ICS Manual (Version 2.0)

The ICS group

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

Contents

Contents **1**

1 Introduction **1**
1.1 Availability 3
1.2 Organization 3

2 Installation **3**

3 The ICS interactor **5**
3.1 ICS in action 5
3.2 The Command Language 11

4 Module `Ics` : Application programming interface. **19**

5 Calling ICS from Ocaml **44**

6 Calling ICS from C/C++ **45**

7 Calling ICS from Lisp **47**

A A Bakery Mutual Exclusion Protocol **48**

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 the combination of

`U`	uninterpreted functions,
`LA`	linear arithmetic (real and integer),
`NL`	power products (nonlinear arithmetic),
`P`	products,
`COP`	coproducts (direct sums),
`ARR`	functional arrays,
`BV`	fixed-sized bitvectors,
`PSET`	propositional sets, and
`APP`	functional abstraction and application.

This combination is particularly interesting for many applications in the realm of software and hardware verification, since the 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 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 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

- a C library,
- an Ocaml library, and
- a CommonLisp interface.

The C library API, for example, has been used to connect ICS with PVS, 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 ICS2.0 is available free of charge under a license agreement

http://ics.csl.sri.com/fm-license.pdf

with SRI. ICS is also an integral part of PVS 310. The complete sources and documentation of ICS are available at

ics.csl.sri.com

Binaries for many popular hardware architectures and operating systems including Linux, Mac OSX, Solaris, and Windows XP can also be found there.

1.2 Organization

This document describes the interfaces and implementation aspects of the ICS decision procedures.

2 Installation

Before trying to compile ICS on your prefered hardware architecture and operating system one might try one of the ICS binaries provided in the download section at ics.csl.sri.com. Compilations should only be necessary for developers or if ICS is used on a "nonstandard" platform.

Distribution. The file `ics2.0.tar.gz` can be downlowded from ics.csl.sri.com. Unpack this file using

```
> tar zxvf ics2.0.tar.gz
```

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

`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 propositional SAT solver (in C++)
`src/`	:	Source files for core ICS (in Ocaml and C)
`ics`	:	Shell script for invoking ICS interactor

Installation 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 3.06 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/`.

Installation.

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

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

   The `prefix` option specifies the path for installing ICS binaries (`prefix` defaults to `/usr/local/bin`. The optional `with-gmp` option is used to specify the path to a particular GMP library. Configure tries to find an appropriate `gmp` package, but this automatic search is somewhat unreliable and might fail on some computer systems. In this case, you have to locate an appropriate `gmp` and run configure with the `with-gmp` option.

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.

   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` manually.

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 processing 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 2.0 (Experimental, August 10 2003): Integrated Canonizer and Solver.
Copyright (c) 2003 SRI International.
Type 'help help.' for help about help, and 'Ctrl-d' to exit.

ics>
```

The '`ics>`' is the prompt and ICS is ready to interpret your commands. All commands are terminated by a '.'. Help about available commands and the syntax of input is obtained using the `help` command.

`help`	-	Display all commands
`help <term>`	-	Display definition of nonterminal `<term>`
`help assert`	-	Display description of command `assert`

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(x)))) = f(x).
:valid {f(x) = f(f(f(x)))}
```

In this case, the current state is unchanged. ICS also returns a subset of the asserted equalities, the so-called *justification*, from which the validity of the asserted atom follows. Such a justification is not necessarily minimal. The generation of dependencies can be disabled by using the `-dependencies` flag when calling the ICS interactor.

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).
:term v!1
:justification {}
ics> can f(f(f(f(x)))).
:term v!1
:justification {f(x) = f(f(f(x)))}
```

The second result of `can` is a justification for the equality between the argument and the resulting term. Since the canonical forms of these terms are identical in the current context, the equality `f(f(f(x))) = f(x)` holds indeed. Simplification of atoms is performed using the `simplify` command.

```
ics> simplify f(x) = f(f(f(f(x)))).
:atom tt
:justification {f(x) = f(f(f(x)))}
```

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.
:state
 v: [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. Let's first reset the current context to the empty context.

```
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.
:state
  v:[z |-> {v!2}]
  u: {v!2 = f(v!1)}
  la: {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 `y`, and deducing that `v!1` is equal to `z`. Now, `v!1` is replaced 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.
:state
  v:[z |-> {v!2}] with: [z |-> real]
```

```
u: {v!2 = f(z)}
la: {y = x + -1 * z}
```

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

```
ics> assert -y <> -(x - f(f(z))).
:unsat {-1 * x + f(f(z)) <> -1 * y, x = y + z, z = f(-1 * y + x)}
```

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.
 :term -1 * x + z
 :justification {x = y + z}
ics> can -(x - f(f(z))).
 :term -1 * x + z
 :justification {x = y + z, z = f(-1 * y + x)}
```

Besides arithmetic, ICS includes other theories such as the theory of products, functional 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 {y = 2 + x}
```

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 {-3 * cdr(x) + 2 * car(x) = f(cdr(x))}
```

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

```
ics> show.
 :state
   u: {v!3 = f(v!1)}
   la: {v!3 = 2 * v!2 + -3 * v!1}
   p: {v!1 = cdr(x),
   v!2 = car(x)}
```

So far, we have only dealt with equalities and disequalities, but constraints 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`.

```
ics> reset.
:unit
ics> assert x <= y + 2.
:ok s1
ics> show.
:state
  la: {y = -2 + x + k!1}
```

Now, the inequality `y <= z + 4` is rewritten as the nonnegativity constraint `6 + z + -1 * x + -1 * k!1 >= 0` and a new slack variable `k!2` is introduced to express this constraint in terms of an equality, which is solved for the largest variable `z`.

```
ics> assert y <= z + 4.
:ok s2
ics> show.
 :state la: {y = -2 + x + k!1, z = -6 + x + k!2 + k!1}
```

The inequality `z + 6 <= x` is reduced to the nonnegativity constraint `6 + z + -1 * x >= 0`, and `6 + z + -1 * x = k!3`, with `k!3` and the equality is solved and merged into state `s2` to obtain `s3`.

```
ics> assert z + 6 <= x.
:ok s3
```

```
ics> show.
:state
  la: {y = -2 + x + k!1, z = -6 + x + -1 * k!2}
```

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

```
ics> can x.
:val x
:justification {}

ics> can y + 2.
:val x
:justification {2 + y + -1 * x >= 0,  4 + z + -1 * y >= 0, -6 + -1 * z + x >= 0}

ics> can z + 6.
:val x
:justification {2 + y + -1 * x >= 0, 4 + z + -1 * y >= 0, -6 + -1 * z + x >= 0}
```

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.
 :symtab [
    empty |-> [];
   s1 |-> [2 + y + -1 * x >= 0];
   s2 |-> [4 + z + -1 * y >= 0; 2 + y + -1 * x >= 0];
   s3 |-> [-6 + -1 * z + x >= 0; 4 + z + -1 * y >= 0; 2 + y + -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, `&` is conjunction, and brackets `[`, `]` are used for grouping.

```
ics> sat [x = 1 | x = 4] & x > 5.
:unsat
```

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

```
ics> sat [x = 1 | x = 2 | x = 3] & x > 1.
:sat s5
:model [-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
:model [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. Each command is followed by a '.'.

Asserting facts.

```
assert [@<ident>] <atom>,...,<atom>
```

An `assert atm` adds the atom `atm` to the current context. 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. In addition, a *justification* in terms of an inconsistent subset of the current context is output if the generation of justifications is enabled.

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 that the result `:ok` does not necessarily mean that the database is indeed consistent, since the language accepted by ICS is undecidable. As long as one restricts oneself to a decidable fragment (such as the union of convex Shostak theories) of the ICS input language, `:ok` can be interpreted to mean *satisfiable*. For nonconvex theories such as functional arrays and linear integer arithmetic, the `split` command can be used for case splitting. In contrast, a result `:valid` indicates that the current atom is indeed valid in the current context, and `:unsat` indicates that the current context conjoined with the currently asserted atom is indeed unsatisfiable.

`assert@s atm` works as explained above but the atom is asserted to the context `s` in the current symbol table, and `assert@s atm1,...,atm` asserts the conjunction of atoms `atm1` to context `s`.
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 {v = f(v),  u = v, -1 + u = f(u)}
```

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.
:state v:[x |-> {y}] la: {x = 2}
ics> ctxt.
:atoms [x = y;  y = 2]
```

See Also: `symtab`3.2,

Canonization.

```
can <term>
```

For a term `t`, `can t` returns a term, which is a *canonical* representative of the equivalence class of `t` as induced by the atoms in the current context. If the generation of dependencies is enabled, then also a justification of the equality between `t` and `can t` is returned. There are no side effects.
See Also: `simplify`3.2

Simplification.

```
simplify <atom>
```

`simplify a` returns an atom equivalent to `b` in the current context. If proofmode is enabled, then, in addition, a justification of this equivalence is returned. *See Also:* `can`3.2

Logical Context.

```
ctxt [@<ident>]
```

`ctxt` returns the set of atoms asserted in the current logical context, and `ctxt@s` returns the set of asserted atoms in state 's' from the symbol table. These atoms are not necessarily in canonical form.

Term Definition.

```
def <ident> := <term>
```

Extend the symbol table with a definition `<ident>` for term `<term>`. In such a context, variable `<ident>` is always macro-expanded to `<term>`, but different occurrences of *term*are structure-shared. Also, `<ident>` may occur in `<term>`, since the expansion is not performed recursively.
Examples: `def x := y + z`

Disequalities.

```
diseq [@<ident>] <term>
```

Returns a list of variables known to be disequal to `<term>` in the context `<ident>` or the current context if `<ident>` is not specified. In addition, in proof generation mode, justifications for each disequality are returned

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 {}
```

See Also: `reset`3.2, `symtab`3.2, `forget`3.2

Finds in solution sets.

```
find [@<ident>] <th> <term>
```

If the equality `x = t` is, in the current context, in the equality set for theory `<th>`, 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.
See Also: `inv`3.2

Help.

```
help [<command> | < <nonterminal> >]
```

Help about ICS interactor commands and syntactic categories.
Examples.

`help`	Display all commands
`help help`	Display this message
`help <term>`	Display definition of nonterminal `<term>`
`help assert`	"Display description of command `assert`

Inverse find in solution sets.

```
inv [@<ident>] <th> <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`. 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.
See Also: `find`3.2

Definition of Propositions.

`prop` <ident> := <prop>

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 <ident>`

Updating the current logical state to be the state named by `ident` in the symbol table.
See Also: `symtab`3.2

Removing symbol table entries

`remove <ident>`

Remove the symbol table entry corresponding to `<ident>`.
See Also: `symtab`3.2

Saving the current logical context.

`save [<ident>]`

Adding a symbol table entry *var*for the current logical state.
See Also: `symtab`3.2, `forget`3.2,

Satisfiability Solver.

`sat [@<ident>] <prop>`

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 used 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 <ident> :  <sig>
```

Declare a variable <ident> to be interpreted over the set of bitvectors of width i or the integers or the reals. For example, after declaring sig x : int, every occurrence of the variable x is interpreted to mean the variable x{int}, that is the variable of name x with associated interpretation domain int. Notice that ICS treats y and y{int} as different variables. Bitvector variables have to be declared before use, when using infix operators, since context information is used for inferring parameters when applying infix bitvector operators.

Sigmatization.

```
sigma <term>
```

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.
See Also: can3.2, simplify3.2

Displaying the context.

```
show [@<ident>] [<th>]
```

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.

See Also: `ctxt`3.2,

Solving.

```
solve <th> <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 (`la`), tuples (`t`), bitvectors (`bv`), and propositional sets.

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

```
ics> solve la x + 2 = y - 3.
:subst [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).
:subst [x |-> cons(cons(u, v), cdr(x))]
```

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 [<ident>]`

`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.
:symtab
   empty |-> []
   s1 |-> [x = y]

ics> def x := y + z.
:unit
ics> prop z := a | b.
:unit
ics> symtab.
 :symtab [
   empty |-> [];
   x |-> z + y;
   s1 |-> [y = x];
   z |-> 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.
```

```
:symtab[
  empty |-> [];
  x |-> z + y;
  s1 |-> [y = x];
  z |-> a | b;  b |-> bitvector[2];
  q |-> int]
```

Trace.

```
trace <levels>
```

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`, `la`, can be used to trace updates on internal data structures.
See Also: `untrace`3.2,

Disable tracing.

```
untrace [<levels>]
```

Disable specified trace levels. If no trace levels are given, all tracing is disabled.
See Also: `trace`3.2,

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 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`[4], 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`[4] 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.

`val version : unit -> string`

Returns this ICS's version number.

Parameters

The following flags determine the current *configuration* of ICS.

`val set_profile : bool -> unit`

Enable profiling of used time and memory resources for selected functions. Used mainly for debugging.

`val set_pretty : string -> unit`

Determine pretty-printing.

- `mixfix` enables pretty-printing in mixfix and infix form,
- `prefix` disables mixfix and infix printing, and
- `sexpr` enables printing in terms of S-expressions of the form *(:op arg1 ... argn)*.

`val set_compactify : bool -> unit`

`set_compactify false` disables garbage collection of internally generated variables (default `true`).

`val set_assertion_frequency : int -> unit`

`set_assertion_frequency n` determines how often (frequency) the SAT solver sends (the relevant) information to ground decision procedures.

`val set_verbose : bool -> unit`

Using `set_verbose true`, the SAT solver reports all kinds of statistics and progress reports (default `false`).

`val set_remove_subsumed_clauses : bool -> unit`

Internal configuration of the SAT solver.

`val set_validate : bool -> unit`

With `set_validate` set to `true`, the SAT solver validates all generated assignments and all justifications for inconsistencies.

`val set_polarity_optimization : bool -> unit`

Internal configuration of the SAT solver.

`val set_clause_relevance : int -> unit`

Internal configuration of the SAT solver.

`val set_cleanup_period : int -> unit`
Internal configuration of the SAT solver.

`val set_num_refinements : int -> unit`
Internal configuration of the SAT solver.

`val set_statistic : bool -> unit`
Enable/Disable SAT solver to print statistics (default `false`).

`val set_show_explanations : bool -> unit`
Display explanations generated for SAT solver on `Format.err_formatter` when flag is enabled.

`val set_justifications : bool -> unit`
Print justifications of internally xgenerated facts (default `false`).

`val set_integer_solve : bool -> unit`
Enable/disable integer solver (default `true`). Disabling the integer solver makes the procedure incomplete, but (usually) faster.

`val set_proofmode : string -> unit`
ICS supports various proof modes.

- `No` disables generation of justifications
- `Dep` enables generation of dependencies (default).
- `Yes` enables generation of proof terms (disabled in ICS 2.0).

`val set_gc_mode : string -> unit`
Various settings for garbage collection

- `Lazy` delay garbage collection
- `Eager` garbage collection.

`val set_gc_space_overhead : int -> unit`
GC will work more if `space_overhead` is smaller (default 80).

`val set_gc_max_overhead : int -> unit`
Controlling heap compaction (default 500), `gc_max_overhead >= 1000000` disables compaction.

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

```
type  name
```

Representation of strings with constant equality test.

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

`name_of_string str` constructs a name `n` from a string such that `Ics.name_to_string[4](n)` yields `str`.

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

`name_to_string n` is the inverse operation of `Ics.name_of_string[4]`.

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

`name_eq n m` holds iff the corresponding strings `Ics.name_to_string[4](n)` and `Ics.name_to_string[4](m)` are equal. This equality test is constant in the length of strings.

Arithmetic domains

```
type  dom
```

Arithmetic domains

```
val dom_mk_int : unit -> dom
val dom_mk_real : unit -> dom
val dom_is_int : dom -> bool
val dom_is_real : dom -> bool
```

Theories

```
type  th
```

A **theory** is associated with each function symbol of terms.

- `u` Theory of uninterpreted function symbols.
- `la` Linear arithmetic theory.
- `p` Product theory.
- `bv` Bitvector theory.
- `cop` Coproducts.
- `nl` Power products.
- `app` Theory of function abstraction and application.
- `arr` Array theory.
- `pset` Theory of propositional sets

`val th_to_string : th -> string`

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

`val th_of_string : string -> th`

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

Function symbols

`type sym`

Representation of function symbols. Function symbols are partitioned into

- *uninterpreted* function symbols (of theory `u`) and
- *interpreted* function symbols from the theories `la`, `p`, `bv`, `cop`, `nl`, `cop`, `app`, `arr`, and `pset` above.

`val sym_theory_of : sym -> th`

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

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

`sym_eq` tests, in constant time, 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`[4]`(f, g)`, and
- a positive numbe, then `f` is said to be larger than `g`.

`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`[4]`(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`[4].

```
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`[4] 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`[4] does not hold.

Symbols of the **product theory** `p` consist of

- consing
- and first and second projections `car`, `cdr`.

```
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`.

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.

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` associated with it.

```
val sym_mk_apply : unit -> 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_mk_s : unit -> sym
```

`sym_mk_s()` constructs the symbol for the `S` combinator.

```
val sym_is_s : sym -> bool
```

`sym_is_s f` holds iff `f` represents the `S` combinator.

```
val sym_mk_k : unit -> sym
```

`sym_mk_s()` constructs the symbol for the `K` combinator.

```
val sym_is_k : sym -> bool
```

`sym_is_k f` holds iff `f` represents the `K` combinator.

```
val sym_mk_i : unit -> sym
```

`sym_mk_i()` constructs the symbol for the `I` combinator.

```
val sym_is_i : sym -> bool
```

`sym_is_i f` holds iff `f` represents the `I` combinator.

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 **propositional sets** include

- empty set
- full set
- conditional set.

`val sym_mk_empty : unit -> sym`
The empty set symbol

`val sym_is_empty : sym -> bool`
`sym_is_empty f` holds iff `f` represents the empty set symbol.

`val sym_mk_full : unit -> sym`
The full set symbol

`val sym_is_full : sym -> bool`
`sym_is_full f` holds iff `f` represents the full set symbol.

`val sym_mk_ite : unit -> sym`
The conditional set symbol

`val sym_is_ite : sym -> bool`
`sym_is_ite f` holds iff `f` represents the conditional set constructor.

Terms
Terms are either

- variables or

- applications of function symbols of type `Ics.sym`[4] 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`[4] 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`[4].

`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`[4]), 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`, and
- applications are ordered lexicographically using `Ics.sym_cmp`[4] 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`[4]`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.

Terms in this theory 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`.

Boolean Constants. are `true` and `false`.

`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 consist of

- injections `inj n`
- outjections `out n`.

`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.

Array terms are built up from

- constant arrays
- updates of arrays

- lookup of arrays.

`val term_mk_create : term -> term`

`term_mk_create a` represents an array with elements `a`.

`val term_mk_update : term -> term -> term -> term`

`term_mk_update a i x` represent an array `a` updated at position `i` with value `x`.

`val term_mk_select : term -> term -> term`

`term_mk_select a j` represents the value of array `a` at position `j`.

Nonlinear terms are sum-of-products with power products `a1^n1 * ... an^nk` with `ai` terms and `ni` integers at uninterpreted positions.

`val term_mk_mult : term -> term -> term`

`term_mk_mult a b` constructs a nonlinear term for representing the multiplication of `a` and `b`.

`val term_mk_multl : term list -> term`

`term_mk_multl [a1;...;an]` constructs a nonlinear term for representing the multiplication `a1 * ... * an`.

Function application

`val term_mk_apply : term -> term -> term`

`term_mk_apply a b` represents the application of `a`, viewed as a function, to the argument `b`.

`type terms`

Representation of a set of terms.

`val terms_of_list : term list -> terms`

Constructing a set of terms from a list of terms.

`val terms_to_list : terms -> term list`

Converting a set of terms into a list of terms.

Atoms

`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`.

`val atom_pp : atom -> unit`
Pretty-printing an atom to `stdout`.

`val atom_of_string : string -> atom`
Parsing a string to obtain an atom.

`val atom_to_string : atom -> string`
Printing an atom to a string.

`val atom_mk_true : unit -> atom`
Constructing the trivially true atom.

`val atom_mk_false : unit -> atom`
Constructing an unsatisfiable atom.

`val atom_mk_equal : term -> term -> atom`
`atom_mk_equal a b` constructs an atom for representing the equality between `a` and `b`.

`val atom_mk_diseq : term -> term -> atom`
`atom_mk_diseq a b` constructs an atom for representing the disequality of `a` and `b`.

`val atom_mk_le : term -> term -> atom`
`atom_mk_le a b` constructs an atom for representing `a <= b`.

`val atom_mk_lt : term -> term -> atom`
`atom_mk_lt a b` constructs an atom for representing `a < b`.

`val atom_mk_ge : term -> term -> atom`
`atom_mk_ge a b` constructs an atom for representing `a >= b`.

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

`atom_mk_gt a b` constructs an atom for representing `a > b`.

`val atom_negate : atom -> atom`

Constructs the negation of an atom.

Justifications

`type justification`

A *justification* is either

- a tag `Unjustified` or
- a set of context atoms.

`val justification_pp : justification -> unit`

Print a justification to `stdout`.

Processing

`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 -> atom list`

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

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

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

```
val context_apply :
  th -> context -> Term.t -> Term.t * justification
```

`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 :
  th -> context -> Term.t -> Term.t * justification
```

`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 : th -> 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 : th -> 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`[4]. There are three possible outcomes.

- `Redundant` implies the argument `a` in `Ics.process`[4]`s a` is valid in context `s`.
- `Inconsistent` implies the argument `a` conjoined with `s` in `Ics.process`[4]`s a` is inconsistent.
- `Consistent` neither a redundancy nor an inconsistency could be detected.

```
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 the extended 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`[4] holds of the result.
- the atom `a` could be demonstrated to be derivable in the context `s`. In this case, `Ics.is_redundant`[4] 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`[4].

Notice that a result `res` with `Ics.is_consistent`[4]`(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`[4].

`val split : context -> atom list`

Suggested case splits.

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

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 dom : context -> term -> dom * justification`

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.

Propositional logic

`type prop`

Representation of propositional formulas with propositional variables and atoms as literals. A propositional formual is either

- one of the propositional constants `tt`, `ff`
- a propositional variable `x`,
- a literal `l` with `l` an atom (atoms are closed under negation),
- a conjunction `p1 & ... & pn`,
- a disjunction `p1 | ... | pn`,
- a negation `~p`, or
- a let binding `let x := p in q`.

`val prop_pp : prop -> unit`

Printing a propositional formula.

`val prop_of_string : string -> prop`

Parsing a string to obtain a propositional formula. The syntax of propositional formulas is roughly given by the grammar above, and brackets `[]` are used for grouping. For details of the grammar see file `parser.mly`.

`val prop_to_string : prop -> string`

Pretty-print a propositional variable to a string.

`val prop_mk_true : unit -> prop`

The trivially true propositional formula.

`val prop_mk_false : unit -> prop`

The trivially false propositional formula.

`val prop_mk_var : name -> prop`

Constructing a propositional variable.

`val prop_mk_poslit : atom -> prop`

Injecting an atom into a propositional formula.

`val prop_mk_neglit : atom -> prop`

Injecting a negated atom into a propositional formula.

`val prop_mk_ite : prop -> prop -> prop -> prop`

`prop_mk_ite p q r` constructs a propositional formula equivalent to `prop_mk_disj (prop_mk_conj p q) (prop_mk_conj (prop_mk_neg p) r)`.

`val prop_mk_conj : prop list -> prop`

`prop_mk_conj [p1;...;pn]` constructs a representation of the conjunction of `p1 & ... & pn` with the empty list `[]` equivalent to `prop_mk_true()`.

`val prop_mk_disj : prop list -> prop`

`prop_mk_disj p q` constructs a representation of the disjunction of `p` and `q`.

`val prop_mk_iff : prop -> prop -> prop`

`prop_mk_iff p q` constructs a representation of the equivalence of `p` and `q`.

`val prop_mk_neg : prop -> prop`

`prop_mk_neg p` constructs a representation of the negation of `p`.

`val prop_mk_let : name -> prop -> prop -> prop`

`prop_mk_let x p q` constructs a structure-shared representation of the formula where `x` is replaced by `p` in `q`.

```
val prop_is_true : prop -> bool
```

Exactly one of the following recognizers is true for a propositional formula.

```
val prop_is_false : prop -> bool
val prop_is_var : prop -> bool
val prop_is_atom : prop -> bool
val prop_is_ite : prop -> bool
val prop_is_disj : prop -> bool
val prop_is_iff : prop -> bool
val prop_is_neg : prop -> bool
val prop_is_let : prop -> bool
val prop_d_var : prop -> name
```

If the corresponding recognizer above holds, propositional formulas may be destructured using the following.

```
val prop_d_atom : prop -> atom
val prop_d_ite : prop -> prop * prop * prop
val prop_d_disj : prop -> prop list
val prop_d_iff : prop -> prop * prop
val prop_d_neg : prop -> prop
val prop_d_let : prop -> name * prop * prop
type  assignment
```

Representation of assignments for propositional formulas.

```
val assignment_pp : assignment -> unit
```

Pretty-printing assignments.

```
val assignment_valuation : assignment -> (name * bool) list
```

Assignments to propositional variables.

```
val assignment_literals : assignment -> atom list
```

Assignment to nonpropositional literas.

```
val prop_sat : context -> prop -> assignment option
```

`prop_sat s p` determines if the propositional formula `p` is satisfiable in context `s`. It returns

- `None`, if `p` is unsatisfiable,

- `Some(ms)`, if `p` is satisfiable; in this case, `ms` implicitly represents a set of candidate models.

Imperative states

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 -> 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 set_outchannel : outchannel -> unit`

`val set_inchannel : inchannel -> unit`

`val set_prompt : string -> unit`

`val set_eot : string -> unit`

`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 : inchannel -> int`

Similar to `Ics.cmd_rep`[4], but syntax error messages contain line numbers, and processing is aborted after state is unsatisfiable.

`val flush : unit -> unit`

Flush currently active output channel.

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 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

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
```

5 Calling ICS from Ocaml

The following Ocaml program tries to asserts the trivial atom `5 <= 4` to the empty context using the `process` function in the interface and outputs the result to standard output.

```
open Ics

let main () =
  let c = context_empty () in
  let a =
    atom_mk_le
      (term_mk_num (num_of_int 5)) (term_mk_num (num_of_int 4)) in
  let s = process c a in

  begin if is_consistent s then
    print_string "Consistent"
  else if is_inconsistent s then
    print_string "Inconsistent"
  else if is_redundant s then
    print_string "Redundant"
  else
    failwith "Error"
  end;
  print_newline ();
```

```
  Pervasives.flush Pervasives.stdout ;;

main () ;;
```

Given that this program is stored in file `test.ml`, it is compiled with

```
$ ocamlopt -I <icspath>/lib/i686-pc-linux-gnu/ -c test.ml
$ ocamlopt -I <icspath>/lib/i686-pc-linux-gnu/ -o test unix.cmxa ics.cmxa test.cmx
```

So the only things needed are to give the Ocaml compiler the path to the library `-I <path>` and the linker the ICS library itself plus `unix.cmxa` as the Ocaml library `unix` is used by ICS. Notice that libraries for different platforms are distributed with ICS, and in the above we assume the `i686-pc-linux-gnu` architecture. The architecture name can also be obtained using `config.guess`.

Now, the `test` program can be run to get the not too unexpected result.

```
$ ./test
Inconsistent
```

6 Calling ICS from C/C++

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.

Calls to the C functions in the interface must obey the typing restrictions of Ocaml, otherwise the result is undefined (typically, the program crashes). For example, the function `ics_term_cmp` may only be called with two arguments representing term values, since the signature of this function in the interface is given as `term -> term -> int`.

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 ICS library `libics.a` can be found in the linking stage.

Appendix A contains an implementation of a bounded model checker for the Bakery mutual exclusion protocol in `C++`. Assuming that the name of the corresponding file is `bakery.cpp`, then this program can be compiled on a Linux platform using the static library `libics.a` with the following command:

```
g++ -o bakery -L $ICSPATH/lib/i686-pc-linux-gnu/ -I $ICSPATH/obj/i686-pc-linux-gnu/ -l
```

Here, `ICSPATH` is assumed to be set to the the ICS home directory which contains ...

[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.

7 Calling ICS from Lisp

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. In order to use ICS in Lisp, the shared object file `libicsall.so` has to be loaded followed by loading the foreign function interface.

```
> (load "./lib/i686-pc-linux-gnu/libicsall.so")
 ; Foreign loading lib/i686-pc-linux-gnu/libicsall.so.
t
> (load "./include/ics.lisp")
t
```

Now, all the functions in the ICS interface are available in Lisp. It is the Lisp programmer's responsibility to call the ICS functions in a type-correct way. Calls to ICS functions violating the Ocaml type discipline may have fatal consequences for the Lisp image. The ICS data structures can be garbage collection using the Lisp garbage collector using finalization on wrappers of ICS pointers. In the following, the Allegro Lisp function `excl:schedule-finalization` directs the Lisp garbage collector to call `wrap-free!` when garbage collecting the wrapper, and the function `wrap-free!` calls the ICS deregistration function on the unwrapped ICS structure.

```
(defstruct (wrap
    (:predicate wrap?)
```

```
    (:constructor make-wrap (address))
    (:print-function
     (lambda (p s k)
       (declare (ignore k))
               (format t "<#wrap: ~a>" (wrap-address p)))))
  address)

(defun wrap-finalize! (w)
  (excl:schedule-finalization w 'wrap-free!))

(defun wrap-free! (w)
  (ics_deregister (unwrap w)))
```

In this way it is ensured that the Lisp and the Ocaml garbage collector cooperate as long as every ICS wrapper has been finalized. A typical construction is demonstrated below.

```
(defun ics-empty-state ()
   (let ((empty (make-wrap (ics_context_empty))))
      (wrap-finalize! empty)
       empty))
```

The empty ICS context is obtained using `ics_context_empty`, and the corresponding Lisp wrapper `empty` is finalized before being returned by this function.

ICS errors and exceptions are being handled through the Lisp exception mechanism, and ICS functions are interruptable using `Ctrl-C Ctrl-C`.

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.

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.

  Comiling using static library:
     g++ -o bakery -L ../lib/i686-pc-linux-gnu/ \
                   -I ../obj/i686-pc-linux-gnu/ \
                   -lics bakery.cpp

  Compiling using dynamic library:
     g++ -o bakery -L ../lib/i686-pc-linux-gnu/ \
                   -I ../obj/i686-pc-linux-gnu/ \
                   -licsall -lgmp bakery.cpp
***/
#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;

  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);
}

}
```