ICS Manual (Version 1.1)

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,
- linear arithmetic operators,
- tupling operators and tuple projection,
- coproducts
- S-expressions,
- array lookup and update,
- fixed-sized bitvectors,
- arithmetic interpretation of fixed-sized bitvectors.
- higher-order function application and abstraction.

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. 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 for free.

ICS is implemented in Ocaml [Oca], which offers satisfactory run-time performance, efficient garbage collection, and interfaces well with other languages such as C.

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.

The efficiency and scalability of ICS in processing formulas, the richness of its API, and its ability for fast context-switching 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

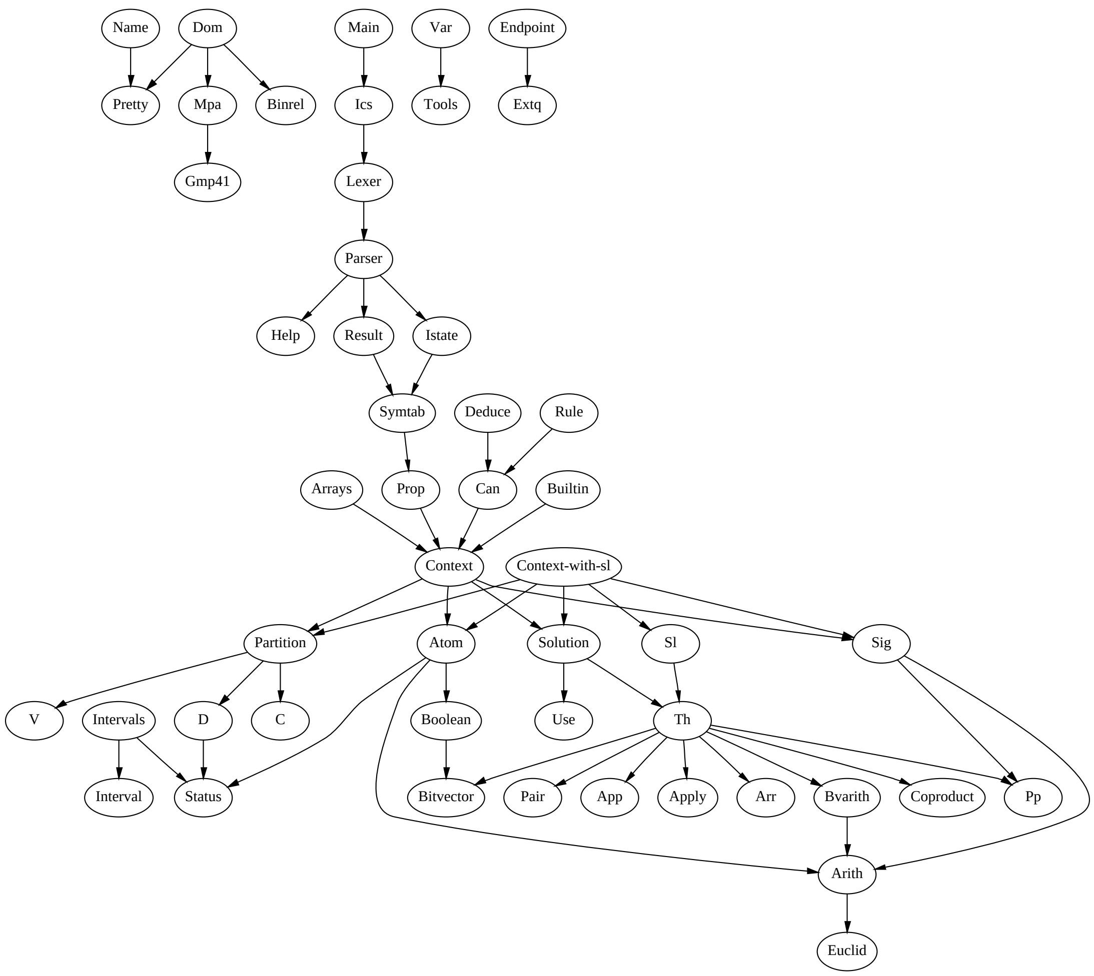
`http://www.icansolve.com/fm-license.pdf`

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`

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.pdf`	:	Noncommercial license.
`bin/`	:	Binaries
`configure`	:	Configuration script
`lib/`	:	Archives and shared object files
`README`	:	Short installation guide
`doc/`	:	Documentation files
`obj/`	:	Object files
`sat/`	:	Sources for Boolean SAT solver
`src/`	:	Source files for core ICS
`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:

- Ocaml version 3.04 or later. Freely available at `http://caml.inria.fr`.
- GNU MP version 4.1 or later. This package is freely available at `http://www.swox.com/gmp/`.

So far, we have only compiled ICS on Linux Redhat 7.3, but it should be possible to compile it on a wide range of machines and operating systems.

Installation.

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

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

   The optional `with-gmp` option is used to specify the path to a particular GMP library, and the `prefix` option specifies the path for installing ICS binaries (`prefix` defaults to `/usr/local/bin`.

2. Now, `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.

Notice that some C compilers on some operating systems such as `gcc` on `OS X` are not able to build dynamic libraries using the `-shared` option. In these cases it is necessary to edit the generated `Makefile` and disable the creation of `libics.so`.

3. The build directory is `./obj/$(ARCH)/`, and the generated binary and byte code are put in `./bin/$(ARCH)/`. The binaries are installed at the location specified by the `prefix` option to `configure` above using the following command.

```
> make install
```

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 ICS in action

ICS can either be used in batch, interactive, or in server mode. Here we demonstrate some of the capabilities of ICS using its interactive mode. ICS maintains a *state* which can be manipulated and queried by a series of command. Most importantly, the `assert` command extends the current logical state with a new fact. The following command, for example, adds an equality over terms built from the the variable `x` and the uninterpreted function symbol `f`.

```
ics> assert f(f(f(x))) = f(x).
:ok s1
```

It adds this fact to the current logical, which can be queried using the `ctxt` command.

```
ics> ctxt.
:val {f(f(f(x))) = f(x)}
```

In addition, the `assert` command generates a fresh name, here `s1`, for the extended state and keeps this association in a symbol table.

```
ics> symtab.
:val empty |-> state({})
        s1 |-> state({f(f(f(x))) = f(x)})
```

Now, asserting the equality `f(f(f(x))) = f(x)` to the current logical state yields `:valid`, since indeed this equality logically follows from the previously asserted equality using congruence closure.

```
ics> assert f(f(f(f(f(x))))) = f(x).
:valid
```

In this case, the current state is unchanged.

Validity of equalities is established in ICS using *canonization*. For example, canonizing the left-hand and the right-hand side of the equality above both yields the (internally generated) variable `v!1`.

```
ics> can f(x).
:val v!1
ics> can f(f(f(f(f(x))))).
:val v!1
```

Since the canonical forms of these terms are identical in the current context, the equality `f(f(f(x))) = f(x)` holds indeed.

Processing of new facts using `assert` is done by building up an internal representation, which can be queried using the `show` command. The current state after processing `f(f(f(x))) = f(x)`, for example, is obtained by introducing names for all subterms in this equality.

```
ics> show.
:val
v:[v!1:{v!1,  v!3}]
u:[v!1 = f(x);  v!2 = f(v!1);  v!3 = f(v!2)]
```

Thus, `f(f(f(x)))` equals `f(f(v!1))`, `f(v!2)`, `v!3`, and, finally, `v!1`. The `v` part of the state above represents the variable equality `v!1 = v!3` with `v!1` the canonical representative of the generated equivalence class, whereas the `u` part consists of equalites `x = f(...)` with `x` a variable, and `f(...)` a flat term with only variables as arguments. Notice that ICS does not keep the left-hand side of equations in `u` in canonical form. Also, equations in the `u` part are not necessarily in solved form, that is, an equation of the form `x = f(x)` may be added.

ICS supports also a number of interpreted theories in the combination with uninterpreted function symbols.

```
ics> reset.
:unit
ics> assert z = f(x - y).
:ok s1
```

Here, `x - y` is interpreted as the difference between `x` and `y` in the theory of *linear arithmetic*. Besides the variable equalities `v` and the set `u` of uninterpreted equalities, the resulting logical state also contains a set `a` of linear arithmetic equalities.

```
ics> show.
:val
v:[z:{z,  v!2}]
u:[v!2 = f(v!1)]
a:[v!1 = -1 * y +  x]
```

Since the term `f(x - y)` in the input equality contains both the uninterpreted function symbol `f` and the interpreted function symbol `-`, it is rewritten as `f(v!1)` with `v!1 = -y + x`, with `v!1` a fresh variable. In contrast to equalities in `u`, equality sets for interpreted theories are always in solved form, that is, a variable on the left-hand side does not occur on any right-hand side. Now, `x = z + y` is asserted to state `s1` by solving it for the largest—in some given ordering—variable `z`, and deducing that `v!1` is equal to `z`. Now, `v!1` is replace with `z` in right-hand sides of `u`, and, since the non-canonical `v!1` does not occur in any of the equality sets anymore, the variable equality between `v!1` and `z` can safely been forgotten.

```
ics> assert x = z + y.
:ok s2
ics> show.
:val
v:[z:{z,  v!2}]
u:[v!2 = f(z)]
a:[z = -1 * y +  x]
```

Asserting the disequality `-y <> -(x - f(f(z))` yields unsatisfiability.

```
ics> assert -y <> -(x - f(f(z))).
:unsat
```

That is, the conjunction of the facts in the current context with this disequality has been shown to be unsatisfiable. The current state is unchanged in this case. This inconsistency is detected by canonization, since the canonical forms of `-y` and `-(x - f(f(z)))` are identical in context `s2`.

```
ics> can -y.
:val -1 * y
ics> can -(x - f(f(z))).
:val -1 * y
```

Besides arithmetic, ICS currently includes other theories such as the theory of tuples and projections, arrays, coproducts, or bitvectors, and the combination of the theory of tuples and coproducts is used to describe abstract datatypes such as binary trees. The following shows an example for the combination of linear arithmetic, the theory arrays with function update `a[i:=x]` and lookup `a[i]`, and uninterpreted functions.

```
ics> reset.
:unit
ics> assert x+2=y.
:ok s1
ics> assert f(a[x:=3][y - 2]) = f(y - x +1).
:valid
```

The next example demonstrates the combination of linear arithmetic with S-expressions built from the pairing function `cons(.,.)` and the first and second projection `car(.)` and `cdr(.)`, and uninterpreted functions.

```
ics> reset.
:unit
ics> assert 2 * car(x) - 3 * cdr(x) = f(cdr(x)).
:ok s1
ics> assert f(cons(4 * car(x) - 2 * f(cdr(x)), y)) = f(cons(6 * cdr(x), y)).
:valid
```

Again, variables are introduced for abstracting terms and the state `s1` also contains an equality sets for the theory of products `p`.

```
ics> show.
:val
u:[v!3 = f(v!2)]
a:[v!3 = -3 * v!2 +  2 * v!1]
p:[v!1 = car(x);  v!2 = cdr(x)]
```

So far, we have only dealt with equalities and disequalities, but consstraints over inequalities with arithmetical operations appear in almost all verification conditions from simple sequential programs over reactive, real-time, and hybrid systems. It is crucial to tightly integrate equality and inequality reasoning in that equalities are propagated to all known inequalities, and the inequality reasoner generates all possible equalities. In ICS, we achieve such an efficient integration using slack variables to reduce problems about inequalities to equality reasoning and simple constraint propagation. For example, the equality `x <= y + 2` is reduced to the equality `x - y - 2 = k!1` with `k!1` a newly generated, non-negative slack variable. This equality is solved for the largest variable `y` and asserted to the equality set `a`. In addition, the constraint of `k!1` to non-negative reals is recorded in the constraint set `c`.

```
ics> reset.
:unit
ics> assert x <= y + 2.
:ok s1
ics> show.
:val
c:[k!1>=0]
a:[y = -2 +  x +  k!1]
```

Now, the inequality `y <= z + 4` is rewritten as `-6 + -1 * z + x + k!1 = k!2`, with `k!2` a nonnegative slack variable, since `-6 + -1 * z + x + k!1` is the canonical form of `y - z - 4` in state `s1` above, and solved for the largest variable `z`.

```
ics> assert y <= z + 4.
:ok s2
ics> show.
:val
c:[k!2>=0;  k!1>=0]
a:[y = -2 +  x +  k!1;  z = -6 +  x +  k!1 +  k!2]
```

The inequality `z + 6 <= x` is reduced to the equality `-k!1 - k!2 = k!3` with `k!3` nonnegative, since `z + 6 - x` canonizes to `k!1 + k!2`. The sign interpretation of `-k!1 - k!2` is `<=0`, that is, it is constraint to nonpositive real numbers.

```
ics> sign -k!2 - k!1.
:val <=0
```

Thus, because of the equality `-k!1 - k!2 = k!3`, `k!3` is `0`, and therfore both `k!1` and `k!2` are zero.

```
ics> assert z + 6 <= x.
:ok s3
ics> show.
:val
 c:[k!2=0;  k!1=0]
 a:[k!2 = 0;  y = -2 +  x;  z = -6 +  x]
```

In effect, the implied equalities `x = y + 2` and `x = z + 6` are respected by the canonizer.

```
ics> can x.
:val x
ics> can y + 2.
:val x
ics> can z + 6.
:val x
```

One distinguishing feature of ICS is its management of dynamics contexts. In the example above, all intermediate states are maintained in a symbol table.

```
ics> symtab.
:val empty |-> state({})
s1 |-> state({-2 +  -1 * y +  x<=0})
s2 |-> state({-4 +  -1 * z +  y<=0,  -2 +  -1 * y +  x<=0})
s3 |-> state({-4 +  -1 * z +  y<=0,  -2 +  -1 * y +  x<=0,  6 +  z +  -1 * x<=0})
```

Most commands can access a state directly through its name in the symbol table. For example, the logical context of the second state is obtained using the command `ctxt@s2`.

```
ics> ctxt@s2.
:val {-4 +  -1 * z +  y<=0,  -2 +  -1 * y +  x<=0}
```

Names are also used for asserting facts to specific contexts. The following command, for example, extends the state `s3` with the disequality `x <> 2`.

```
ics> assert@s3 x <> 2.
:ok s4
```

Now, `s4` is the current state, but `s3` can be restored to be the current state using

```
ics> restore s2.
```

Thus, the ICS interface includes the management of dynamic contexts, which is important for using it as a verification backend in symbolic simulation or proof search.

Dynamic contexts are also used in extending the core ICS as described above with a SAT solver for deciding the satisfiability of Boolean combinations of equalities and inequalities. Such a decision procedure is available as the command `sat`. Obviously, the following Boolean formula is unsatisfiable (`|` denotes disjunction and `&` is conjunction.

```
ics> sat (x = 1 | x = 4) & x > 5.
:unsat
```

In case such a propositional formula is satisfiable, a conjunction of facts is returned for describing a set of satisfying assignments.

```
ics> sat (x = 1 | x = 2 | x = 3) & x > 1.
:sat(s5) [-1 + x > 0;  x = 3]
```

Here, all assignments to `x` satisfying both `-1 + x > 0` and `x = 3`, describe models for the input formula. Here, there is obviously only one possible assignent, and the description is not minimal. In addition, a new context, of name `s5`, is created for these set of assignments. Using only propositional variables, the `sat` command reduces to a Boolean satisfiability solver.

```
ics> sat p & [~p | r].
:sat(s6) [r |-> true;  p |-> true]
```

3.2 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* has neither been shown to be valid nor inconsistent in the current context, the current context is modified to include new information obtained from *atom*. In addition, a new name is generated for this context and a symbol table entry is added for this name. The result is of the form `:ok si`.

   Notice: the result `:ok` does not necessarily mean that the database is indeed consistent, since the language accepted by ICS is undecidable.

Examples: Asserting the equalities `f(v) = v` and `f(u) = u - 1` yields new contexts of name `s1` and `s2`. Only after asserting `u = v` is a contradiction detected.

```
ics> assert f(v) = v.
:ok s1
ics> assert f(u) = u - 1.
:ok s2
ics> assert u = v.
:unsat
```

Names of context such as `s1` may also be used to address contexts in the symbol table as in `assert@s1` below.

```
ics> assert x = y.
:ok s1
ics> assert y = z.
:ok s2
ics> assert@s1 y = 2.
:ok s3
ics> symtab s3.
:val state({x = y,  y = 2})
ics> ctxt.
:val {x = y,  y = 2}
```

Canonization.

`can` *term* .

For a term `t`, `can t` returns a tern, which is a *canonical* representative of the equivalence class of `t` as induced by the atoms in the current context.

Logical Context.

`ctxt .`

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

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*but different occurrences of *term*are structure-shared.

Disequalities.

`diseq` *term* .

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

Domain Constraint.

`dom` *term* .

Returns domain constraints `Real` or `Int` for arithmetic terms and `None` for non-arithmetic terms. The result is independent of the current logical context, since only information from the symbol table is accessed.

Examples:

```
ics> dom x.
:val None
ics> dom x{int}.
:val int
ics> dom y + z.
:val real
ics> dom y{int} + z{int}.
:val int
ics> sig y : int.
:unit
ics> dom y^2.
```

By default, the domain of interpretation of a variable `x` is unconstrained. Explicit interpretation domains are given by `x{int}` or `x{real}`. The domain of an arithmetic terms is found by an abstract domain interpretation, which encodes, for example, the closedness of integers under addition. Interpretation domains for variables may also be declared in the symbol table.

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.

Exit.

`exit .`

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

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.

Examples:

```
ics> assert x = y.
:ok s1
ics> ctxt.
:val {x = y}
ics> forget.
:unit
ics> ctxt.
:val {}
```

Finds in solution sets.

`find` [`@`*name*] (`u` | `a` | `t` | `bv`) *term* .

If the equality `x = t` is, in the current context, in the specified solution set for the specified equality theory, say `a`, then `find a x` returns `t` and otherwise `x`. The addressing `find@s1 a x` may be used to address the solution set for, say, the arithmetic theory in the context `s1` in the symbol table.

Help.

`help` [`commands` | `syntax`].

Display help texts about commands and syntax of terms, atoms, and propositions.

Inverse find in solution sets.

`inv` [`@`*name*] (`u` | `a` | `t` | `bv`) *term* .

If the equality `x = t` is, in the current context, in the specified solution set for the specified equality theory, say `a`, then `inv a x` returns `t` and otherwise `None`. Thus, whenever `inv` is defined, it is inverse to `find`. The addressing `inv@s1 a x` may be used to address the solution set for, say, the arithmetic theory in the context `s1` in the symbol table.

Definition of Propositions.

`prop` *var* `:=` *proposition* .

Extend the symbol table with a definition *var*for the proposition *proposition*. In such a context, variable *var*is always expanded to *proposition*but different occurrences of *proposition*are structure-shared. see also command `def`. This command fails if there is already a *var*in the symbol table.

Resetting.

`reset .`

Reinitializes all internal data structures including setting the current logical context to the empty context and the symbol table is emptied out.

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*.

Saving the current logical context.

`save` {*var*} .

Adding a symbol table entry *var*for the current logical state.

Satisfiability Solver.

`sat` *proposition* .

A satisfiability solver for propositional formulas over atoms. Returns `:unsat` if the formulas has been shown to be unsatisfiable or `:sat` together with an assignment to the Boolean variables and the truth values of the atoms in a satisfying assignment. In addition, a name is added in the symbol table for the state corresponding to the conjunction of the atoms in a satisfying assignment, but the current logical state is unchanged.

Examples: Literals might be just Boolean variables and the satisfibility of the Boolean probem (`|` is disjunction, `&` is conjunction, `#` is exclusive or, and `~` is negation) is tested as follows.

```
ics> sat x | y | [z & ~x] # y.
:sat(s1) [x |-> true]
```

Notice that brackets `[` and `]` as in `[z & x̃]` are preferred for structuring propositional formulas. The values for the variables `y` and `z` are *don't cares* and therefore not explicitly stated. In addition to Boolean formulas, the command `sat` also handles Boolean formulas over atomic constraints.

```
ics> sat x > y & [y = 2 # ~[x <> 3]].
:sat(s1) [-1 * y +  x>0;  y <> 2;  x = 3]
```

Now, each possible assignment to `x` and `y`, which satisfy the given constraints, is a candidate satisfying assignment of the input formula.

Signature Declaration.

`sig` *var* : `bv[i]` | (int | real) .

Declare a variable to be interpreted over the set of bitvectors of width `i` or the integers or the reals. Notice that only bitvector variables have to be declared before use, since context information is used for inferring parameters when applying infix bitvector operators.

Sigmatization.

`sigma` | .

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

Sign Constraints.

`sign` *term* .

A *sign* constraint in `<0`, `<=0`, `=0`, `>0`, `>=0` may be assigned to terms relative to the current constraint table. For example, assigning sign `<0` to a term `t` restricts the interpretations of `t` to the negative numbers, and similarly for all the other signs.

Examples: Sign constraints are usually generated from inequalities such as `x >= y` below. This inequality is transformed into an arithmetic equality using a newly generated *slack variable* `k!1` which is assigned the sign `>=0` in the constrainit part `c`.

```
ics> assert x >= y.
:ok s1
ics> show.
:val
c:[k!1>=0]
a:[y = x +  -1 * k!1]
ics> sign k!1.
:val >=0
ics> sign y - x.
:val <=0
```

Displaying the context.

`show` .

Displays the current logical state which consists of a

Variable equalities. The `v` part represents a set of equalities over variables. For example

```
v:[a:{a,  b};  x:{x,  y,  z}]
```

says that `a` and `b` are equivalent and that `x`, `y`, and `z` are equivalent. The canonical representatives each the two non-trivial equivalence classes are `a` and `x`.

Variable disequalities. The `d` part is just a conjunction of disequalities over variables

```
d:[y <> x;  z <> y]
```

The set of variables known to be disequal can also be obtained using the `diseq` command.

Variable constraints are conjunctions are sign interpretations for internally generated slack variables. This information is used, for example, by the `sign` command.

Theory-specific solution sets. A theory-specific solution set is a conjunction of equalities `x = t` with `x` a variable and `t` a non-variable term with function symbols in only one theory. Variables in terms might also be internally generated variables of the form `x!i`. For all interpreted theories, the equations in a solved form are actually solved in that variables `x` on a rhs do not occur in any of the lhs. The solution sets can be queried with the `find`, `inv`, and the `use` command.

Slack equalities. Are equalities between internally generated slack variables. These equalities can not be manipulated or queried with any other command.

Solving.

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

Theory-specific solver for input equality. Returns either a solved list of equalities with variables on the lhs which is, in the given theory, 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`).

Examples. The first example demonstrates solving in the theory `a` of linear arithmetic.

```
ics> solve a x + 2 = y - 3.
 :val [y = 5 +  x]
```

Solving in the theory `p` of pairs might introduce fresh variables such `t!2` below.

```
ics> solve p car(x) = cons(u, v).
:val [x = cons(cons(u,  v),  t!2)]
```

The following illustrates solving in the theory of bitvectors.

```
ics> sig x2 : bitvector[2].
:unit
ics> sig x3 : bitvector[3].
:unit
ics> solve bv x2 ++ 0b10 = 0b10 ++ x2.
:val [x2 = 0b10]
ics> solve bv x3 ++ 0b10 = 0b10 ++ x3.
:unsat
```

Symbol Table.

`symtab` {*var*} .

`symtab` display the current symbol table, and `symtab` *var*displays the symbol table entry associated with *var*. Such an entry might either be a logical context entry, a term definition, a definition of a proposition, or a signature entry for domain restrictions of variables.

Examples:

```
ics> assert x = y.
:ok s1
ics> symtab.
:val empty |-> state({})
s1 |-> state({x = y})

ics> def x := y + z.
:unit
ics> prop z := a | b.
:unit
ics> symtab.
:val empty |-> state({})
s1 |-> state({x = y})
x |-> def(z +  y)
z |-> def(a |  b)

ics> sig x : bitvector[2].
:error Name x already in table

ics> sig b : bitvector[2].
:unit
ics> sig q : int.
:unit
ics> symtab.
:val b |-> sig(2)
empty |-> state({})
```

```
q |-> type(int)
s1 |-> state({x = y})
x |-> def(z +  y)
z |-> def(a |  b)
```

Type Definition.

`trace` *name*, ..., *name* .

Tracing facility is used mainly for debugging purposes. However, using `trace rule` might sometimes be useful to analyze which facts are internally being asserted by ICS. Similarly, trace levels such as `v`, `d`, `a`, `c` can be used to trace updates on internal data structures.

Type Definition.

`untrace .`

Disable all tracing.

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

See Appendix C.

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

[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.

```
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.

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.

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 A.

References

[Oca] The Objective Caml language. `http://caml.inria.fr/`. 3

[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. 3

[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. 2

A 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.

```
/***
   PURPOSE

   NOTES

   HISTORY
     demoura - Aug 8, 2002: Created.
***/
#include<stdio.h>
#include<stdlib.h>
#include<iostream.h>

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

extern "C" {
void ics_deregister(value* r);
}

#define MAX_ARRAY_SIZE 1024

int CALLS_TO_ICS = 0;

value * y1_ge_0;
value * y2_ge_0;
value * ny1_eq_y1[MAX_ARRAY_SIZE];
value * ny2_eq_y2[MAX_ARRAY_SIZE];
value * ny1_eq_y2_plus_1[MAX_ARRAY_SIZE];
value * ny2_eq_y1_plus_1[MAX_ARRAY_SIZE];
value * y1_eq_0[MAX_ARRAY_SIZE];
value * y2_eq_0[MAX_ARRAY_SIZE];
value * y1_lt_y2[MAX_ARRAY_SIZE];
value * y2_lt_y1[MAX_ARRAY_SIZE];

#define BUFFER_SIZE 8192

void init_arrays(int max_depth)
{
static char buffer[BUFFER_SIZE];
y1_ge_0 = ics_atom_of_string("y10 >= 0");
y2_ge_0 = ics_atom_of_string("y20 >= 0");
```

```
for(int i = 0; i <= max_depth; i++) {
sprintf(buffer, "y1%d = y1%d", i+1, i);
ny1_eq_y1[i] = ics_atom_of_string(buffer);

sprintf(buffer, "y2%d = y2%d", i+1, i);
ny2_eq_y2[i] = ics_atom_of_string(buffer);

sprintf(buffer, "y1%d = y2%d + 1", i+1, i);
ny1_eq_y2_plus_1[i] = ics_atom_of_string(buffer);

sprintf(buffer, "y2%d = y1%d + 1", i+1, i);
ny2_eq_y1_plus_1[i] = ics_atom_of_string(buffer);

sprintf(buffer, "y1%d = 0", i);
y1_eq_0[i] = ics_atom_of_string(buffer);

sprintf(buffer, "y2%d = 0", i);
y2_eq_0[i] = ics_atom_of_string(buffer);

sprintf(buffer, "y1%d < y2%d", i,i);
y1_lt_y2[i] = ics_atom_of_string(buffer);

sprintf(buffer, "y2%d < y1%d", i,i);
y2_lt_y1[i] = ics_atom_of_string(buffer);
}
}

bool process(value * state, value ** next, bool d_prev, value * atom) {
CALLS_TO_ICS++;
value * status = ics_process(state, atom);
bool result;
if (ics_is_consistent(status)) {
*next = ics_d_consistent(status);
result = true;
}
else if (ics_is_redundant(status)) {
*next = state;
result = true;
}
else if (ics_is_inconsistent(status)) {
result = false;
}
if (d_prev) {
ics_deregister(state);
```

```
free(state);
}
ics_deregister(status);
free(status);
return result;
}

int MAX_DEPTH = 0;

bool printed = false;

#define ERROR() { \
  if (!printed) { \
   ics_context_pp(state); \
printed = true; \
} \
  cout << endl; \
cout << "pc1 = " << pc1 << ", pc2 = " << pc2 << ", at depth = " << depth << endl;
\
return 0; \
}

int bakery_step(int pc1, int pc2, int depth, value * state)
{
if (depth >= MAX_DEPTH)
return 1;

// cout << "AT pc1 = " << pc1 << ", pc2 = " << pc2 << " depth = " << depth << endl;

if (pc1 == 3 && pc2 == 3) {
cout << "Error detected.... pc1 = " << pc1 << ", pc2 = " << pc2 << " at depth = " << depth << endl;
return 0;
}

value * new_state;

switch (pc1) {
case 1:
if(!process(state, &new_state, 0, ny1_eq_y2_plus_1[depth]))
break;
if(!process(new_state, &new_state, 1, ny2_eq_y2[depth]))
break;
if (!bakery_step(2, pc2, depth+1, new_state)) {
ERROR();
```

```
}
break;
case 2:
if(!process(state, &new_state, 0, ny1_eq_y1[depth]))
break;
if(!process(new_state, &new_state, 1, ny2_eq_y2[depth]))
break;
{
value * saved_state = new_state;
if (process(saved_state, &new_state, 0, y2_eq_0[depth]) &&
!bakery_step(3, pc2, depth+1, new_state)) {
ERROR();
}
if (process(saved_state, &new_state, 0, y1_lt_y2[depth]) &&
!bakery_step(3, pc2, depth+1, new_state)) {
ERROR();
}
ics_deregister(saved_state);
free(saved_state);
}
break;
case 3:
if (!process(state, &new_state, 0, y1_eq_0[depth+1]))
break;
if (!process(new_state, &new_state, 1, ny2_eq_y2[depth]))
break;
if (!bakery_step(1, pc2, depth+1, new_state)) {
ERROR();
}
break;
}

switch(pc2) {
case 1:
if(!process(state, &new_state, 0, ny2_eq_y1_plus_1[depth]))
break;
if(!process(new_state, &new_state, 1, ny1_eq_y1[depth]))
break;
if (!bakery_step(pc1, 2, depth+1, new_state)) {
ERROR();
}
break;
case 2:
if(!process(state, &new_state, 0, ny1_eq_y1[depth]))
```

```
break;
if(!process(new_state, &new_state, 1, ny2_eq_y2[depth]))
break;
{
value * saved_state = new_state;
if (process(saved_state, &new_state, 0, y1_eq_0[depth]) &&
!bakery_step(pc1, 3, depth+1, new_state)) {
ERROR();
}
if (process(saved_state, &new_state, 0, y2_lt_y1[depth]) &&
!bakery_step(pc1, 3, depth+1, new_state)) {
ERROR();
}
ics_deregister(saved_state);
free(saved_state);
}
break;
case 3:
if (!process(state, &new_state, 0, y2_eq_0[depth+1]))
break;
if (!process(new_state, &new_state, 1, ny1_eq_y1[depth]))
break;
if (!bakery_step(pc1, 1, depth+1, new_state)) {
ERROR();
}
break;
}

// ics_deregister(state);
// free(state);

return 1;
}


int main(int argc, char ** argv)
{
ics_caml_startup(1, argv);
cout << "depth = " << argv[1] << endl;
int depth = atoi(argv[1]);
cout << "ICS Started...\n";
init_arrays(depth);
cout << "Atoms initialized...\n";
```

```
value * ini_state = ics_context_empty();
process(ini_state, &ini_state, 0, y1_ge_0);
process(ini_state, &ini_state, 0, y2_ge_0);

MAX_DEPTH = depth;

if (!bakery_step(1, 1, 0, ini_state))
cout << "ERROR...." << endl;

cout << "calls to ICS = " << CALLS_TO_ICS << endl;
return 0;
}

extern "C" {

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

}
```

B Yacc Input Grammar

```
/*
 * 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."))

let lastresult = ref(Result.Unit())

let name_of_nonstrict_slack = Name.of_string "k"
let name_of_strict_slack = Name.of_string "l"


%}

%token DROP CAN ASSERT EXIT SAVE RESTORE REMOVE FORGET RESET SYMTAB SIG VALID UNSAT
%token TYPE SIGMA
%token SOLVE HELP DEF PROP TOGGLE SET TRACE UNTRACE CMP FIND USE INV SOLUTION PARTITION MODEL
%token SHOW SIGN DOM SYNTAX COMMANDS SPLIT SAT ECHO
%token DISEQ CTXT
%token IN TT FF DEF
%token EOF

%token ARITH TUPLE

%token <string> IDENT
%token <string> STRING
%token <int> INTCONST
%token <Mpa.Q.t> RATCONST
%token <Name.t> PROPVAR

%token IN
%token BOT INT REAL BV TOP
```

```
%token INF NEGINF
%token ALBRA ACLBRA CLBRA

%token LPAR RPAR LBRA RBRA LCUR RCUR PROPLPAR PROPRPAR 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 PROPVAR DISJ XOR IMPL BIIMPL CONJ NEG
%token IF THEN ELSE END
%token PROJ
%token CREATE

%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 <Result.t> commands
%type <Result.t> commandseof
%type <unit> commandsequence


%start termeof
%start atomeof
%start commands
%start commandsequence
%start commandseof



%%

termeof : term EOF           { $1 }
atomeof : atom EOF           { $1 }
commandseof : command EOF    { $1 }

commands : command DOT       { $1 }
| EOF                        { raise End_of_file }

commandsequence :
  command DOT commandsequence    {  lastresult := $1 }
| command DOT EOF                { lastresult := $1; raise(Result.Result(!lastresult)) }


int:
  INTCONST  { $1 }

rat:
  int       { Q.of_int $1 }
| RATCONST  { $1 }
;


name: IDENT            { Name.of_string $1 }

namelist : name        { [$1] }
| namelist COMMA name  { $3 :: $1 }
```

```
funsym:
  name                                    { Sym.Uninterp($1) }
| PLUS                                    { Sym.Arith(Sym.Add) }
| TIMES                                   { Sym.Pp(Sym.Mult) }
| EXPT LBRA int RBRA                      { Sym.Pp(Sym.Expt($3)) }
| CONS                                    { Sym.Pair(Sym.Cons) }
| CAR                                     { Sym.Pair(Sym.Car) }
| CDR                                     { Sym.Pair(Sym.Cdr) }
| UNSIGNED                                { Sym.Bvarith(Sym.Unsigned) }
| 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 }

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 }
| list              { $1 }
| apply             { $1 }
;

var:
  name  { try
    match Symtab.lookup $1 (Istate.symtab()) with
      | Symtab.Def(Symtab.Term(a)) -> a
      | Symtab.Type(d)  -> Term.mk_var $1 (Some(d))
```

```
      | _ -> Term.mk_var $1 None
  with
      Not_found -> Term.mk_var $1 None }
| name LCUR dom RCUR { Term.mk_var $1 (Some($3))}
| FRESH optdom  { let (x, k) = $1 in
  let n = Name.of_string x in
    if Name.eq n name_of_strict_slack then
      Term.mk_slack (Some(k)) false $2
    else if Name.eq n name_of_nonstrict_slack then
      Term.mk_slack (Some(k)) true $2
    else
      Term.mk_rename n (Some(k)) $2 }
| FREE   { Term.Var(Var.mk_free $1) }
;

varlist : var        { [$1] }
| varlist COMMA var  { $3 :: $1 }


optdom:   { None }
| LCUR dom RCUR { Some($2) }


app:
  funsym LPAR termlist RPAR     { Th.sigma $1 (List.rev $3) }
| constsym                      { Th.sigma $1 [] }

list:
  term LISTCONS term            { Coproduct.mk_inj 1 (Pair.mk_cons $1 $3) }
| HEAD LPAR term RPAR           { Pair.mk_car (Coproduct.mk_out 1 $3) }
| TAIL LPAR term RPAR           { Pair.mk_cdr (Coproduct.mk_out 1 $3) }
| NIL                           { Coproduct.mk_inj 0 (Bitvector.mk_eps) }

apply:
  term APPLY term               { Apply.mk_apply
    Th.sigma
                                     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 int                 { 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:
  CREATE LPAR term RPAR      { Arr.mk_create $3 }
| term LBRA term ASSIGN term RBRA { Arr.mk_update Istate.is_equal $1 $3 $5 }
| term LBRA term RBRA        { Arr.mk_select Istate.is_equal $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 }
;

prop:
  LPAR prop RPAR                { $2 }
| LBRA prop RBRA                { $2 }
| name                            { try
      match Symtab.lookup $1 (Istate.symtab()) with
| Symtab.Def(Symtab.Prop(p)) -> p
| _ -> Prop.mk_var $1
      with
  Not_found -> Prop.mk_var $1 }
| atom                            { Prop.mk_poslit $1 }
| prop CONJ prop                  { Prop.mk_conj [$1; $3] }
| prop DISJ prop                  { Prop.mk_disj [$1; $3] }
| prop BIIMPL prop                { Prop.mk_iff $1 $3 }
| prop XOR prop                   { Prop.mk_neg (Prop.mk_iff $1 $3) }
| prop IMPL prop                  { Prop.mk_disj [Prop.mk_neg $1; $3] }
| NEG prop %prec prec_unary       { Prop.mk_neg $2 }
| IF prop THEN prop ELSE prop END { Prop.mk_ite $2 $4 $6 }
;

atom:
  FF                     { Atom.mk_false }
| TT                     { Atom.mk_true }
| term EQUAL term        { Atom.mk_equal($1, $3)}
| term DISEQ term        { Atom.mk_diseq($1, $3) }
| term LESS term         { Atom.mk_lt ($1, $3) }
| term GREATER term      { Atom.mk_gt ($1, $3) }
| term LESSOREQUAL term  { Atom.mk_le ($1, $3) }
| term GREATEROREQUAL term { Atom.mk_ge ($1, $3) }


dom:
  INT          { Dom.Int }
| REAL         { Dom.Real }
;


termlist:             { [] }
| term                { [$1] }
```

```
| termlist COMMA term { $3 :: $1 }
;

signature:
  BV LBRA INTCONST RBRA     { $3 }
;

command:
  CAN term                  { Result.Term(Istate.can $2) }
| ASSERT optname atom       { Result.Process(Istate.process $2 $3) }
| DEF name ASSIGN term      { Result.Unit(Istate.def $2 (Symtab.Term($4))) }
| PROP name ASSIGN prop     { Result.Unit(Istate.def $2 (Symtab.Prop($4))) }
| SIG name COLON dom        { Result.Unit(Istate.typ [$2] $4) }
| SIG name COLON signature  { Result.Unit(Istate.sgn $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) }
| SIGN optname term         { Result.Cnstrnt(Istate.sign $2 $3) }
| DOM optname term          { Result.Dom(Istate.dom $2 $3) }
| 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 ()) }
| SAT prop                  { Result.Sat(Istate.sat $2) }
```

```
| MODEL optname varlist     { Result.Solution(Istate.model $2 $3) }
| ECHO STRING               { Result.Unit(Format.eprintf "%s@." $2) }
| help                      { Result.Unit($1) }
;


identlist :
  IDENT                     { [$1] }
| identlist COMMA IDENT     { $3 :: $1 }


th: IDENT  { Th.of_string $1 } /* may raise [Invalid_argument]. */

help:
  HELP                      { Help.on_help () }
| HELP SYNTAX               { Help.syntax () }
| HELP COMMANDS             { Help.commands () }
;


optname:                    { None }
| KLAMMERAFFE name          { Some($2) }
;


%%
```

C Module `Ics` : Application programming interface.

The ICS API includes function for

- asserting formulas to a logical context,
- switching between different logical contexts, and
- 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 `Ics.process`[C], 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 `Ics.cmd_rep`[C] 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.

Channels

`type inchannel = Pervasives.in_channel`

`inchannel` is the type of input channels.

`type outchannel = Format.formatter`

Formattable output channel.

`val channel_stdin : unit -> inchannel`

`channel_stdin` is the predefined standard input channel.

`val channel_stdout : unit -> outchannel`

`channel_stdout` is the predefined standard output channel.

`val channel_stderr : unit -> outchannel`

`channel_stdout` is the predefined standard error channel. All ICS trace messages are put onto this channel.

`val inchannel_of_string : string -> inchannel`

`inchannel_of_string str` opens an input channel for reading from a string (file name). This function raises `Sys_error` in case such a channel can not be opened.

`val outchannel_of_string : string -> outchannel`

`outchannel_of_string str` opens an output channel for writing from a string (file name). This function raises `Sys_error` in case such a channel can not be opened.

Multi-precision arithmetic

`type q`

Type for representing the rational numbers.

`val num_of_int : int -> q`

`num_of_int n` constructs a rational from the integer `n`.

`val num_of_ints : int -> int -> q`

`num_of_ints n m`, for `m <> 0`, constructs a normalized representation of the rational `n/m` in `q`.

```
val ints_of_num : q -> string * string
```

`ints_of_num q` decomposes a rational with numerator `n` and denumerator `m` into `("n", "m")`.

```
val string_of_num : q -> string
```

`string_of_num q` constructs a string (usually for printout) of a rational number

```
val num_of_string : string -> q
```

`num_of_string s` constructs a rational, whenever `s` is of the form `n/m` where `n` and `m` are integers.

Names

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
```

Representation of strings.

```
val name_of_string : string -> name
```

`name_of_string str` constructs a name `n` from a string such that `Ics.name_to_string`[C]`(n)` yields `str`.

```
val name_to_string : name -> string
```

`name_to_string n` is the inverse operation of `Ics.name_to_string`[C].

```
val name_eq : name -> name -> bool
```

`name_eq n m` holds iff the corresponding strings `Ics.name_to_string`[C]`(n)` and `Ics.name_to_string`[C]`(m)` are equal. This equality test is constant in the length of strings.

Arithmetic Constraints

An **arithmetic constraint** consists of

- a domain restriction, and

- a sign restriction. A real number satisfies such a constraint if it satisfies both
- the domain restriction and
- the sign restriction. To each constraint `s` we associate the set `D(s)` of reals satisfying these requirements.

```
type  cnstrnt
```

```
val cnstrnt_pp : cnstrnt -> unit
```

Printing a constraint to standard output.

```
val cnstrnt_mk_zero : unit -> cnstrnt
```

Constructing a constraints with `D(cnstrnt_mk_zero()) = {0}`

```
val cnstrnt_mk_pos : unit -> cnstrnt
```

`D(cnstrnt_mk_pos)` is the set of positive reals.

```
val cnstrnt_mk_neg : unit -> cnstrnt
```

`D(cnstrnt_mk_neg)` is the set of negative reals.

```
val cnstrnt_mk_nonneg : unit -> cnstrnt
```

`D(cnstrnt_mk_nonneg)` is the set of non-negative reals.

```
val cnstrnt_mk_nonpos : unit -> cnstrnt
```

`D(cnstrnt_mk_nonpos)` is the set of non-positive reals.

```
val cnstrnt_is_empty : cnstrnt -> bool
```

`cnstrnt_is_empty(s)` holds iff `D(s)` is the empty set.

```
val cnstrnt_is_pos : cnstrnt -> bool
```

`cnstrnt_is_pos s` holds iff `D(s)` is the set of positive reals

```
val cnstrnt_is_neg : cnstrnt -> bool
```

`cnstrnt_is_neg s` holds iff `D(s)` is the set of negative reals

```
val cnstrnt_is_nonneg : cnstrnt -> bool
```

`cnstrnt_is_nonneg s` holds iff `D(s)` is the set of non-negative reals `val cnstrnt_is_nonpos : cnstrnt -> bool`

`cnstrnt_is_nonpos s` holds iff `D(s)` is the set of non-positive reals

`val cnstrnt_eq : cnstrnt -> cnstrnt -> bool`

`cnstrnt_eq s t` holds iff `D(s)` equals `D(t)`.

`val cnstrnt_sub : cnstrnt -> cnstrnt -> bool`

`cnstrnt_sub s t` holds iff `D(s)` is a subset of `D(t)`.

`val cnstrnt_disjoint : cnstrnt -> cnstrnt -> bool`

`cnstrnt_disjoint s t` holds iff the intersection of `D(s)` and `D(t)` is empty.

`val cnstrnt_inter : cnstrnt -> cnstrnt -> cnstrnt`

`D(cnstrnt_inter s t)` is the intersection of `D(s)` and `D(t)`.

Equality theories

An **equality theory** is associated with each function symbol of terms. These theories are indexed by naturals between `0` and `maxtheories = 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).

`val th_to_string : int -> string`

`th_to_string th` returns the unique name associated to theory `th`.

`val th_of_string : string -> int`

`th_of_string s` returns theory `th` if `to_string th` is `s`; otherwise the result is unspecified.

Function Symbols

```
type  sym
```

```
val sym_theory_of : sym -> int
```

`sym_theory_of f` returns the theory `th` associated with the function symbol `f`.

```
val sym_eq : sym -> sym -> bool
```

`sym_eq` tests for equality of two function symbols.

```
val sym_cmp : sym -> sym -> int
```

`sym_cmp f g` provides a total ordering on function symbols. If it returns

- a negative integer, then `f` is said to be smaller than `g`,
- `0`, then `f` is equal to `g` and `Ics.sym_eq`[C]`(f, g)`, and
- a positive numbe, then `f` is said to be larger than `g`.

Uninterpreted function symbols

```
val sym_is_uninterp : sym -> bool
```

`sym_is_uninterp f` holds iff `f` is an uninterpreted function symbol.

```
val sym_d_uninterp : sym -> name
```

`sym_d_uninterp f` returns the name associated with an uninterpreted function symbol `f`. This accessor is undefined if `Ics.sym_is_uninterp`[C]`(f)` does not hold.

Linear arithmetic function symbols are either

- *numerals* for representing all rational numbers,
- the *addition* symbols,
- symbols for representing *linear multiplication* by a rational of type `Ics.q`[C].

```
val sym_mk_num : q -> sym
```

`sym_mk_num q` constructs a numeral symbol for representing `q`.

```
val sym_is_num : sym -> bool
```

`sym_is_num f` holds iff `f` represents a numeral.

```
val sym_d_num : sym -> q
```

`sym_d_num f` returns the rational `q` if `f` represents `q`. This accessor is undefined if `Ics.sym_is_num`[C] does not hold.

```
val sym_mk_add : unit -> sym
```

`sym_mk_add()` constructs the addition symbol.

```
val sym_is_add : sym -> bool
```

`sym_is_add f` holds iff `f` represents the addition symbol.

```
val sym_mk_multq : q -> sym
```

`sym_mk_multq q` constructs the symbol for linear multiplication by a rational `q`.

```
val sym_is_multq : sym -> bool
```

`sym_is_multq f` holds iff `f` represents a linear multiplication symbol.

```
val sym_d_multq : sym -> q
```

`sym_d_multq f` returns `q` if `f` represents linear multiplication by `q`. This accessor is undefined if `Ics.sym_d_multq`[C] does not hold.

Symbols of the **product theory** consist of

- tupling
- projections of the `i`-th component in a tuple of length `n`.

```
val sym_mk_cons : unit -> sym
```

`sym_mk_cons()` constructs the symbol for tupling.

```
val sym_is_cons : sym -> bool
```

`sym_is_cons f` holds iff `f` represents tupling.

```
val sym_is_car : sym -> bool
```

`sym_is_car f` holds iff `f` represents a projection.

```
val sym_mk_car : unit -> sym
```

`sym_mk_car()` constructs the symbol for the first projection

`val sym_is_cdr : sym -> bool`

`sym_is_cdr f` holds iff `f` represents a projection.

`val sym_mk_cdr : unit -> sym`

`sym_mk_cdr()` constructs the symbol for the first projection

Symbols of the theory of **coproducts** are eith

- left and right injections,
- left and right coinjections

`val sym_mk_inl : unit -> sym`

`sym_mk_inl ()` constructs symbol for left injection.

`val sym_is_inl : sym -> bool`

`sym_is_inl f` holds iff `f` represents left injection.

`val sym_mk_inr : unit -> sym`

`sym_mk_inr ()` constructs symbol for right injection.

`val sym_is_inr : sym -> bool`

`sym_is_inr f` holds iff `f` represents right injection.

`val sym_mk_outl : unit -> sym`

`sym_mk_outl ()` constructs symbol for left injection.

`val sym_is_outl : sym -> bool`

`sym_is_outl f` holds iff `f` represents left coinjection.

`val sym_mk_outr : unit -> sym`

`sym_mk_outr ()` constructs symbol for right coinjection.

`val sym_is_outr : sym -> bool`

`sym_is_outr f` holds iff `f` represents right coinjection.

Symbols in the fixed-sized **bitvector** theory include

- constant bitvectors of length `n >= 0`,
- concatenation of a bitvector of width `n >= 0` with a bitvector of width `m >= 0`,
- extraction of bits `i` through `j` of a bitvector of length `n >= 0`, (`0 <= i <= j < n`), and
- bitwise conditionals for bitvectors of length `n`.

`val sym_mk_bv_const : string -> sym`

> `sym_mk_bv_const str` constructs, say, a bitvector constant `01001` from a string of the form `"01001"`. The result is undefined if characters other than `'0'` or `'1'` appear in the string.

`val sym_is_bv_const : sym -> bool`

> `sym_is_bv_const f` holds iff `f` represents a bitvector constant symbol.

`val sym_mk_bv_conc : int -> int -> sym`

> `sym_mk_bv_conc n m` constructs a concatenation symbol with indices `n` and `m`, for `n, m >= 0`, for concatenating a bitvector of width `n` with a bitvector of length `m`.

`val sym_is_bv_conc : sym -> bool`

> `sym_is_bv_conc f` holds iff `f` represents a concatenation symbol.

`val sym_d_bv_conc : sym -> int * int`

> `sym_d_bv_conc f` returns `(n, m)` iff `f` represents a concatenation symbol for bitvectors of width `n` with a bitvector of width `m`.

`val sym_mk_bv_sub : int -> int -> int -> sym`

> `sym_mk_bv_sub i j n` constructs a bitvector extraction symbol for the indices `0 <= i <= j < n`.

`val sym_is_bv_sub : sym -> bool`

> `sym_is_bv_sub f` holds iff `f` represents a bitvector extraction symbol.

`val sym_d_bv_sub : sym -> int * int * int`

`sym_d_bv_sub f` returns `(i, j, n)` iff `f` represents a bitvector extraction of bits `i` through `j` of a bitvector of width `n`.

`val sym_mk_bv_bitwise : int -> sym`

`sym_mk_bv_bitwise n` constructs a symbol for bitwise logical conditionals for bitvectors of width `n >= 0`.

`val sym_is_bv_bitwise : sym -> bool`

`sym_is_bv_bitwise f` holds iff `f` represents a logical bitwise operation symbol.

`val sym_d_bv_bitwise : sym -> int`

`sym_d_bitwise f` returns `n` iff `f` represents a logical bitwise operation of width `n`.

Symbols from the theory of **power products** include

- Multi-ary nonlinear multiplication symbol
- Exponentiation with an integer.

`val sym_mk_mult : unit -> sym`

`sym_mk_mult()` constructs the nonlinear multiplication symbol.

`val sym_is_mult : sym -> bool`

`sym_is_mult f` holds iff `f` represents the nonlinear multiplication symbol.

`val sym_mk_expt : int -> sym`

`sym_mk_expt n` constructs the 'exponentiation by `n`' symbol.

`val sym_is_expt : sym -> bool`

`sym_is_expt f` holds iff `f` represents an exponentiation symbol.

`val sym_d_expt : sym -> int`

`sym_d_expt f` returns `n` if `f` represents the 'exponentiation by `n`' symbol.

Symbols from the theory of **function abstraction and application** include

- function abstraction
- function application

A function application symbol may have a constraint of type `Ics.cnstrnt`[C] associated with it.

`val sym_mk_apply : cnstrnt option -> sym`

`sym_mk_apply co` constructs a symbol for function application with associated constraint `co`.

`val sym_is_apply : sym -> bool`

`sym_is_apply f` holds iff `f` represents the function application symbol.

`val sym_d_apply : sym -> cnstrnt option`

`sym_d_apply f` returns the constraint associated with a function application symbol.

`val sym_mk_abs : unit -> sym`

`sym_mk_abs()` constructs the symbol for function abstraction.

`val sym_is_abs : sym -> bool`

`sym_is_abs f` holds iff `f` represents the function abstraction symbol.

Symbols from the theory of **arrays** include

- array updates (write)
- array selection (read)

`val sym_mk_select : unit -> sym`

The array select symbol.

`val sym_is_select : sym -> bool`

`sym_is_select f` holds iff `f` represents the array selection symbol.

`val sym_mk_update : unit -> sym`

The array update symbol.

`val sym_is_update : sym -> bool`

`sym_is_update f` holds iff `f` represents the array update symbol.

Symbols from the theory of **arithmetic interpretations of bitvectors** include

- the *unsigned* interpretation symbol.

`val sym_mk_unsigned : unit -> sym`

Constructing the unsigned interpretation symbol.

`val sym_is_unsigned : sym -> bool`

`sym_is_unsigned f` holds iff `f` represents the unsigned interpretation symbol.

Variables

The set of all variables is partitioned into different **kinds** of variables, namely

- *external*,
- *fresh*, and
- **bound** variables.

There is a name of type `Name.t` associated with each variable. Names for fresh variables are always of the form `"x!i"`, where `x` is an arbitrary string and `i` is an integer string. The name associated with a bound variable is of the form `"!i"` for an integer `i`.

`type var`

`val var_name_of : var -> name`

`name_of x` returns the name associated with a variable `x`.

`val var_eq : var -> var -> bool`

`eq x y` holds iff `x` and `y` are in the same category (that is, external, fresh, and bound) of variables and if their names are identical.

`val var_cmp : var -> var -> int`

`cmp x y` realizes a total ordering on variables. The result is `0` if `eq x y` holds, it is less than `0` we say, 'x is less than y', and, otherwise, 'x is greater than y'. An external variable `x` is always less than a nonexternal (that is, a fresh or a free) variable `y`. Otherwise, the outcome of `cmp x y` is unspecified.

`val var_mk_external : name -> var`

`var_mk_external x` creates an external variable with associated name `x`.

`val var_mk_bound : int -> var`

`var_mk_bound i` constructs a bound variable with associated name `!i`.

```
val var_is_external : var -> bool
```

`is_var x` holds iff `x` is an external variable.

```
val var_is_bound : var -> bool
```

`is_bound x` holds iff `x` is a bound variable.

```
val var_d_bound : var -> int
```

For a free variable `x`, `var_d_bound x` returns `i`, if `"!i"` is the name associated with `x`. The result is undefined for non-bound variables.

Terms

Terms are either

- variables or
- applications of function symbols of type `Ics.sym`[C] to a list of terms.

```
type  term
val term_of_string : string -> term
```

`term_of_string` parses a string according to the grammar for the nonterminal `Parser.termeof` (see its specification in file `parser.mly`) and builds a corresponding term.

```
val term_input : inchannel -> term
```

`term_input inch` is similar to `Ics.term_of_string`[C] but builds a term by reading from input channel `inch`.

```
val term_output : outchannel -> term -> unit
```

`term_output outch a` prints term `a` on the output channel `out`.

```
val term_to_string : term -> string
```

`term_to_string a` prints a term to a string. This string is parsable by `Ics.term_of_string`[C].

```
val term_pp : term -> unit
```

term_pp a is equivalent to term_output (Ics.stdout()) a.

```
val term_eq : term -> term -> bool
```

term_eq a b holds iff a and b are syntactically equal, that is, either

- both a and b are variables of the same kind and their associated names are equal
- both a and b are application terms with equal function symbols (see Ics.sym_eq[C]), the number of arguments in a and b is equal, and the respective arguments at every position are term equal.

```
val term_cmp : term -> term -> int
```

Comparison term_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.

- Variables are always greater than applications,
- variables are ordered according to Ics.var_cmp[C], and
- applications are ordered lexicographically using Ics.sym_cmp[C] on the function symbols and comparing respective term arguments.

```
val term_mk_var : string -> term
```

Given a string s, term_mk_var s constructs an *external* variable with name s.

```
val term_mk_uninterp : string -> term list -> term
```

term_mk_uninterp s al constructs an application of an uninterpreted function symbol s to a list al of argument terms.

Linear arithmetic terms are built-up from rational constants, linear multiplication of a rational with a variable, and n-ary addition.

Linear arithmetic terms are always normalized as a **sum-of-product** q0 + q1*x1+...+qn*xn where the qi are rational constants and the xi are variables (or any other term not interpreted in this theory), which are ordered such that Ics.term_cmp[C]xi xj is greater than zero for i < j. This implies that any such variable occurs at most once. In addition, qi, for i > 0, is never zero. If qi is one, we just write xi instead of qi * xi, and if q0 is zero, it is simply omitted in the sum-of-product above.

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_is_arith : term -> bool
```

term_is_arith a holds if the toplevel symbol of a is interpreted in linear arithmetic.

```
val term_mk_num : q -> term
```

term_mk_num q constructs a numeral term for representing the rational q.

```
val term_mk_multq : q -> term -> term
```

term_mk_multq q a constructs a term for representing the term a multiplied by q. If a is in sum-of-product form, then so is term_mk_multq q a.

```
val term_mk_add : term -> term -> term
```

term_mk_add a b constructs a term for representing the sum of a and b. If both a and b are in sum-of-product form, then so is term_mk_add a b.

```
val term_mk_addl : term list -> term
```

Iteration of binary addition

- term_mk_addl [] is Ics.term_mk_num(),
- term_mk_addl [a] is a, and
- term_mk_addl (a :: al) is term_mk_add a (term_mk_addl al).

```
val term_mk_sub : term -> term -> term
```

term_mk_sub a b represents the difference a - b. If both a and b are in sum-of-product form, then so is the result.

```
val term_mk_unary_minus : term -> term
```

term_mk_unary_minus a represents the negation of a. If a is in sum-of-product form, then so is the result.

Tuple terms. Tuple terms in normal form do not contain (applicable) projections on tuples.

```
val term_mk_tuple : term list -> term
```

term_mk_tuple [a1;...;an] constructs tuple term for respresenting the tuple (a1,...,an). The result is in tuple normal form, when all ai are in tuple normal form

```
val term_mk_proj : int -> term -> term
```

`term_mk_proj i a` constructs, for `0 <= i < n`, a term for representing the `i`-th projection of an `n`-tuple. If `a` is in tuple normal form, then so is the result.

Bitvector terms are built up from bitvector constants, concatenation of two bitvectors, extraction of a contiguous subrange from a bitvector, and logical bitwise operations. Each bitvector term has a nonnegative *width* associated with it, and bits in a bitvector of width `n` are addressed from `0` to `n-1` in increasing order from left-to-right. All bitvector terms are in *concatenation normal form*, that is, a left-associative concatenation of

- terms uninterpreted in the bitvector theory
- bitvector constants (with adjacent constants merged)
- single extractions from uninterpreted terms in this theory
- bitvector BDDs, which are BDDs with nodes consisting of one of the above classes of terms.

The constructors below all construct concatenation normal forms, whenever their arguments are in this form.

```
val term_mk_bvconst : string -> term
```

`term_mk_bvconst str` constructs a bitvector constant.

```
val term_mk_bvsub : int * int * int -> term -> term
```

`term_mk_bvsub i j n a` constructs, for `0 <= i <= j < n` a term for representing the extraction of the `j-i+1` bits from position `i` through `j` in a term of width `n`.

```
val term_mk_bvconc : int * int -> term -> term -> term
```

`term_mk_bvconc n m a b` constructs the concatenation `a ++ b` of bitvector terms `a` of width `n` with `b` of width `m`.

```
val term_mk_bwite : int -> term * term * term -> term
```

`term_mk_bwite n (a, b, c)` constructs a bitwise logical `ITE(a, b, c)`; the argument bitvector terms are assumed to have width `n`.

```
val term_mk_bwand : int -> term -> term -> term
```

`term_mk_bwand n a b` constructs a bitwise conjunction term for bitvectors `a`, `b` of width `n`.

```
val term_mk_bwor : int -> term -> term -> term
```

`term_mk_bwor n a b` constructs a bitwise disjunction term for bitvectors `a`, `b` of width `n`.

```
val term_mk_bwnot : int -> term -> term
```

`term_mk_bwnot n a` constructs a bitwise negation term for a bitvectors `a` of width `n`.

Boolean Term Constants.

```
val term_mk_true : unit -> term
```

The propositional constant `term_mk_true()` is encoded as the bitvector constant of width `1` with a `1` at position `0`.

```
val term_mk_false : unit -> term
```

The propositional constant `term_mk_false()` is encoded as the bitvector constant of width `1` with a `0` at position `0`.

```
val term_is_true : term -> bool
```

`term_is_true a` holds iff `a` is term equal to `term_mk_true()`.

```
val term_is_false : term -> bool
```

`term_is_false a` holds iff `a` is term equal to `term_mk_false()`.

Coproducts

```
val term_mk_inj : int -> term -> term
```

`term_mk_inj n a` constructs a term for `n`-ary injection.

```
val term_mk_out : int -> term -> term
```

`term_mk_out n a` constructs a term for `n`-ary outjection.

Builtin simplifying constructors.

```
val term_mk_unsigned : term -> term
val term_mk_update : term -> term -> term -> term
val term_mk_select : term -> term -> term
val term_mk_div : term -> term -> term
val term_mk_mult : term -> term -> term
val term_mk_multl : term list -> term
val term_mk_expt : int -> term -> term
val term_mk_apply : term -> term list -> term
```

Set of terms.

```
type terms
```

```
val terms_of_list : term list -> terms
```

```
val terms_to_list : terms -> term list
```

Atoms

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
```

An **atom** is either

- the trivially true atom `atom_mk_true`,
- the unsatisfiable `atom_mk_false`,
- an equality atom `atom_mk_equal a b`,
- a disequality atom `atom_mk_diseq a b`, or
- a constraint atom `atom_mk_in a c`, which constrains `a` to be interpreted over the domain `D(c)` associated with the constraint `c` of type `Ics.cnstrnt`[C].

```
val atom_pp : atom -> unit
```

```
val atom_of_string : string -> atom
```

```
val atom_to_string : atom -> string
```

```
val atom_mk_true : unit -> atom
```

```
val atom_mk_false : unit -> atom
```

```
val atom_mk_equal : term -> term -> atom
```

```
val atom_mk_diseq : term -> term -> atom
```

```
val atom_mk_in : term -> cnstrnt -> atom
```

```
val atom_mk_le : term -> term -> atom
```

```
val atom_mk_lt : term -> term -> atom
```

```
val atom_mk_ge : term -> term -> atom
```

```
val atom_mk_gt : term -> term -> atom
```

```
val atom_is_negatable : atom -> bool
```

```
val atom_negate : atom -> atom
```

```
type atoms
```

```
val atoms_empty : unit -> atoms
```

```
val atoms_singleton : atom -> atoms
```

```
val atoms_add : atom -> atoms -> atoms
```

```
val atoms_to_list : atoms -> atom list
```

Solutions sets

```
type solution
```

A **solution set** is a set of equalities `a = b` over terms `a`, `b`.

`val solution_to_list : solution -> (term * term) list`

Representing a solution set `{a1 = b1,...,an = bn}` as a list of pairs `(ai, bi)`.

Logical Context

`type context`

A logical context represents a conjunction of atoms.

`val context_pp : context -> unit`

Pretty-printing a context to standard output.

`val context_ctxt_pp : context -> unit`

Pretty-printing the logical context in a way that can be read in again by the parser.

`val context_eq : context -> context -> bool`

`context_eq s1 s2` is a constant-time predicate for testing for identity of two states. Thus, whenever this predicate holds, its corresponding contexts are logically equivalent.

`val context_ctxt_of : context -> atoms`

`context_ctxt_of s` returns the logical context of `s` as a set of atoms.

`val context_solution_of : context -> int -> solution`

`context_solution_of th s` returns the solution set for theory `th` in `s`.

`val context_mem : int -> context -> Term.t -> bool`

`context_mem th s x` iff `x = _` is in the solution set for theory `th` in `s`.

`val context_apply : int -> context -> Term.t -> Term.t`

`apply th s x` is `a` when `x = a` is in the solution set for theory `th` in `s`; otherwise `Not_found` is raised.

`val context_find : int -> context -> Term.t -> Term.t`

`find th s x` is `a` if `x = a` is in the solution set for theory `th` in `s`; otherwise, the result is just `x`.

`val context_inv : int -> context -> Term.t -> Term.t` `inv th s a` is `x` if there is `x = a` in the solution set for theory `th`; otherwise `Not_found` is raised.

`val context_use : int -> context -> Term.t -> Term.Set.t`

`use th s x` consists of the set of all term variables `y` such that `y = a` in `s`, and `x` is a variable `a`.

`val context_empty : unit -> context`

`context_empty()` represents the empty logical context.

`type status`

Inhabitants of type status are used as return values for `Ics.process`[C].

`val is_consistent : status -> bool`

`val is_redundant : status -> bool`

`val is_inconsistent : status -> bool`

`val d_consistent : status -> context`

In case `is_consistent st` holds, `d_consistent st` returns a new context.

`val process : context -> atom -> status`

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 an atom

- the atom `a` could be demonstrated to be inconsistent in `s`. In this case, `Ics.is_inconsistent`[C] holds of the result.
- the atom `a` could be demonstrated to be derivable in the context `s`. In this case, `Ics.is_redundant`[C] holds.
- Neither of the above holds. In this case, a modified context for representing the context of `s` conjoined with `a` is obtained using the destructor `Ics.d_consistent`[C].

Notice that a result `res` with `Ics.is_consistent`[C]`(res)` does not necessarily imply that atom `a` is indeed satisfiable, since the theory of ICS is indeed undecidable. Moreover, ICS includes a number of nonconvex theories, which requires case-splitting for completeness. `process` does not perform these case-splits in order to keep worst-case runtimes polynomial (with the notable exception of canonization of logical bitwise operators). Instead, it is in the responsibility of the application programmer to perform these splits; see also `Ics.split`[C].

`val split : context -> atom list`

Suggested case splits.

`val can : context -> term -> term`

Given a logical context `s` and an atom `a`, `can s a` computes a semicanonical 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.

`val cnstrnt : context -> term -> cnstrnt`

Given a logical context `s` and a term `a`, `cnstrnt s a` computes an arithmetic constraint for `a` in `s` using constraint information in `s` and abstraction interval interpretation. If no such constraint can be deduced, `None` is returned.

Propositions

```
type  prop
val prop_mk_true : unit -> prop
val prop_mk_false : unit -> prop
val prop_mk_var : name -> prop
val prop_mk_poslit : Atom.t -> prop
val prop_mk_neglit : Atom.t -> prop
val prop_mk_ite : prop -> prop -> prop -> prop
val prop_mk_conj : prop list -> prop
val prop_mk_disj : prop list -> prop
val prop_mk_iff : prop -> prop -> prop
val prop_mk_neg : prop -> prop
type  assignment
val prop_sat : context -> prop -> assignment option
```

Commands

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.

`val init : int * bool * string * inchannel * outchannel -> unit`

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 `^C^C`. The `init` function should be called before using any other function in this API.

`val cmd_rep : unit -> unit`

`cmd_rep` 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_batch : unit -> unit`

Similar to `Ics.cmd_rep`[C], but syntax error messages contain line numbers, and processing is aborted after state is unsatisfiable.

`val flush : unit -> unit`

Controls.

`val reset : unit -> unit`

`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.

`val gc : unit -> unit`

`gc()` triggers a full major collection of ocaml's garbage collector.

`val do_at_exit : unit -> unit`

`do_at_exit` clears out internal data structures.

`val sleep : int -> unit`

Sleeping for a number of seconds.

Tracing

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.

`val trace_reset : unit -> unit`

`trace_reset()` disables all tracing.

`val trace_add : string -> unit`

`trace_add str` enables tracing of functions associated with trace level `str`. For example, `trace_add "rule"` traces the calls for processing all generated equalities, disequalities, and constraints.

`val trace_remove : string -> unit`

`trace_remove str` removes `str` from the set of active trace levels

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

trace_get() returns the set of active trace levels.

Lists

```
val is_nil : 'a list -> bool
val cons : 'a -> 'a list -> 'a list
val head : 'a list -> 'a
val tail : 'a list -> 'a list
```

Pairs

```
val pair : 'a -> 'b -> 'a * 'b
```

`pair a b` builds a pair `(a,b)`.

```
val fst : 'a * 'b -> 'a
```

`fst p` returns `b` if `p` is equal to some `pair a _`.

```
val snd : 'a * 'b -> 'b
```

`snd p` returns `b` if `p` is equal to some `pair _ b`.

Triples

```
val triple : 'a -> 'b -> 'c -> 'a * 'b * 'c
val fst_of_triple : 'a * 'b * 'c -> 'a
val snd_of_triple : 'a * 'b * 'c -> 'b
val third_of_triple : 'a * 'b * 'c -> 'c
```

Quadruples

```
val fst_of_quadruple : 'a * 'b * 'c * 'd -> 'a
val snd_of_quadruple : 'a * 'b * 'c * 'd -> 'b
val third_of_quadruple : 'a * 'b * 'c * 'd -> 'c
val fourth_of_quadruple : 'a * 'b * 'c * 'd -> 'd
```

Option types

Options. An element of type `'a option` either satisfies the recognizer `is_some` or `is_none`. In case, `is_some` holds, a value of type `'a` can be obtained by `value_of`.

```
val is_some : 'a option -> bool
val is_none : 'a option -> bool
val value_of : 'a option -> 'a
```