \rightarrow bool$
val $term\_cmp$: $term \rightarrow term \rightarrow int$

666. Given a string *s*, *term_mk_var s* constructs a variable with name *s* and *term_mk_uninterp s al* constructs an application of an uninterpreted function symbol *s* to a list of argument terms. If *s* is any of the builtin function symbols specified in module *Builtin*, the builtin simplifications are applied to this application.

val $term\_mk\_var$: $string \rightarrow term$

val $term\_mk\_uninterp$: $string \rightarrow term\ list \rightarrow term$

667. Arithmetic terms include rational constants built from *term_mk_num q*, linear multiplication *term_mk_multq q a*, addition *term_mk_add a b* of two terms, n-ary addition *term_mk_addl al* of a list of terms *al*, subtraction *term_mk_sub a b* of term *b* from term *a*, negation *term_mk_unary_minus a*, multiplication *term_mk_mult a b*, and exponentiation *term_mk_expt n a*. These constructors build up arithmetic terms in a canonical form as defined in module *Arith*. *term_is_arith a* holds iff the toplevel function symbol of *a* is any of the function symbols interpreted in the theory of arithmetic.

val $term\_mk\_num : q \to term$
val $term\_mk\_multq : q \to term \to term$
val $term\_mk\_add : term \to term \to term$
val $term\_mk\_addl : term\ list \to term$
val $term\_mk\_sub : term \to term \to term$
val $term\_mk\_unary\_minus : term \to term$

val $term\_is\_arith : term \to bool$

668. Tuples are built using the *mk_tuple* constructor, and projection of the *i*-th component of a tuple *t* of length *n* is realized using *mk_proj i n t*.

val $term\_mk\_tuple : term\ list \to term$
val $term\_mk\_proj : int \to int \to term \to term$

669. Boolean constants.

val $term\_mk\_true : unit \to term$
val $term\_mk\_false : unit \to term$

val $term\_is\_true : term \to bool$
val $term\_is\_false : term \to bool$

670. Bitvectors

val $term\_mk\_bvconst : string \to term$
val $term\_mk\_bvsub : (int \times int \times int) \to term \to term$
val $term\_mk\_bvconc : int \times int \to term \to term \to term$
val $term\_mk\_bwite : int \to term \times term \times term \to term$
val $term\_mk\_bwand : int \to term \to term \to term$
val $term\_mk\_bwor : int \to term \to term \to term$
val $term\_mk\_bwnot : int \to term \to term$

671. Coproducts.

val $term\_mk\_inj : int \to term \to term$
val $term\_mk\_out : int \to term \to term$

672. Set of terms.

type *terms*

673. Atoms. *atom_mk_true*() is the trivially true atom, *atom_mk_false*() is the trivially false atom, and given terms *a*, *b*, the constructor *mk_equal a b* constructs an equality constraint, *mk_diseq a b* a disequality constraint, and *atom_mk_in a c* constructs a membership constraint for *a* and a arithmetic constraint *c*. Atoms are printed to *stdout* using *atom_pp*.

type *atom*

val $atom\_pp$: $atom \rightarrow unit$

val $atom\_of\_string$: $string \rightarrow atom$
val $atom\_to\_string$: $atom \rightarrow string$

val $atom\_mk\_equal$: $term \rightarrow term \rightarrow atom$
val $atom\_mk\_diseq$: $term \rightarrow term \rightarrow atom$
val $atom\_mk\_in$: $cnstrnt \rightarrow term \rightarrow atom$
val $atom\_mk\_true$: $unit \rightarrow atom$
val $atom\_mk\_false$: $unit \rightarrow atom$

674. Derived atomic constraints. $atom\_mk\_int\ t$ restricts the domain of interpretations of term t to the integers. Similarly, $atom\_mk\_real$ restricts its argument to the real numbers. $atom\_mk\_lt\ a\ b$ generates the constraint `'a'` $<$ `'b'`, $atom\_mk\_le\ a\ b$ yields `'a'` $\leq$ `'b'`, $atom\_mk\_gt\ a\ b$ yields `'a'` $>$ `'b'`, and $atom\_mk\_ge\ a\ b$ yields `'a'` $\geq$ `'b'`.

val $atom\_mk\_real$: $term \rightarrow atom$
val $atom\_mk\_int$: $term \rightarrow atom$
val $atom\_mk\_nonint$: $term \rightarrow atom$

val $atom\_mk\_lt$: $term \rightarrow term \rightarrow atom$
val $atom\_mk\_le$: $term \rightarrow term \rightarrow atom$
val $atom\_mk\_gt$: $term \rightarrow term \rightarrow atom$
val $atom\_mk\_ge$: $term \rightarrow term \rightarrow atom$

675. Solution sets.

type *solution*

val $solution\_apply$: $solution \rightarrow term \rightarrow term$

val $solution\_find$: $solution \rightarrow term \rightarrow term$

val $solution\_inv$: $solution \rightarrow term \rightarrow term$

val $solution\_mem$: $solution \rightarrow term \rightarrow bool$

val $solution\_occurs$: $solution \rightarrow term \rightarrow bool$

val $solution\_use$: $solution \rightarrow term \rightarrow terms$

val $solution\_is\_empty$: $solution \rightarrow bool$

676. Logical context. An element of type *state* is a logical context with $state\_empty$ the empty context. $state\_eq\ s1\ s2$ is a constant-time predicate for testing for identity of two states. Thus, whenever this predicate holds its arguments states are equivalent, but not necessarily the other way round. Logical contexts are printed using $state\_pp$. The set of atoms in a context s are obtained with $state\_ctxt\_of\ s$.

type *context*

val $context\_eq$: $context \rightarrow context \rightarrow bool$

val $context\_empty$: $unit \rightarrow context$

val $context\_ctxt\_of$: $context \rightarrow atom\ list$

val $context\_u\_of$: $context \rightarrow solution$

val $context\_a\_of$: $context \rightarrow solution$

val $context\_t\_of$: $context \rightarrow solution$

val $context\_bv\_of$: $context \rightarrow solution$

val $context\_pp$: $context \rightarrow unit$

val $context\_ctxt\_pp$: $context \rightarrow unit$

677. Builtin simplifying constructors.

val $term\_mk\_unsigned$: $term \rightarrow term$

val $term\_mk\_update$: $term \rightarrow term \rightarrow term \rightarrow term$
val $term\_mk\_select$: $term \rightarrow term \rightarrow term$

val $term\_mk\_div$: $term \rightarrow term \rightarrow term$

val $term\_mk\_mult$: $term \rightarrow term \rightarrow term$
val $term\_mk\_multl$: $term\ list \rightarrow term$
val $term\_mk\_expt$: $int \rightarrow term \rightarrow term$
(∗ $term\_mk\_expt\ n\ x$ represent $x\hat{}n$. ∗)

val $term\_mk\_apply$: $term \rightarrow term\ list \rightarrow term$

val $term\_mk\_arith\_apply$: $cnstrnt \rightarrow term \rightarrow term\ list \rightarrow term$

678. The operation $process\ s\ a$ adds a new atom a to a logical context s. The codomain of this function is of type $status$, elements of which represent the three possible outcomes of processing a proposition: 1. the atom a is inconsistent in s, 2. it is valid, or, 3., it is satisfiable but not valid. In the third case, a modified state is obtained using the destructor $d\_consistent$.

type $status$

val $is\_consistent$: $status \rightarrow bool$
val $is\_redundant$: $status \rightarrow bool$
val $is\_inconsistent$: $status \rightarrow bool$

val $d\_consistent$: $status \rightarrow context$

val $process$: $context \rightarrow atom \rightarrow status$

679. Suggesting finite case split.

val $split$: $context \rightarrow atom\ list$

680. Canonization. Given a logical context s and an atom a, $can\ s\ a$ computes a semi-canonical form of a in s, that is, if a holds in s it returns $Atom.True$, if the negation of a

holds in s then it returns *Atom.False*, and, otherwise, an equivalent normalized atom built up only from variables is returned. The returned logical state might contain fresh variables.

val can : $context \rightarrow atom \rightarrow atom$

681. Given a logical context s and a term a, *cnstrnt* s a computes the best possible arithmetic constraint for a in s using constraint informatin in s and abstraction interval interpretation. If no such constraint can be deduced, *None* is returned.

val $cnstrnt$: $context \rightarrow term \rightarrow cnstrnt\ option$

682. An imperative state *istate* does not only include a logical context of type *state* but also a symbol table and input and output channels. A global *istate* variable is manipulated and destructively updated by commands.

683. Initialization. *init* n sets the verbose level to n. The higher the verbose level, the more trace information is printed to *stderr* (see below). There are no trace messages for $n = 0$. In addition, initialization makes the system to raise the *Sys.Break* exception upon user interrupt $\hat{}C\hat{}C$. The *init* function should be called before using any other function in this API.

val $init$: $int \times bool \times string \times inchannel \times outchannel \rightarrow unit$

684. *cmd_eval* reads a command from the current input channel according to the grammar for the nonterminal *commandeof* in module *Parser* (see its specification in file parser.*mly*, the current internal *istate* accordingly, and outputs the result to the current output channel.

val cmd_rep : $unit \rightarrow unit$

685. Sleeping for a number of seconds.

val $sleep$: $int \rightarrow unit$

686. Lists.

val is_nil : $\alpha\ list \rightarrow bool$
val $cons$: $\alpha \rightarrow \alpha\ list \rightarrow \alpha\ list$
val $head$: $\alpha\ list \rightarrow \alpha$
val $tail$: $\alpha\ list \rightarrow \alpha\ list$

687. Pairs. *pair* a b builds a pair (a, b) and *fst* (*pair* a b) returns a and *snd* (*pair* b a) returns b.

val $pair$: $\alpha \rightarrow \beta \rightarrow \alpha \times \beta$
val fst : $\alpha \times \beta \rightarrow \alpha$
val snd : $\alpha \times \beta \rightarrow \beta$

688. Triples. Accessors for triples (a, b, c).

val $triple$: $\alpha \rightarrow \beta \rightarrow \gamma \rightarrow \alpha \times \beta \times \gamma$
val fst_of_triple : $\alpha \times \beta \times \gamma \rightarrow \alpha$

val $snd\_of\_triple \ : \ \alpha \times \beta \times \gamma \rightarrow \beta$
val $third\_of\_triple \ : \ \alpha \times \beta \times \gamma \rightarrow \gamma$

689. Quadruples. Accessors for quadruples (a, b, c, d).

val $fst\_of\_quadruple \ : \ \alpha \times \beta \times \gamma \times \delta \rightarrow \alpha$
val $snd\_of\_quadruple \ : \ \alpha \times \beta \times \gamma \times \delta \rightarrow \beta$
val $third\_of\_quadruple \ : \ \alpha \times \beta \times \gamma \times \delta \rightarrow \gamma$
val $fourth\_of\_quadruple \ : \ \alpha \times \beta \times \gamma \times \delta \rightarrow \delta$

690. Options. An element of type α *option* either satisfies the recognizer *is_some* or *is_none*. In case, *is_some* holds, a value of type α can be obtained by *value_of*.

val $is\_some \ : \ \alpha \ option \rightarrow \ bool$
val $is\_none \ : \ \alpha \ option \rightarrow \ bool$

val $value\_of \ : \ \alpha \ option \rightarrow \ \alpha$

Module Ics.ml

691. let *init* (*n*, *pp*, *eot*, *inch*, *outch*) =

```
  Istate.initialize pp eot inch outch;
  if n = 0 then
    Sys.catch_break true (*s raise Sys.Break exception upon *)
                         (*s user interrupt. *)

let set_maxloops n =
  Rule.maxclose := n
let _ = Callback.register "set_maxloops" set_maxloops

let do_at_exit () = Tools.do_at_exit ()
let _ = Callback.register "do_at_exit" do_at_exit

let _ = Callback.register "init" init
```

692. Channels.

```
type inchannel = in_channel
type outchannel = Format.formatter

let channel_stdin () = Pervasives.stdin
let _ = Callback.register "channel_stdin" channel_stdin

let channel_stdout () = Format.std_formatter
let _ = Callback.register "channel_stdout" channel_stdout

let channel_stderr () = Format.err_formatter
let _ = Callback.register "channel_stderr" channel_stderr

let inchannel_of_string = Pervasives.open_in
```

```
let _ = Callback.register "inchannel_of_string" inchannel_of_string

let outchannel_of_string str =
  Format.formatter_of_out_channel (Pervasives.open_out str)
let _ = Callback.register "outchannel_of_string" outchannel_of_string
```

Names.

```
type name = Name.t

let name_of_string = Name.of_string
let _ = Callback.register "name_of_string" name_of_string

let name_to_string = Name.to_string
let _ = Callback.register "name_to_string" name_to_string

let name_eq = Name.eq
let _ = Callback.register "name_eq" name_eq
```

693. Constrains.

```
type cnstrnt = Cnstrnt.t

let cnstrnt_of_string s =
  let lb = Lexing.from_string s in
  Parser.cnstrnteof Lexer.token lb
let _ = Callback.register "cnstrnt_of_string" cnstrnt_of_string

let cnstrnt_input ch =
 let lb = Lexing.from_channel ch in
  Parser.cnstrnteof Lexer.token lb
let _ = Callback.register "cnstrnt_input" cnstrnt_input

let cnstrnt_output = Cnstrnt.pp
let _ = Callback.register "cnstrnt_output" cnstrnt_output

let cnstrnt_pp = Cnstrnt.pp Format.std_formatter
let _ = Callback.register "cnstrnt_pp" cnstrnt_pp

let cnstrnt_mk_int () = Cnstrnt.mk_int
let _ = Callback.register "cnstrnt_mk_int" cnstrnt_mk_int

let cnstrnt_mk_nonint () = Cnstrnt.mk_nonint
let _ = Callback.register "cnstrnt_mk_nonint" cnstrnt_mk_nonint

let cnstrnt_mk_nat () = Cnstrnt.mk_nat
let _ = Callback.register "cnstrnt_mk_nat" cnstrnt_mk_nat

let cnstrnt_mk_singleton = Cnstrnt.mk_singleton
let _ = Callback.register "cnstrnt_mk_singleton" cnstrnt_mk_singleton

let cnstrnt_mk_diseq = Cnstrnt.mk_diseq
let _ = Callback.register "cnstrnt_mk_diseq" cnstrnt_mk_diseq

let cnstrnt_mk_oo = Cnstrnt.mk_oo Dom.Real
```

let _ = *Callback.register* "cnstrnt_mk_oo" *cnstrnt_mk_oo*

let *cnstrnt_mk_oc* = *Cnstrnt.mk_oc Dom.Real*
let _ = *Callback.register* "cnstrnt_mk_oc" *cnstrnt_mk_oc*

let *cnstrnt_mk_co* = *Cnstrnt.mk_co Dom.Real*
let _ = *Callback.register* "cnstrnt_mk_co" *cnstrnt_mk_co*

let *cnstrnt_mk_cc* = *Cnstrnt.mk_cc Dom.Real*
let _ = *Callback.register* "cnstrnt_mk_cc" *cnstrnt_mk_cc*

let *cnstrnt_mk_lt* = *Cnstrnt.mk_lt Dom.Real*
let _ = *Callback.register* "cnstrnt_mk_lt" *cnstrnt_mk_lt*

let *cnstrnt_mk_le* = *Cnstrnt.mk_le Dom.Real*
let _ = *Callback.register* "cnstrnt_mk_le" *cnstrnt_mk_le*

let *cnstrnt_mk_gt* = *Cnstrnt.mk_gt Dom.Real*
let _ = *Callback.register* "cnstrnt_mk_gt" *cnstrnt_mk_gt*

let *cnstrnt_mk_ge* = *Cnstrnt.mk_ge Dom.Real*
let _ = *Callback.register* "cnstrnt_mk_ge" *cnstrnt_mk_ge*

let *cnstrnt_inter* = *Cnstrnt.inter*
let _ = *Callback.register* "cnstrnt_inter" *cnstrnt_inter*

let *cnstrnt_add* = *Cnstrnt.add*
let _ = *Callback.register* "cnstrnt_add" *cnstrnt_add*

let *cnstrnt_multq* = *Cnstrnt.multq*
let _ = *Callback.register* "cnstrnt_multq" *cnstrnt_multq*

let *cnstrnt_mult* = *Cnstrnt.mult*
let _ = *Callback.register* "cnstrnt_mult" *cnstrnt_mult*

let *cnstrnt_div* = *Cnstrnt.div*
let _ = *Callback.register* "cnstrnt_div" *cnstrnt_div*

694. Theories.

type *th* = *int*

let *th_to_string n* = *Th.to_string* (*Th.of_int n*)
let _ = *Callback.register* "th_to_string" *th_to_string*

695. Function symbols. These are partitioned into uninterpreted function symbols and function symbols interpreted in one of the builtin theories. For each interpreted function symbol there is a recognizer function *is_xxx*. Some values of type *sym* represent families of function symbols. The corresponding indices can be obtained using the destructor *d_xxx* functions (only after checking that *is_xxx* holds.

open *Sym*

type *sym* = *Sym.t*

```
let sym_is_uninterp = function Uninterp _ → true | _ → false
let _ = Callback.register "sym_is_uninterp" sym_is_uninterp

let sym_is_interp n sym =
  Th.eq (Th.of_int n) (Th.of_sym sym)
let _ = Callback.register "sym_is_interp" sym_is_interp

let sym_eq = Sym.eq
let _ = Callback.register "sym_eq" sym_eq

let sym_cmp = Sym.cmp
let _ = Callback.register "sym_cmp" sym_cmp

let sym_is_num = function Arith(Num _) → true | _ → false
let _ = Callback.register "sym_is_num" sym_is_num

let sym_d_num = function Arith(Num(q)) → q | _ → assert false
let _ = Callback.register "sym_d_num" sym_d_num

let sym_is_multq = function Arith(Multq _) → true | _ → false
let _ = Callback.register "sym_is_multq" sym_is_multq

let sym_d_multq = function Arith(Multq(q)) → q | _ → assert false
let _ = Callback.register "sym_d_multq" sym_d_multq

let sym_is_add = function Arith(Add) → true | _ → false
let _ = Callback.register "sym_is_add" sym_is_add

let sym_is_tuple = function Product(Tuple) → true | _ → false
let _ = Callback.register "sym_is_tuple" sym_is_tuple

let sym_is_proj = function Product(Proj _) → true | _ → false
let _ = Callback.register "sym_is_proj" sym_is_proj

let sym_d_proj = function Product(Proj(i, n)) → (i, n) | _ → assert false
let _ = Callback.register "sym_d_proj" sym_d_proj

let sym_is_inl = function Coproduct(InL) → true | _ → false
let _ = Callback.register "sym_is_inl" sym_is_inl

let sym_is_inr = function Coproduct(InR) → true | _ → false
let _ = Callback.register "sym_is_inr" sym_is_inr

let sym_is_outl = function Coproduct(OutL) → true | _ → false
let _ = Callback.register "sym_is_outl" sym_is_outl

let sym_is_outr = function Coproduct(OutR) → true | _ → false
let _ = Callback.register "sym_is_outr" sym_is_outr

let sym_is_bv_const = function Bv(Const _) → true | _ → false
let _ = Callback.register "sym_is_bv_const" sym_is_bv_const

let sym_is_bv_conc = function Bv(Conc _) → true | _ → false
let _ = Callback.register "sym_is_bv_conc" sym_is_bv_conc
```

```
let sym_d_bv_conc = function Bv(Conc(n, m)) → (n, m) | _ → assert false
let _ = Callback.register "sym_d_bv_conc" sym_d_bv_conc

let sym_is_bv_sub = function Bv(Sub _) → true | _ → false
let _ = Callback.register "sym_is_bv_sub" sym_is_bv_sub

let sym_d_bv_sub = function Bv(Sub(n, i, j)) → (n, i, j) | _ → assert false
let _ = Callback.register "sym_d_bv_sub" sym_d_bv_sub

let sym_is_bv_bitwise = function Bv(Bitwise _) → true | _ → false
let _ = Callback.register "sym_is_bv_bitwise" sym_is_bv_bitwise

let sym_d_bv_bitwise = function Bv(Bitwise(n)) → n | _ → assert false
let _ = Callback.register "sym_d_bv_bitwise" sym_d_bv_bitwise

let sym_is_mult = function Pp(Mult) → true | _ → false
let _ = Callback.register "sym_is_mult" sym_is_mult

let sym_is_expt = function Pp(Mult) → true | _ → false
let _ = Callback.register "sym_is_expt" sym_is_expt

let sym_is_apply = function Fun(Apply _) → true | _ → false
let _ = Callback.register "sym_is_apply" sym_is_apply

let sym_d_apply = function Fun(Apply(i)) → i | _ → assert false
let _ = Callback.register "sym_d_apply" sym_d_apply

let sym_is_abs = function Fun(Abs) → true | _ → false
let _ = Callback.register "sym_is_abs" sym_is_abs

let sym_is_select = function Arrays(Select) → true | _ → false
let _ = Callback.register "sym_is_select" sym_is_select

let sym_is_update = function Arrays(Update) → true | _ → false
let _ = Callback.register "sym_is_update" sym_is_update

let sym_is_unsigned = function Bvarith(Unsigned) → true | _ → false
let _ = Callback.register "sym_is_unsigned" sym_is_unsigned
```

696. Terms ar either variables, uninterpreted applications, or interpreted applications including boolean terms.

```
type term = Term.t

let term_of_string s =
  let lb = Lexing.from_string s in
  Parser.termeof Lexer.token lb
let _ = Callback.register "term_of_string" term_of_string

let term_to_string = Pretty.to_string Term.pp
let _ = Callback.register "term_to_string" term_to_string

let term_input ch =
  let lb = Lexing.from_channel ch in
```

```
  Parser.termeof Lexer.token lb
let _ = Callback.register "term_input" term_input

let term_output = Term.pp
let _ = Callback.register "term_output" term_output

let term_pp a = Term.pp Format.std_formatter a; Format.print_flush ()
let _ = Callback.register "term_pp" term_pp
```

697. Construct a variable.

```
let term_mk_var str =
  let x = Name.of_string str in
  Term.mk_var x
let _ = Callback.register "term_mk_var" term_mk_var
```

698. Uninterpred function application and function update.

```
let term_mk_uninterp x l =
  let f = Sym.Uninterp(Name.of_string x) in
  App.sigma f l
let _ = Callback.register "term_mk_uninterp" term_mk_uninterp
```

699. Constructing arithmetic expressions.

```
let term_mk_num = Arith.mk_num
let _ = Callback.register "term_mk_num" term_mk_num

let term_mk_multq q = Arith.mk_multq q
let _ = Callback.register "term_mk_multq" term_mk_multq

let term_mk_add = Arith.mk_add
let _ = Callback.register "term_mk_add" term_mk_add

let term_mk_addl = Arith.mk_addl
let _ = Callback.register "term_mk_addl" term_mk_addl

let term_mk_sub = Arith.mk_sub
let _ = Callback.register "term_mk_sub" term_mk_sub

let term_mk_unary_minus = Arith.mk_neg
let _ = Callback.register "term_mk_unary_minus" term_mk_unary_minus

let term_is_arith = Arith.is_interp
let _ = Callback.register "iterm_s_arith" term_is_arith
```

700. Constructing tuples and projections.

```
let term_mk_tuple = Tuple.mk_tuple
let _ = Callback.register "term_mk_tuple" term_mk_tuple

let term_mk_proj i j = Tuple.mk_proj i j
```

```
let _ = Callback.register "term_mk_proj" term_mk_proj
```

701. Bitvector terms.

```
let term_mk_bvconst s = Bitvector.mk_const (Bitv.from_string s)
let _ = Callback.register "term_mk_bvconst" term_mk_bvconst

let term_mk_bvsub (n, i, j) = Bitvector.mk_sub n i j
let _ = Callback.register "term_mk_bvsub" term_mk_bvsub

let term_mk_bvconc (n, m) = Bitvector.mk_conc n m
let _ = Callback.register "term_mk_bvconc" term_mk_bvconc

let term_mk_bwite n (a, b, c) = Bitvector.mk_bitwise n a b c
let _ = Callback.register "term_mk_bwite" term_mk_bwite

let term_mk_bwand n a b =
  Bitvector.mk_bitwise n a b (Bitvector.mk_zero n)
let _ = Callback.register "term_mk_bwand" term_mk_bwand

let term_mk_bwor n a b =
  Bitvector.mk_bitwise n a (Bitvector.mk_one n) b
let _ = Callback.register "term_mk_bwor" term_mk_bwor

let term_mk_bwnot n a =
  Bitvector.mk_bitwise n a (Bitvector.mk_zero n) (Bitvector.mk_one n)
let _ = Callback.register "term_mk_bwnot" term_mk_bwnot
```

702. Boolean terms.

```
let term_mk_true = Boolean.mk_true
let _ = Callback.register "term_mk_true" term_mk_true

let term_mk_false = Boolean.mk_false
let _ = Callback.register "term_mk_false" term_mk_false
```

703. Coproducts.

```
let term_mk_inj = Coproduct.mk_inj
let _ = Callback.register "term_mk_inj" term_mk_inj

let term_mk_out = Coproduct.mk_out
let _ = Callback.register "term_mk_out" term_mk_out
```

704. Atoms.

```
type atom = Atom.t
type atoms = Atom.Set.t

let atom_pp a =
  Atom.pp Format.std_formatter a;
  Format.print_flush ()
let _ = Callback.register "atom_pp" atom_pp
```

```
let atom_of_string s =
  let lb = Lexing.from_string s in
  Parser.atomeof Lexer.token lb
let _ = Callback.register "atom_of_string" atom_of_string

let atom_to_string = Pretty.to_string Atom.pp
let _ = Callback.register "atom_to_string" atom_to_string

let atom_mk_equal a b =
  Atom.mk_equal (Fact.mk_equal a b (Some(Fact.Axiom)))
let _ = Callback.register "atom_mk_equal" atom_mk_equal

let atom_mk_diseq a b =
  Atom.mk_diseq (Fact.mk_diseq a b Fact.mk_axiom)
let _ = Callback.register "atom_mk_diseq" atom_mk_diseq

let atom_mk_in i a =
  Atom.mk_in (Fact.mk_cnstrnt a i Fact.mk_axiom)
let _ = Callback.register "atom_mk_in" atom_mk_in

let atom_mk_true = Atom.mk_true
let _ = Callback.register "atom_mk_true" atom_mk_true

let atom_mk_false = Atom.mk_false
let _ = Callback.register "atom_mk_false" atom_mk_false

let atom_mk_real a =
  Atom.mk_in (Fact.mk_cnstrnt a Cnstrnt.mk_real Fact.mk_axiom)
let _ = Callback.register "atom_mk_real" atom_mk_real

let atom_mk_int a =
  Atom.mk_in (Fact.mk_cnstrnt a Cnstrnt.mk_int Fact.mk_axiom)
let _ = Callback.register "atom_mk_int" atom_mk_int

let atom_mk_nonint a =
  Atom.mk_in (Fact.mk_cnstrnt a Cnstrnt.mk_nonint Fact.mk_axiom)
let _ = Callback.register "atom_mk_nonint" atom_mk_nonint

let atom_mk_lt a b =
  Atom.mk_in (Fact.mk_cnstrnt
                (Arith.mk_sub a b)
                (Cnstrnt.mk_neg Dom.Real)
                Fact.mk_axiom)
let _ = Callback.register "atom_mk_lt" atom_mk_lt

let atom_mk_le a b =
  Atom.mk_in (Fact.mk_cnstrnt
                (Arith.mk_sub a b)
                (Cnstrnt.mk_nonpos Dom.Real)
                Fact.mk_axiom)
let _ = Callback.register "atom_mk_le" atom_mk_le
```

```
let atom_mk_gt a b = atom_mk_lt b a
let _ = Callback.register "atom_mk_gt" atom_mk_gt

let atom_mk_ge a b = atom_mk_le b a
let _ = Callback.register "atom_mk_ge" atom_mk_ge

let term_is_true = Boolean.is_true
let _ = Callback.register "term_is_true" term_is_true

let term_is_false = Boolean.is_false
let _ = Callback.register "term_is_false" term_is_false
```

705. Nonlinear terms.

```
let term_mk_mult = Sig.mk_mult
let _ = Callback.register "term_mk_mult" term_mk_mult

let rec term_mk_multl = function
  | [] → Arith.mk_one
  | [a] → a
  | a :: b :: l → term_mk_multl (term_mk_mult a b :: l)
let _ = Callback.register "term_mk_multl" term_mk_multl

let term_mk_expt = Sig.mk_expt
let _ = Callback.register "term_mk_expt" term_mk_expt
```

706. Builtin applications.

```
let term_mk_unsigned = Bvarith.mk_unsigned
let _ = Callback.register "term_mk_unsigned" term_mk_unsigned

let term_mk_update = Arr.mk_update
let _ = Callback.register "term_mk_update" term_mk_update

let term_mk_select = Arr.mk_select
let _ = Callback.register "term_mk_select" term_mk_select

let term_mk_div = Sig.mk_div
let _ = Callback.register "term_mk_div" term_mk_div

let term_mk_apply =
  Apply.mk_apply (Context.sigma Context.empty) None
let _ = Callback.register "term_mk_apply" term_mk_apply

let term_mk_arith_apply c =
  Apply.mk_apply (Context.sigma Context.empty) (Some(c))
let _ = Callback.register "term_mk_arith_apply" term_mk_arith_apply
```

707. Set of terms.

```
type terms = Term.Set.t
```

Term map.

type α *map* = α *Term.Map.t*

708. Equalities.

let *term_eq* = *Term.eq*
let _ = *Callback.register* "term_eq" *term_eq*

let *term_cmp* = *Term.cmp*
let _ = *Callback.register* "term_cmp" *term_cmp*

709. Trace level.

type *trace_level* = *string*

let *trace_reset* = *Trace.reset*
let _ = *Callback.register* "trace_reset" *trace_reset*

let *trace_add* = *Trace.add*
let _ = *Callback.register* "trace_add" *trace_add*

let *trace_remove* = *Trace.add*
let _ = *Callback.register* "trace_remove" *trace_remove*

let *trace_get* = *Trace.get*
let _ = *Callback.register* "trace_get" *trace_get*

710. Solution sets.

type *solution* = *Solution.t*

let *solution_apply s x* = *Solution.apply s x*

let *solution_find s x* = *Solution.find s x*

let *solution_inv s b* = *Solution.inv s b*

let *solution_mem* = *Solution.mem*

let *solution_occurs* = *Solution.occurs*

let *solution_use* = *Solution.use*

let *solution_is_empty* = *Solution.is_empty*

711. States.

open *Process*

type *context* = *Context.t*

let *context_eq* = *Context.eq*
let _ = *Callback.register* "context_eq" *context_eq*

let *context_empty* () = *Context.empty*
let _ = *Callback.register* "context_empty" *context_empty*

let *context_ctxt_of s* = (*Atom.Set.elements* (*Context.ctxt_of s*))

```
let _ = Callback.register "context_ctxt_of" context_ctxt_of

let context_u_of s = Context.eqs_of s Th.u
let _ = Callback.register "context_u_of" context_u_of

let context_a_of s = Context.eqs_of s Th.la
let _ = Callback.register "context_a_of" context_a_of

let context_t_of s = Context.eqs_of s Th.p
let _ = Callback.register "context_t_of" context_t_of

let context_bv_of s = Context.eqs_of s Th.bv
let _ = Callback.register "context_bv_of" context_bv_of

let context_pp s = Context.pp Format.std_formatter s; Format.print_flush()
let _ = Callback.register "context_pp" context_pp

let context_ctxt_pp s =
  let al = (Atom.Set.elements (Context.ctxt_of s)) in
  let fmt = Format.std_formatter in
    List.iter
      (fun a →
          Pretty.string fmt "\nassert␣";
          Atom.pp fmt a;
          Pretty.string fmt "␣.")
      al;
    Format.print_flush()
let _ = Callback.register "context_ctxt_pp" context_ctxt_pp
```

712. Processing of new equalities.

```
type status = Context.t Process.status

let is_consistent r =
  (match r with
     | Process.Ok _ → true
     | _ → false)
let _ = Callback.register "is_consistent" is_consistent

let is_redundant r = (r = Process.Valid)
let _ = Callback.register "is_redundant" is_redundant

let is_inconsistent r =
   (r = Process.Inconsistent)
let _ = Callback.register "is_inconsistent" is_inconsistent

let d_consistent r =
  match r with
    | Process.Ok s → s
    | _ → (context_empty())
         (* failwith "Ics.d_consistent:␣fatal␣error" *)
```

```
let _ = Callback.register "d_consistent" d_consistent

let process s =
  Trace.func "api" "Process" Atom.pp (Process.pp Context.pp)
    (Process.atom s)
let _ = Callback.register "process" process

let split s =
  Atom.Set.elements (Context.split s)
let _ = Callback.register "split" split
```

713. Normalization functions

```
let can = Can.atom
let _ = Callback.register "can" can

let read_from_channel ch =
  Parser.commands Lexer.token (Lexing.from_channel ch)

let read_from_string str =
  Parser.commandseof Lexer.token (Lexing.from_string str)

let rec cmd_rep () =
  let inch = Istate.inchannel() in
  let outch = Istate.outchannel() in
  try
    cmd_output outch (read_from_channel inch);
  with
    | Invalid_argument str -> cmd_error outch str
    | Parsing.Parse_error -> cmd_error outch "Syntax"
    | End_of_file -> cmd_quit 0 outch;
    | Sys.Break -> cmd_quit 1 outch;
    | Failure "drop" -> raise (Failure "drop")
    | exc -> (cmd_error outch ("Exception␣" ^ (Printexc.to_string exc)); exit 2)

and cmd_output fmt result =
  (match result with
     | Result.Process(status) ->
         Process.pp Name.pp fmt status
     | Result.Unit() ->
         Format.fprintf fmt ":unit"
     | Result.Bool(true) ->
         Format.fprintf fmt ":true"
     | Result.Bool(false) ->
         Format.fprintf fmt ":false"
     | value ->
         Format.fprintf fmt ":val␣";
         Result.output fmt value);
  cmd_endmarker fmt
```

```
and cmd_error fmt str =
   Format.fprintf fmt ":error %s" str;
   Format.fprintf fmt "\n%s@?" (Istate.eot())

and cmd_quit n fmt =
  Format.fprintf fmt ":quit\n";
  cmd_endmarker fmt;
  exit n

and cmd_endmarker fmt =
   let eot = Istate.eot () in
   if eot = "" then
      Format.pp_print_flush fmt ()
   else
      begin
         Format.fprintf fmt "\n%s" eot;
         Format.pp_print_flush fmt ()
      end

let _ = Callback.register "cmd_rep" cmd_rep
```

714. Abstract sign interpretation.

```
let cnstrnt s a =
   try
      Some(Context.cnstrnt s a)
   with
         Not_found → None
```

715. Tools

```
let reset () = Tools.do_at_reset ()
let _ = Callback.register "reset" reset

let gc () = Gc.full_major ()
let _ = Callback.register "gc" gc

let flush = print_flush
let _ = Callback.register "flush" flush
```

716. Lists.

```
let is_nil = function [] → true | _ → false
let _ = Callback.register "is_nil" is_nil

let cons x l = x :: l
let _ = Callback.register "cons" cons

let head = List.hd
let _ = Callback.register "head" head
```

```
let tail = List.tl
let _ = Callback.register "tail" tail
```

717. Pairs.

```
let pair x y = (x, y)
let _ = Callback.register "pair" pair

let fst = fst
let _ = Callback.register "fst" fst

let snd = snd
let _ = Callback.register "snd" snd
```

718. Triples.

```
let triple x y z = (x, y, z)
let _ = Callback.register "triple" triple

let fst_of_triple = function (x, _, _) → x
let _ = Callback.register "fst_of_triple" fst_of_triple

let snd_of_triple = function (_, y, _) → y
let _ = Callback.register "snd_of_triple" snd_of_triple

let third_of_triple = function (_, _, z) → z
let _ = Callback.register "third_of_triple" third_of_triple
```

719. Quadruples.

```
let fst_of_quadruple = function (x1, _, _, _) → x1
let _ = Callback.register "fst_of_quadruple" fst_of_quadruple

let snd_of_quadruple = function (_, x2, _, _) → x2
let _ = Callback.register "snd_of_quadruple" snd_of_quadruple

let third_of_quadruple = function (_, _, x3, _) → x3
let _ = Callback.register "third_of_quadruple" third_of_quadruple

let fourth_of_quadruple = function (_, _, _, x4) → x4
let _ = Callback.register "fourth_of_quadruple" fourth_of_quadruple
```

720. Options.

```
let is_some = function
  | Some _ → true
  | None → false

let is_none = function
  | None → true
  | _ → false

let value_of = function
```

```
  | Some(x) → x
  | _ → assert false
let _ = Callback.register "is_some" is_some
let _ = Callback.register "is_none" is_none
let _ = Callback.register "value_of" value_of
```

721. Sleeping.

```
let sleep = Unix.sleep
let _ = Callback.register "sleep" sleep
```

722. Multi-precision arithmetic.

```
open Mpa

type q = Q.t

let ints_of_num q = (Z.to_string (Q.numerator q), Z.to_string (Q.denominator q))
let _ = Callback.register "ints_of_num" ints_of_num

let num_of_int = Q.of_int
let _ = Callback.register "num_of_int" num_of_int

let num_of_ints i1 i2 = Q.div (num_of_int i1) (num_of_int i2)
let _ = Callback.register "num_of_ints" num_of_ints

let string_of_num = Q.to_string
let _ = Callback.register "string_of_num" string_of_num

let num_of_string = Q.of_string
let _ = Callback.register "num_of_string" num_of_string
```

Acknowledgements. The algorithms and data structures underlying ICS have been developed by N. Shankar and Harald Rueß. The core ICS code is by Harald Rueß and Jean-Christophe Filliâtre, and the Lisp interface has been developed by Sam Owre, Harald Rueß, and Jean-Christophe Filliâtre. The GMP ocaml interface was originally written by David Monniaux, and adjusted for use with `ocaml 3.00` by Jean-Christophe Filliâtre. Leonardo de Moura wrote the simulator in Appendix 2.

This document has been automatically produced from the source code using the literate programming tool ocamlweb[2].

References

[Fil00] Jean-Christophe Filliâtre. Hash-consing in an ML framework. 2000. 4

[2]Objective Caml and ocamlweb are both freely available, respectively at http://caml.inria.fr and http://www.lri.fr/~filliatr/ocamlweb

[MR98] O. Möller and H. Rueß. Solving bit-vector equations. In G. Gopalakrishnan and Ph. Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *Lecture Notes in Computer Science*, pages 36–48, Palo Alto, CA, November 1998. Springer-Verlag. 3

[Oca] The Objective Caml language. http://caml.inria.fr/. 4

[OG98] Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, September 1998. 4

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag. 4

[RS01] Harald Rueß and N. Shankar. Deconstructing Shostak. Accepted for publication at LICS, 2001, available from http://www.csl.sri.com/~ruess/papers/LICS2001/index.html, 2001. 3

A Bakery Mutual Exclusion Protocol

The following program realizes a symbolic simulator for a simplified Bakery mutual exclusion protocol using the ICS interface to C.

```
/*
 * The contents of this file are subject to the ICS(TM) Community Research
 * License Version 1.0 (the ``License''); you may not use this file except in
 * compliance with the License. You may obtain a copy of the License at
 * http://www.icansolve.com/license.html.  Software distributed under the
 * License is distributed on an ``AS IS'' basis, WITHOUT WARRANTY OF ANY
 * KIND, either express or implied. See the License for the specific language
 * governing rights and limitations under the License.  The Licensed Software
 * is Copyright (c) SRI International 2001, 2002.  All rights reserved.
 * ``ICS'' is a trademark of SRI International, a California nonprofit public
 * benefit corporation.
 *
 * Author: Harald Ruess
 */

/*s Module [Parser]: parser for ICS syntactic categories. */

%{
  open Mpa
  open Tools
```

```
let out = Istate.outchannel

let pr str =  Format.fprintf (out()) str

let nl () = pr "\n"

let equal_width_of a b =
 match Istate.width_of a, Istate.width_of a with
   | Some(n), Some(m) when n = m -> n
   | Some(n), None -> n
   | None, Some(n) -> n
   | Some _, Some _ ->
       raise (Invalid_argument "Argument mismatch")
   | None, None ->
       raise (Invalid_argument (Term.to_string a ^ " not a bitvector."))


%}

%token DROP CAN ASSERT EXIT SAVE RESTORE REMOVE FORGET RESET SYMTAB SIG VALID UNSAT
%token TYPE SIGMA
%token SOLVE HELP DEF TOGGLE SET TRACE UNTRACE CMP FIND USE INV SOLUTION PARTITION
%token SHOW CNSTRNT SYNTAX COMMANDS SPLIT
%token DISEQ CTXT
%token EOF

%token ARITH TUPLE

%token <string> IDENT
%token <int> INTCONST
%token <Mpa.Q.t> RATCONST

%token IN
%token BOT INT NONINT REAL BV TOP
%token INF NEGINF
%token ALBRA ACLBRA CLBRA

%token LPAR RPAR LBRA RBRA LCUR RCUR UNDERSCORE KLAMMERAFFE
%token COLON COMMA DOT DDOT ASSIGN UNION TO ENDMARKER BACKSLASH

%token <string> BVCONST
%token <string * int> FRESH
%token <int> FREE
```

```
%token CONC SUB BWITE BWAND BWOR BWXOR BWIMP BWIFF BWNOT
%token BVCONC
%token EQUAL DISEQ
%token TRUE FALSE
%token PLUS MINUS TIMES DIVIDE EXPT
%token LESS GREATER LESSOREQUAL GREATEROREQUAL
%token UNSIGNED APPLY LAMBDA
%token WITH CONS CAR CDR NIL
%token INL INR OUTL OUTR
%token INJ OUT
%token HEAD TAIL LISTCONS
%token DISJ XOR IMPL BIIMPL CONJ NEG
%token PROJ

%right DISJ XOR IMPL
%left BIIMPL CONJ
%nonassoc EQUAL DISEQ LESS GREATER LESSOREQUAL GREATEROREQUAL
%left APPLY
%left UNION
%left MINUS PLUS
%left DIVIDE
%left TIMES
%right EXPT
%left LISTCONS
%right BVCONC
%right BWOR BWXOR BWIMP
%left BWAND BWIFF
%nonassoc TO
%nonassoc IN NOTIN
%nonassoc LCUR
%nonassoc LBRA
%nonassoc prec_unary

%type <Term.t> termeof
%type <Atom.t> atomeof
%type <Cnstrnt.t> cnstrnteof
%type <Result.t> commands
%type <Result.t> commandseof

%start termeof
%start atomeof
%start cnstrnteof
%start commands
%start commandseof
```

```
%%

termeof : term EOF          { $1 }
atomeof : atom EOF          { $1 }
cnstrnteof : cnstrnt EOF    { $1 }
commandseof : command EOF   { $1 }

commands : command DOT      { $1 }
| EOF                       { raise End_of_file }

rat:
  INTCONST  { Q.of_int $1 }
| RATCONST  { $1 }
;


name: IDENT             { Name.of_string $1 }

funsym:
  name                                    { Sym.Uninterp($1) }
| PLUS                                    { Sym.Arith(Sym.Add) }
| TIMES                                   { Sym.Pp(Sym.Mult) }
| EXPT LBRA INTCONST RBRA                 { Sym.Pp(Sym.Expt($3)) }
| TUPLE                                   { Sym.Product(Sym.Tuple) }
| UNSIGNED                                { Sym.Bvarith(Sym.Unsigned) }
| PROJ LBRA INTCONST COMMA INTCONST RBRA { Sym.Product(Sym.Proj($3, $5)) }
| CONS                                    { Sym.Product(Sym.Tuple)  }
| CAR                                     { Sym.Product(Sym.Proj(0, 2)) }
| CDR                                     { Sym.Product(Sym.Proj(1, 2)) }
| CONC LBRA INTCONST COMMA INTCONST RBRA              { Sym.Bv(Sym.Conc($3, $5)) }
| SUB LBRA INTCONST COMMA INTCONST COMMA INTCONST RBRA { Sym.Bv(Sym.Sub($3, $5, $7)) }
| BWITE LBRA INTCONST RBRA                            { Sym.Bv(Sym.Bitwise($3)) }
| APPLY range                             { Sym.Fun(Sym.Apply($2)) }
| LAMBDA                                  { Sym.Fun(Sym.Abs) }
;

range:                               { None }
| LBRA cnstrnt RBRA                  { Some($2) }
;

constsym:
  rat       { Sym.Arith(Sym.Num($1)) }
| TRUE      { Sym.Bv(Sym.Const(Bitv.from_string "1")) }
```

```
| FALSE     { Sym.Bv(Sym.Const(Bitv.from_string "0")) }
| BVCONST   { Sym.Bv(Sym.Const(Bitv.from_string $1)) }
;



term:
  var              { $1 }
| app              { $1 }
| LPAR term RPAR   { $2 }
| arith            { $1 }     /* infix/mixfix syntax */
| array            { $1 }
| bv               { $1 }
| coproduct        { $1 }
| boolean          { $1 }
| list             { $1 }
| apply            { $1 }
;

var:
  name  { try
    match Symtab.lookup $1 (Istate.symtab()) with
      | Symtab.Def(a) -> a
      | _ -> Term.mk_var $1
  with
      Not_found -> Term.mk_var $1 }
| FRESH  { let (x,k) = $1 in
   Term.mk_fresh_var (Name.of_string x) (Some(k)) }
| FREE   { Term.Var(Var.mk_free $1) }
;



app:
  funsym LPAR termlist RPAR     { Istate.sigma $1 (List.rev $3) }
| constsym                      { Istate.sigma $1 [] }

list:
  term LISTCONS term            { Coproduct.mk_inj 1 (Tuple.mk_tuple [$1; $3]) }
| HEAD LPAR term RPAR           { Tuple.mk_proj 0 2 (Coproduct.mk_out 1 $3) }
| TAIL LPAR term RPAR           { Tuple.mk_proj 1 2 (Coproduct.mk_out 1 $3) }
| NIL                           { Coproduct.mk_inj 0 (Tuple.mk_tuple []) }

apply:
  term APPLY term               { Apply.mk_apply
```

```
    (Context.sigma (Context.empty))
                                     None $1 [$3] }


arith:
| term PLUS term              { Arith.mk_add $1 $3 }
| term MINUS term             { Arith.mk_sub $1 $3 }
| MINUS term %prec prec_unary   { Arith.mk_neg $2 }
| term TIMES term             { Sig.mk_mult $1 $3 }
| term DIVIDE term            { Sig.mk_div $1 $3 }
| term EXPT INTCONST          { Sig.mk_expt $3 $1 }
;

coproduct:
  INL LPAR term RPAR                    { Coproduct.mk_inl $3 }
| INR LPAR term RPAR                    { Coproduct.mk_inr $3 }
| OUTL LPAR term RPAR                   { Coproduct.mk_outl $3 }
| OUTR LPAR term RPAR                   { Coproduct.mk_outr $3 }
| INJ LBRA INTCONST RBRA LPAR term RPAR { Coproduct.mk_inj $3 $6 }
| OUT LBRA INTCONST RBRA LPAR term RPAR { Coproduct.mk_out $3 $6 }


array:
  term LBRA term ASSIGN term RBRA { Arr.mk_update $1 $3 $5 }
| term LBRA term RBRA        { Arr.mk_select $1 $3 }
;


bv:
  term BVCONC term   { match Istate.width_of $1, Istate.width_of $3 with
  | Some(n), Some(m) ->
      if n < 0 then
raise (Invalid_argument ("Negative length of " ^ Term.to_string $1))
      else if m < 0 then
raise (Invalid_argument ("Negative length of " ^ Term.to_string $3))
      else
Bitvector.mk_conc n m $1 $3
  | Some _, _ ->
      raise (Invalid_argument (Term.to_string $3 ^ " not a bitvector."))
  | _ ->
      raise (Invalid_argument (Term.to_string $1 ^ " not a bitvector.")) }
| term LBRA INTCONST COLON INTCONST RBRA
                      { match Istate.width_of $1 with
  | Some(n) ->
```

```
      if n < 0 then
raise(Invalid_argument ("Negative length of " ^ Term.to_string $1))
      else if not(0 <= $3 && $3 <= $5 && $5 < n) then
raise(Invalid_argument ("Invalid extraction from " ^ Term.to_string $1))
      else
Bitvector.mk_sub n $3 $5 $1
  | None ->
      raise (Invalid_argument (Term.to_string $1 ^ " not a bitvector.")) }
| term BWAND term     { Bitvector.mk_bwconj (equal_width_of $1 $3) $1 $3 }
| term BWOR term      { Bitvector.mk_bwdisj (equal_width_of $1 $3) $1 $3 }
| term BWIMP term     { Bitvector.mk_bwimp (equal_width_of $1 $3) $1 $3 }
| term BWIFF term     { Bitvector.mk_bwiff (equal_width_of $1 $3) $1 $3 }
;

boolean:
| term CONJ term           { Boolean.mk_conj $1 $3 }
| term DISJ term           { Boolean.mk_disj $1 $3 }
| term XOR term            { Boolean.mk_xor $1 $3 }
| NEG term %prec prec_unary { Boolean.mk_neg $2 }
;


atom:
  term EQUAL term                  { Atom.mk_equal (Fact.mk_equal $1 $3 None)}
| term DISEQ term                  { Atom.mk_diseq (Fact.mk_diseq $1 $3 None) }
| term LESS term                   { Atom.mk_lt $1 $3 }
| term GREATER term                { Atom.mk_lt $3 $1 }
| term LESSOREQUAL term            { Atom.mk_le $1 $3 }
| term GREATEROREQUAL term         { Atom.mk_le $3 $1 }
| term IN cnstrnt                  { Atom.mk_in (Fact.mk_cnstrnt $1 $3 None) }
;


cnstrnt:
| interval optdiseqs { Cnstrnt.make ($1, $2) }
| name               { match Istate.type_of $1 with
 | Some(c) -> c
 | None ->
    let str = Name.to_string $1 in
    raise (Invalid_argument ("No type definition for " ^ str)) }
;

optdiseqs:                   { Cnstrnt.Diseqs.empty }
| BACKSLASH LCUR diseqs RCUR { $3 }
```

```
;

diseqs: rat          { Cnstrnt.Diseqs.singleton $1 }
| diseqs COMMA rat   { Cnstrnt.Diseqs.add $3 $1 }



interval:
  INT                                    { Interval.mk_int }
| REAL                                   { Interval.mk_real }
| NONINT                                 { Interval.mk_nonint }
| NONINT leftendpoint DDOT rightendpoint { Interval.make (Dom.Nonint, $2, $4) }
| INT leftendpoint DDOT rightendpoint    { Interval.make (Dom.Int, $2, $4) }
| REAL leftendpoint DDOT rightendpoint   { Interval.make (Dom.Real, $2, $4) }
| leftendpoint DDOT rightendpoint        { Interval.make (Dom.Real, $1, $3) }
;

leftendpoint:
  LPAR NEGINF     { Endpoint.neginf }
| LPAR rat        { Endpoint.strict $2 }
| LBRA rat        { Endpoint.nonstrict $2 }
;

rightendpoint:
  INF RPAR          { Endpoint.posinf }
| rat RPAR          { Endpoint.strict $1 }
| rat RBRA          { Endpoint.nonstrict $1 }
;

termlist:             { [] }
| term                { [$1] }
| termlist COMMA term { $3 :: $1 }
;

signature:
  BV LBRA INTCONST RBRA     { $3 }
;

command:
  CAN atom                  { Result.Atom(Istate.can $2) }
| CAN term                  { Result.Term(Istate.cant $2) }
| ASSERT optname atom       { Result.Process(Istate.process $2 $3) }
| DEF name ASSIGN term      { Result.Unit(Istate.def $2 $4) }
| SIG name COLON signature  { Result.Unit(Istate.sgn $2 $4) }
```

```
| TYPE name ASSIGN cnstrnt  { Result.Unit(Istate.typ $2 $4) }
| RESET                     { Result.Unit(Istate.reset ()) }
| SAVE name                 { Result.Name(Istate.save(Some($2))) }
| SAVE                      { Result.Name(Istate.save(None)) }
| RESTORE name              { Result.Unit(Istate.restore $2) }
| REMOVE name               { Result.Unit(Istate.remove $2) }
| FORGET                    { Result.Unit(Istate.forget()) }
| VALID optname atom        { Result.Bool(Istate.valid $2 $3) }
| UNSAT optname atom        { Result.Bool(Istate.unsat $2 $3) }
| EXIT                      { raise End_of_file }
| DROP                      { failwith "drop" }
| SYMTAB                    { Result.Symtab(Istate.symtab()) }
| SYMTAB name               { match Istate.entry_of $2 with
| Some(e) -> Result.Entry(e)
| None -> raise (Invalid_argument (Name.to_string $2 ^ "not in symbol table")) }
| CTXT optname              { Result.Atoms(Istate.ctxt_of $2) }
| SIGMA term                { Result.Term($2) }
| term CMP term             { Result.Int(Term.cmp $1 $3) }
| SHOW optname              { Result.Context(Istate.get_context $2) }
| FIND optname th term      { Result.Term(Istate.find $2 $3 $4) }
| INV optname th term       { try Result.Optterm(Some(Istate.inv $2 $3 $4))
      with Not_found -> Result.Optterm(None) }
| USE optname th term       { Result.Terms(Istate.use $2 $3 $4) }
| SOLUTION optname th       { Result.Solution(Istate.solution $2 $3) }
| CNSTRNT optname term      { match Istate.cnstrnt $2 $3 with
| Some(c) -> Result.Cnstrnt(Some(c))
| None -> Result.Cnstrnt(None) }
| DISEQ optname term        { Result.Terms(Istate.diseq $2 $3) }
| SPLIT optname             { Result.Atoms(Istate.split()) }
| SOLVE th term EQUAL term  { Result.Solution(Istate.solve $2 ($3, $5)) }
| TRACE identlist           { Result.Unit(List.iter Trace.add $2) }
| UNTRACE                   { Result.Unit(Trace.reset ()) }
| help                      { Result.Unit($1) }
;

identlist :
  IDENT                     { [$1] }
| identlist COMMA IDENT     { $3 :: $1 }


th: IDENT  { failwith "to do" } /* may raise [Invalid_argument]. */

help:
  HELP                      { Help.on_help () }
```

```
| HELP SYNTAX                   { Help.syntax () }
| HELP COMMANDS                 { Help.commands () }
;


optname:                        { None }
| KLAMMERAFFE name              { Some($2) }
;


%%
```