SRI International®

August 23, 2023

Yices Manual
Version 2.6.4

Bruno Dutertre

Computer Science Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 859-2000 • Facsimile: (650) 859-2844

Contents

Chapter 1

Introduction

This manual is an introduction to the logic, language, and architecture of the Yices 2 SMT solver. Yices is developed at SRI International's Computer Science Laboratory. Since version 2.5.3, Yices is released under the GNU General Public License version 3 (reproduced in Appendix A). Previous versions were released under different terms, and were free-of-charge for non-commercial use.

To discuss alternative license terms, please contact us at `fm-license@csl.sri.com`.

1.1 Download and Installation

The latest stable version of Yices 2 can be downloaded at https://yices.csl.sri.com. We provide pre-compiled binaries for the platforms and operating systems listed in Table 1.1. We also provide source code there. For MacOS and Linux, you can also install Yices 2 using package managers (i.e., homebrew for the Mac and apt for Debian/Ubuntu).

For the latest developments, you can clone our Git repository https://github.com/SRI-CSL/yices2.

OS/Hardware	**Notes**
Linux 64 bits	Kernel 2.6.24 or more recent
Mac OS X 64 bits	Mac OS X El Capitan
Windows (64 bits)	

Table 1.1: Binary Distributions

1.1.1 Binary Distributions

To download stable Yices 2 binaries, go to https://yices.csl.sri.com and select the distribution that you want to install. Untar or unzip the file and follow the instructions in the included README file. The binary distributions are self-contained and do not require installation of third-party libraries.

To complete installation on Linux or Mac OS X, the binary distributions include a shell script called `install-yices`. By default, this script installs Yices in `/usr/local`. If this is fine for you, type

```
sudo ./install-yices
```

This will install the binaries in `/usr/local/bin`, the library in `/usr/local/lib`, and the header files in `/usr/local/include`.

To install Yices in a different location, you can type

```
./install-yices <directory>
```

(use `sudo` if necessary).

Homebrew Package

If you use homebrew on Mac OS X, you can easily install Yices as follows:

```
brew install SRI-CSL/sri-csl/yices
```

(you may need `sudo`). This will install the Yices 2 executables, library, and include files.

Debian Package

For Ubuntu or Debian (or any other Linux distribution that uses `apt`), we provide APT packages. To install them, you must first add our PPA to your list of repositories then install package `yices2`:

```
sudo add-apt-repository ppa:sri-csl/formal-methods
sudo apt-get update
sudo apt-get install yices2
```

You can also install the library and development files as follows:

```
sudo apt-get install yices2-dev libyices2.6
```

1.1.2 Source Distribution

The source distribution must be used for operating systems not listed in Table 1.1 (or for old versions of Linux or Mac OS X). It is also useful if you desire to compile Yices with

debugging information, if you want to link Yices with your own version of the GMP library, or if your want to build a thread-safe safe version. The source is available as a tarfile at https://yices.csl.sri.com and on our git repository at https://github.com/SRI-CSL/yices2.

Several optional features can be selected at compilation time:

- MCSAT solver (required for non-linear arithmetic)
- Support for third-party backend SAT solvers
- Support for a thread-safe API.

Instructions for building Yices with these different features are given in Chapter 2.

1.2 Content of the Distributions

The binary distributions and packages include the Yices executables, the Yices library and header files, and examples and documentation. Four solvers are currently included:

- `yices` is the main SMT solver. It can read and process input given in Yices 2's specification language. This language is explained in Chapter 5.
- `yices-smt` is a solver for input in the SMT-LIB 1.2 notation [RT06].
- `yices-smt2` is a solver for input in the SMT-LIB 2.x notation [BFT15].
- `yices-sat` is a Boolean satisfiability solver that can read input in the DIMACS CNF format.

The library and header files allow you to use Yices via its API, as explained in Chapter 7.

The source distribution includes source code for the above four solvers and for the library. It also includes documentation for the source, more examples and regression tests, various scripts and utilities, and the LaTeX source for this manual.

1.3 Language Bindings

We currently provide wrappers to the Yices API for Python, Java, Go and OCaml. Source code for these different wrappers is maintained on different GitHub repositories.

Python: https://github.com/SRI-CSL/yices2_python_bindings. The easiest way to install these Python bindings is to use `pip`:

```
pip install yices
```

Java: https://github.com/SRI-CSL/yices2_java_bindings. The bindings can be installed with `ant` or with a dedicated shell script. Consult the repository's README for details.

Go: https://github.com/SRI-CSL/yices2_go_bindings. The code can be installed with

```
go get github.com/SRI-CSL/yices2_go_bindings/cmd/yices_info
```

OCaml: https://github.com/SRI-CSL/yices2_ocaml_bindings. Follow the instructions in this repository for building and using the OCaml bindings.

1.4 Supported Logics

The current Yices 2 release supports quantifier-free combinations of linear and non-linear integer and real arithmetic, uninterpreted function, arrays, and bitvectors. Currently, Yices 2 supports most SMT-LIB logics that do not involve quantifiers as summarized in Table 1.2. Since version 2.6.4, Yices 2 also supports the UF theory (i.e., first order logic with equality)/ The meaning of the logics and theories in this table is explained at the SMT-LIB website (http://www.smtlib.org). In addition, Yices 2 supports a more general set of array operations than required by SMT-LIB, and Yices 2 has support for tuple and enumeration types, which are not part of SMT-LIB.

Logic	Description	Supported
ALIA	Arrays, Linear Integer Arithmetic, Quantifiers	no
AUFLIA	Arrays, Linear Integer Arithmetic, Quantifiers, Uninterpreted Functions	no
AUFLIRA	Arrays, Mixed Linear Arithmetic, Quantifiers, Uninterpreted Functions	no
AUFNIRA	Arrays, Nonlinear Arithmetic, Quantifiers, Uninterpreted Functions	no
LIA	Linear Integer Arithmetic, Quantifiers	no
LRA	Linear Real Arithmetic, Quantifiers	no
NIA	Nonlinear Integer Arithmetic, Quantifiers	no
NRA	Nonlinear Real Arithmetic, Quantifiers	no
QF_ABV	Arrays and Bitvectors	yes
QF_ALIA	Arrays and Linear Integer Arithmetic	yes
QF_AUFBV	Arrays, Bitvectors Uninterpreted Functions	yes
QF_AUFLIA	Arrays, Linear Integer Arithmetic, Uninterpreted Functions	yes
QF_AX	Arrays (with extensionality)	yes
QF_BV	Bitvectors	yes
QF_IDL	Integer Difference Logic	yes
QF_LIA	Linear Integer Arithmetic	yes
QF_LIRA	Mixed Linear Arithmetic	yes
QF_LRA	Linear Real Arithmetic	yes
QF_NIA	Nonlinear Integer Arithmetic	yes
QF_NIRA	Mixed Nonlinear Arithmetic	yes
QF_NRA	Nonlinear Real Arithmetic	yes
QF_RDL	Real Difference Logic	yes
QF_UF	Uninterpreted Functions	yes
QF_UFBV	Uninterpreted Functions, Bitvectors	yes
QF_UFIDL	Uninterpreted Functions, Integer Difference Logic	yes
QF_UFLIA	Uninterpreted Functions, Linear Integer Arithmetic	yes
QF_UFLRA	Uninterpreted Functions, Linear Real Arithmetic	yes
QF_UFNIA	Uninterpreted Functions, Nonlinear Integer Arithmetic	yes
QF_UFNIRA	Uninterpreted Functions, Mixed Nonlinear Arithmetic	yes
QF_UFNRA	Uninterpreted Functions, Nonlinear Real Arithmetic	yes
UF	Quantifiers, Uninterpreted Functions	yes
UFLRA	Nonlinear Real Arithmetic, Quantifiers, Uninterpreted Functions	no
UFNIA	Nonlinear Integer Arithmetic, Quantifiers, Uninterpreted Functions	no

Table 1.2: Logics Supported by Yices 2

```
From: ...
Subject: Yices 1.0.36 segfault
To: yices-bugs@csl.sri.com

Hi,

I am experiencing a segmentation fault from Yices. I have attached
a small test case that causes the crash. I am using Yices 1.0.36 on
x86_64 statically linked against GMP on Ubuntu 12.04.
...
```

Figure 1.1: Good Bug Report

1.5 Getting Help and Reporting Bugs

The Yices website provides the latest release and information about Yices. The easiest (and preferred) way to report a bug or ask a question about Yices is to post an issue on our GitHub repository (https://github.com/SRI-CSL/yices2).

Alternatively, you can contact us via the Yices mailing lists:

- Send e-mail to `yices-help@csl.sri.com` if you have questions about Yices usage or installation.

  This mailing list is moderated, but you do not need to register to post to it. You can register to this mailing list if you are interested in helping others.

- To report a bug, you can send an e-mail to `yices-bugs@csl.sri.com`.

If you report a bug, please include enough information in your report to enable us to reproduce and fix the problem. Figure 1.1 shows what a good bug report looks like. This example is an edited version of real bug report that we actually received (with private information removed). Figure 1.2 shows an example of poor bug report. This example is fictitious but representative of what we sometimes receive on our mailing list.

Please try to use Figure 1.1 as a template and include answers to the following questions:

- Which version of Yices are you using?
- On which hardware and OS?
- How can we reproduce the bug? If at all possible send an input file or program fragment.

```
From: ...
Subject: Segmentation fault
To: yices-bugs@csl.sri.com

I have just downloaded Yices. After I compile my code and link it
with Yices, there is a segmentation fault when I run the executable.

Can you help?

Thanks,
...
```

Figure 1.2: Poor Bug Report

```
From: ...
Subject: Invitation to Connect on LinkedIn
To: yices-bugs@csl.sri.com

I'd like to add you to my professional network on LinkedIn.

...
```

Figure 1.3: Terrible Bug Report

Chapter 2

Building Yices 2 from Source

If you download the Yices 2 source, you can choose different optional components and features at compilation time. The main options are

- Support for the MCSAT solver (which is necessary for non-linear arithmetic).
- Support for third-party backend SAT solvers (which can provide improved performance on bitvector problems).
- Thread-safe version of the Yices library.

We start with the simplest type of build that does not include any optional components. We then explain how to build the optional components. The instructions are written for a Debian-style Linux distribution such as Ubuntu, but they should work with minor adjustments on other Unix variants.

2.1 Basic Build

Yices 2 is straightforward to compile on UNIX-like systems. Any recent version of GCC or Clang should work. The compilation uses standard tools such as `GNU make` and `sed`. It also requires the `gperf` utility and the GMP library. On many systems, `gperf` and GMP can be installed using package managers. For example, on Ubuntu:

```
sudo apt-get install libgmp-dev
sudo apt-get install gperf
```

After this, compiling and installing Yices use standard steps.

From A Source Tarfile

If you downloaded the Yices 2 source from https://yices.csl.sri.com. Follow these steps:

```
tar xvf yices-2.6.2-src.tgz
cd yices-2.6.2
./configure
make -j
make check
```

This will build binaries and libraries, and run the regression tests. If all goes well, you can then install Yices in `/usr/local` with

```
sudo make install
```

You can change the installation location by giving a `--prefix` option to `configure`.

From The Git Repository

You can also get the latest source from https://github.com/SRI-CSL/yices2 and build Yices as follows:

```
git clone https://github.com/SRI-CSL/yices2
cd yices2
autoconf
./configure
make -j
make check
```

As before, you can then install Yices in `/usr/local` with

```
sudo make install
```

2.2 MCSAT Support

Yices includes a solver for nonlinear arithmetic based on the Model Constructing Satisfiability Calculus (MCSAT). This calculus and its application to nonlinear arithmetic are explained in [JBdM13] and [dMJ12].

The precompiled, binary distributions of Yices include the MCSAT solver and can process nonlinear arithmetic problems. If you build Yices from source and want support for nonlinear arithmetic, you must install two external libraries: LIBPOLY [JD17] and CUDD [Som98] and enable MCSAT when building Yices.

LIBPOLY

The LIBPOLY source is available on GitHub at https://github.com/SRI-CSL/libpoly. Make sure that you download the latest version of libpoly. Yices 2.6 requires libpoly v0.1.3. Follow the instruction in libpoly's README.md to compile and install it.

CUDD

We recommend downloading CUDD from the GitHub repository https://github.com/ivmai/cudd and building it as follows:

```
git clone https://github.com/ivmai/cudd
cd cudd
./configure CFLAGS=-fPIC
make
sudo make install
```

This will install CUDD header files and libraries in `/usr/local`.

Note

The CUDD Makefile was created with `automake-1.14`. Compilation may fail if you have a different version of automake on your system with this error:

```
cudd/build-aux/missing: line 81: automake-1.14: command not found
WARNING: 'automake-1.14' is missing on your system.
...
```

If this happens to you, try this

```
aclocal
automake
```

Another fix is to edit the Makefile and replace '1.14' by your version of automake and aclocal.

Enabling MCSAT Support in Yices

Once you have installed LIBPOLY and CUDD, you can compile Yices with MCSAT support as follows:

```
./configure --enable-mcsat
make -j
sudo make install
```

The configure scripts will check that CUDD and LIBPOLY are present on your system, The usual environment variables (e.g., `CPPFLAGS` and `LDFLAGS`) can be used if you install libpoly or CUDD in a non-standad location.

2.3 Third-Party SAT Solvers

It is now possible for Yices to use third-party backend SAT solvers for bitvector solving. Currently, we support three SAT solvers: CaDiCaL [Bie19], CryptoMiniSat [SNC09], and Kissat [BFFH20]. You can compile Yices with support for all or a subset of these solvers.

CaDiCaL

Here are the steps for downloading and installing CaDiCaL:

1. Clone the CaDiCaL repository:

```
git clone https://github.com/arminbiere/cadical
```

2. Run the CaDiCaL configure script with option `-fPIC`. If you're using `bash`, the following should work:

```
cd cadical
CXXFLAGS=-fPIC ./configure
```

3. Build the code and (optionally) run the tests

```
make
make test
```

4. Install the library and header file:

```
sudo install build/libcadical.a /usr/local/lib
sudo install -m644 src/ccadical.h /usr/local/include
```

CryptoMiniSat

Yices requires a patched version of CryptoMiniSat 5 that we provide at https://github.com/BrunoDutertre/cryptominisat. Here is how you can download and build it.

1. Clone the repository:

```
git clone https://github.com/BrunoDutertre/cryptominisat
```

2. Install CryptoMiniSat's dependencies

```
sudo apt-get install cmake zlib1g-dev \
          libboost-program-options-dev
```

3. Compile and install:

```
cd cryptominisat
mkdir build
cd build
cmake .. -DENABLE_PYTHON_INTERFACE=OFF
make
sudo make install
```

Kissat

We provide a patched version of Kissat that fixes an issue. Download this patched version at https://github.com/BrunoDutertre/kissat. The original is at https://github.com/arminbiere/kissat. To compile the code, follow these instructions:

1. Clone the repository:

```
git clone https://github.com/BrunoDutertre/kissat
```

2. Run the configure script with option `-fPIC`:

```
cd kissat
./configure -fPIC
```

3. Build the code

```
make
```

4. Install the library and header files in a convenient location (such as `/usr/local`):

```
sudo install build/libkissat.a /usr/local/lib
sudo install -m644 src/kissat.h /usr/local/include
```

Configure Yices 2

If all third-party SAT solver are installed, you can configure Yices 2 as follows to use them:

```
./configure CPPFLAGS='-DHAVE_CADICAL -DHAVE_CRYPTOMINISAT -DHAVE_KISSAT' \
     LIBS='-lcryptominisat5 -lcadical -lkissat -lstdc++ -lm'
```

After this, you can build and install Yices as usual:

```
make -j
make check
sudo make install
```

If you want only CaDiCaL, use the following configure command instead

```
./configure CPPFLAGS=-DHAVE_CADICAL \
     LIBS='-lcadical -lstdc++ -lm'
```

If you want only CryptoMiniSat, you can use

```
./configure CPPFLAGS=-DHAVE_CRYPTOMINISAT \
     LIBS='-lcryptominisat5 -lstdc++'
```

If you want only Kissat, use this command:

```
./configure CPPFLAGS=-DHAVE_KISSAT \
     LIBS='-kissat -lm'
```

Compilation with these backend SAT solver is compatible with MCSAT, so you add option `--enable-mcsat` to any of these configure commands.

2.4 Thread-Safe API

By default, the Yices library is not re-entrant and it cannot be used in multi-threaded applications. If you need a re-entrant version of the library, you can configure and build Yices as follows:

```
./configure --enable-thread-safety
make
sudo make install
```

When configured in this fashion, the Yices library will allow multiple threads to manipulate separate contexts and models without causing race conditions (see Chapter 7). Sharing of contexts or models across several threads is not supported (unless you implement your own locking mechanism).

In the current version (Yices 2.6.4), threat-safety and MCSAT are not compatible. It is not possible to build Yices to support MCSAT and be re-entrant.

2.5 Building for Windows

For Windows, we recommend building Yices using Cygwin. If you want a version that works natively on Windows (i.e., does not depend on the Cygwin DLLs) then you can compile from Cygwin using the MinGW cross-compilers. The file `doc/COMPILING` included in the source distribution gives more details.

2.6 Manual and Documentation

The LaTeX source for this manual is included in the Yices 2 repository. To build the manual, make sure that you have a Latex installation (including `pdflatex`) and the `latexmk` utility. On Ubuntu, you can install them with `apt-get`:

```
sudo apt-get install texlive latexmk
```

With these tools installed, you can generate the manual by typing

```
make doc
```

in the top-level Yices source directory.

The repository also includes detailed API documentation that can be built using the Sphinx tool.[1] The generated documentation can be browsed online at https://yices.csl.sri.com/doc/index.html. You can also build a local version of this documentation as follows.

1. Install Sphinx:

```
pip install sphinx
```

2. Build the `html` documentation (from the Yices top-level source directory):

```
cd doc/sphinx
make html
```

   The resulting documentation will be in directory `doc/sphinx/build/html`.

Sphinx can generate documentation in other formats than `html`. For example, you can do

```
cd doc/sphinx
make epub
```

This will generate an electronic book in the `epub` format. This book is in a single file `Yices.epub` in directory `doc/sphinx/build/epub`.

[1] See https://www.sphinx-doc.org/en/master/

Chapter 3

Yices 2 Logic

Yices 2 specifications are written in a typed logic. The language is intended to be simple enough for efficient processing by the tool and expressive enough for most applications. The Yices 2 language is similar to the logic supported by Yices 1, but the most complex type constructs have been removed.

3.1 Type System

Yices 2 has a few built-in types for primitive objects:

- the arithmetic types `int` and `real`
- the Boolean type `bool`
- the type `(bitvector k)` of bitvectors of size k, where k is a positive integer.

All these built-in types are *atomic*. The set of atomic types can be extended by declaring new *uninterpreted types* and *scalar types*. An uninterpreted type denotes a nonempty collection of objects with no cardinality constraint. A scalar type denotes a nonempty, *finite* set of objects. The cardinality of a scalar type is defined when the type is created.

In addition to the atomic types, Yices 2 provides constructors for tuple and function types. The set of all Yices 2 types can be defined inductively as follows:

- Any atomic type τ is a type.
- If $n > 0$ and $\sigma_1, \ldots, \sigma_n$ are n types, then $\sigma = (\sigma_1 \times \ldots \times \sigma_n)$ is a type. Objects of type σ are tuples $(x_1, \ldots, x_n)$ where x_i is an object of type σ_i.
- If $n > 0$ and $\sigma_1, \ldots, \sigma_n$ and τ are types, then $\sigma = (\sigma_1 \times \ldots \times \sigma_n \rightarrow \tau)$ is a type. Objects of type σ are functions of domain $\sigma_1 \times \ldots \times \sigma_n$ and range τ.

By construction, all the types are nonempty. Yices does not have a specific type constructor for arrays since the logic does not distinguish between arrays and functions. For example, an array indexed by integers is simply a function of domain `int`.

Yices 2 uses a simple form of subtyping. Given two types σ and τ, let $\sigma \sqsubset \tau$ denote that σ is a subtype of τ. Then the subtype relation is defined by the following rules:

- $\tau \sqsubset \tau$ (any type is a subtype of itself)
- $\texttt{int} \sqsubset \texttt{real}$ (the integers form a subtype of the reals)
- If $\sigma_1 \sqsubset \tau_1, \ldots, \sigma_n \sqsubset \tau_n$ then $(\sigma_1 \times \ldots \times \sigma_n) \sqsubset (\tau_1 \times \ldots \times \tau_n)$.
- If $\tau \sqsubset \tau'$ then $(\sigma_1 \times \ldots \times \sigma_n \to \tau) \sqsubset (\sigma_1 \times \ldots \times \sigma_n \to \tau')$.

For example, the type $(\texttt{int} \times \texttt{int})$ (pairs of integers) is a subtype of $(\texttt{real} \times \texttt{real})$ (pairs of reals).

Two types, τ and τ', are said to be *compatible* if they have a common supertype, that is, if there exists a type σ such that $\tau \sqsubset \sigma$ and $\tau' \sqsubset \sigma$. If that is the case, then there exists a unique minimal supertype among all the common supertypes. We denote the minimal supertype of τ and τ' by $\tau \sqcup \tau'$. By definition, we then have

$$\tau \sqsubset \sigma \text{ and } \tau' \sqsubset \sigma \;\Rightarrow\; \tau \sqcup \tau' \sqsubset \sigma.$$

For example, the tuple types $\tau = (\texttt{int} \times \texttt{real} \times \texttt{int})$ and $\tau = (\texttt{int} \times \texttt{int} \times \texttt{real})$ are compatible. Their minimal supertype is $\tau \sqcup \tau' = (\texttt{int} \times \texttt{real} \times \texttt{real})$. The type $(\texttt{real} \times \texttt{real} \times \texttt{real})$ is also a common supertype of τ and τ' but it is not minimal.

3.2 Terms and Formulas

In Yices 2, the atomic terms include the Boolean constants (`true` and `false`) as well as arithmetic and bitvector constants.

When a scalar type τ of cardinality n is declared, n distinct constant $c_1, \ldots, c_n$ of type τ are also implicitly defined. In the Yices 2 syntax, this is done via a declaration of the form:

```
(define-type tau (scalar c1 ... cn))
```

An equivalent functionality is provided by the Yices API. The API allows one to create a new scalar type and to access n constants of that type indexed by integers between 0 and $n-1$ (check file `include/yices.h` for explanations).

The user can also declare *uninterpreted constants* of arbitrary types. Informally, uninterpreted constants of type τ can be considered like global variables, but Yices (in particular the Yices API) makes a distinction between *variables* of type τ and *uninterpreted constants*

of type τ. In the Yices API, variables are used to build quantified expressions and to support term substitutions. Free variables are not allowed to occur in assertions.

The term constructors include the common Boolean operators (conjunction, disjunction, negation, implication, etc.), an if-then-else constructor, equality, function application, and tuple constructor and projection. In addition, Yices provides an `update` operator that can be applied to arbitrary functions. The type-checking rules for these primitive operators are described in Figure 3.1, where the notation $t :: \tau$ means "term t has type τ".

There are no separate syntax or constructors for formulas. In Yices 2, a formula is simply a term of Boolean type.

The semantics of most of these operators is standard. The update operator for functions is characterized by the following axioms[1]:

$$\begin{aligned} ((\texttt{update}\ f\ t_1 \ldots t_n\ v)\ t_1 \ldots t_n) &= v \\ u_1 \neq t_1 \vee \ldots \vee u_n \neq t_n \Rightarrow ((\texttt{update}\ f\ t_1 \ldots t_n\ v)\ u_1 \ldots u_n) &= (f\ u_1 \ldots u_n) \end{aligned}$$

In other words, $(\texttt{update}\ f\ t_1 \ldots t_n\ v)$ is the function equal to f at all points except $(t_1, \ldots, t_n)$. Informally, if f is interpreted as an array then the update corresponds to "storing" v at position $t_1, \ldots, t_n$ in the array. Reading the content of the array is nothing other than function application: $(f\ i_1 \ldots i_n)$ is the content of the array at position $i_1, \ldots, i_n$.

The full Yices 2 language has a few more operators not described here, and it includes existential and universal quantifiers. We do not describe the type-checking rules for quantifiers here since Yices 2 has limited support for quantified formulas at this point.

3.3 Theories

In addition to the generic operators presented previously, the Yices language includes the standard arithmetic operators and a rich set of bitvector operators.

3.3.1 Arithmetic

Arithmetic constants are arbitrary precision integers and rationals. Although Yices uses exact arithmetic, rational constants can be written in floating-point notation. Internally, Yices converts floating-point input to rationals. For example, the floating-point expression `3.04e-1` is converted to $38/125$.

The Yices language supports the traditional arithmetic operators (i.e., addition, subtraction, multiplication). The solver for non-linear arithmetic supports arbitrary division. The linear-arithmetic solver is limited to division by non-zero constants. For example, the expression

[1] These are the main axioms of the McCarthy theory of arrays.

Boolean Operators

$$\frac{t :: \texttt{bool}}{(\texttt{not}\ t) :: \texttt{bool}} \qquad \frac{t_1 :: \texttt{bool} \quad t_2 :: \texttt{bool}}{(\texttt{implies}\ t_1\ t_2) :: \texttt{bool}}$$

$$\frac{t_1 :: \texttt{bool} \ldots t_n :: \texttt{bool}}{(\texttt{or}\ t_1 \ldots t_n) :: \texttt{bool}} \qquad \frac{t_1 :: \texttt{bool} \ldots t_n :: \texttt{bool}}{(\texttt{and}\ t_1 \ldots t_n) :: \texttt{bool}}$$

Equality

$$\frac{t_1 :: \tau_1 \quad t_2 :: \tau_2}{(t_1 = t_2) :: \texttt{bool}} \text{ provided } \tau_1 \text{ and } \tau_2 \text{ are compatible}$$

If-then-else

$$\frac{c :: \texttt{bool} \quad t_1 :: \tau_1 \quad t_2 :: \tau_2}{(\texttt{ite}\ c\ t_1\ t_2) :: \tau_1 \sqcup \tau_2} \text{ provided } \tau_1 \text{ and } \tau_2 \text{ are compatible}$$

Tuple Constructor and Projection

$$\frac{t_1 :: \tau_1 \ldots t_n :: \tau_n}{(\texttt{tuple}\ t_1 \ldots t_n) :: (\tau_1 \times \ldots \times \tau_n)} \qquad \frac{t :: (\tau_1 \times \ldots \times \tau_n)}{(\texttt{select}_i\ t) :: \tau_i}$$

Function Application

$$\frac{f :: (\tau_1 \times \ldots \times \tau_n \to \tau) \quad t_1 :: \sigma_1 \ldots t_n :: \sigma_n \quad \sigma_1 \sqsubset \tau_1 \ldots \sigma_n \sqsubset \tau_n}{(f\ t_1 \ldots t_n) :: \tau}$$

Function Update

$$\frac{f :: (\tau_1 \times \ldots \times \tau_n \to \tau) \quad t_1 :: \sigma_1 \ldots t_n :: \sigma_n \quad v :: \sigma \quad \sigma_i \sqsubset \tau_i \quad \sigma \sqsubset \tau}{(\texttt{update}\ f\ t_1 \ldots t_n\ v) :: (\tau_1 \times \ldots \times \tau_n \to \tau)}$$

Figure 3.1: Primitive Operators and Type Checking

$(x + 4y)/3$ is allowed in linear arithmetic, but $3/(x + 4y)$ is not. The arithmetic predicates are the usual comparison operators, including both strict and nonstrict inequalities.

We've added more arithmetic operations since Yices 2.4:

- `abs`: absolute value
- `floor`, `ceil`: integer floor and ceiling
- `div`, `mod`: integer division and modulo
- `divides`, `is-int`: check for divisibility and integrality

These operations have the usual meaning. As in the SMT-LIB Ints theory, the division and modulo operations are defined by the following constraints:

$$(\texttt{div}\ x\ k) \in \mathbb{Z}$$

$$x = k.(\texttt{div}\ x\ k) + (\texttt{mod}\ x\ k)$$

$$0 \leq (\texttt{mod}\ x\ k) < |k|.$$

For these operations, Yices 2 extends the SMT-LIB definitions by allowing both x and k to be arbitrary reals, not just integers.

3.3.2 Bitvectors

Yices supports all the bitvector operators defined in the SMT-LIB standards [RT06, BST12, BFT15]. The most commonly used operators are listed in Table 3.1. They include bitvector arithmetic (where bitvectors are interpreted either as unsigned integers or as signed integers in two's complement representation), logical operators such as bitwise OR or AND, logical and arithmetic shifts, concatenation, and extraction of subvectors. Other operators are defined in the theory QF_BV of SMT-LIB (cf. `http://www.smtlib.org`); Yices 2 supports all of them.

The semantics of all the bitvector operators is defined in the SMT-LIB standard. Like other SMT solvers, Yices 2 follows the BTOR conventions for bitvector division by zero [BBL08]. Until recently, this was not the semantics defined by the SMT-LIB standard. The SMT-LIB semantics changed in October 2015. It is now the same as BTOR:

Unsigned Division: If b is the zero bitvector of n bits then

$$\begin{aligned}(\texttt{bvudiv}\ a\ b) &= \texttt{0b111...1}\\ (\texttt{bvurem}\ a\ b) &= a\end{aligned}$$

Operator and Type	Meaning
$\texttt{bvadd} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	addition
$\texttt{bvsub} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	subtraction
$\texttt{bvmul} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	multiplication
$\texttt{bvneg} :: ((\texttt{bv}\ n) \to (\texttt{bv}\ n))$	2's complement opposite
$\texttt{bvudiv} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	quotient in unsigned division
$\texttt{bvudiv} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	remainder in unsigned division
$\texttt{bvsdiv} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	quotient in signed division with rounding toward zero
$\texttt{bvsrem} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	remainder in signed division with rounding toward zero
$\texttt{bvsmod} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	remainder in signed division with rounding toward $-\infty$
$\texttt{bvule} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	unsigned less than or equal
$\texttt{bvuge} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	unsigned greater than or equal
$\texttt{bvult} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	unsigned less than
$\texttt{bvugt} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	unsigned greater than
$\texttt{bvsle} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	signed less than or equal
$\texttt{bvsge} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	signed greater than or equal
$\texttt{bvslt} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	signed less than
$\texttt{bvsgt} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	signed greater than
$\texttt{bvand} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	bitwise and
$\texttt{bvor} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	bitwise or
$\texttt{bvnot} :: ((\texttt{bv}\ n) \to (\texttt{bv}\ n))$	bitwise negation
$\texttt{bvxor} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	bitwise exclusive or
$\texttt{bvshl} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	shift left
$\texttt{bvlshr} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	logical shift right
$\texttt{bvashr} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	arithmetic shift right
$\texttt{bvconcat} :: ((\texttt{bv}\ n) \times (\texttt{bv}\ m) \to (\texttt{bv}\ n+m))$	concatenation
$\texttt{bvextract}_{i,j}((\texttt{bv}\ n) \to (\texttt{bv}\ m))$	extract bits i down to j from a bitvector of size n

Table 3.1: Bitvector Operators

In general, the quotient $(\texttt{bvudiv}\ a\ b)$ is the largest unsigned integer that can be represented on n bits, and is smaller than a/b, and the following identity holds for all bitvectors a and b

$$a \quad = \quad (\texttt{bvadd}\ (\texttt{bvmul}\ (\texttt{bvudiv}\ a\ b)\ b)\ (\texttt{bvurem}\ a\ b)).$$

Signed Division If b is the zero bitvector of n bits then

$$\begin{aligned}
(\texttt{bvsdiv}\ a\ b) &= \texttt{0b000..01} \text{ if } a \text{ is negative} \\
(\texttt{bvsdiv}\ a\ b) &= \texttt{0b111...1} \text{ if } a \text{ is non-negative} \\
(\texttt{bvsrem}\ a\ b) &= a \\
(\texttt{bvsmod}\ a\ b) &= a
\end{aligned}$$

Beside the SMT-LIB operations, Yices includes two operators to convert between arrays of Booleans and bitvectors. These operators were introduced in Yices 2.2.2.

- $(\texttt{bool-to-bv}\ b_1 \ldots b_n)$ is the bitvector obtained by concatenating n Boolean terms $b_1, \ldots, b_n$. The high-order bit is b_1 and the low-order bit is b_n. For example, the expression

  $$(\texttt{bool-to-bv true false false false})$$

  is the same as the bitvector constant `0b1000`.

- $(\texttt{bit}\ a\ i)$ extracts the i-th bit of bitvector a as a Boolean term. If a has n bits, then i must be an index between 0 and $n-1$. The low-order bit has index 0, and the high-order bit has index $n-1$. For example, we have

  $$(\texttt{bit}\ (\texttt{bool-to-bv false b true true})\ 2) \quad = \quad \texttt{b},$$

  where `b` is a Boolean term.

Chapter 4

Yices 2 Architecture

Yices 2 has a modular architecture. You can select a specific combination of theory solvers for your needs using the API or the `yices` executable. With the API, you can maintain several independent contexts in parallel, possibly each using different solvers and settings.

4.1 Main Components

The Yices 2 software can be conceptually decomposed into three main modules:

Term Database Yices 2 maintains a global database in which all terms and types are stored. Yices 2 provides an API for constructing terms, formulas, and types in this database.

Context Management A context is a central data structure that stores asserted formulas. Each context contains a set of assertions to be checked for satisfiability. The context-management API supports operations for creating and initializing contexts, for asserting formulas into a context, and for checking the satisfiability of the asserted formulas. Optionally, a context can support operations for retracting assertions using a push/pop mechanism. Several contexts can be constructed and manipulated independently.

Contexts are highly customizable. Each context can be configured to support a specific theory, and to use a specific solver or combination of solvers.

Model Management If the set of formulas asserted in a context is satisfiable, then one can construct a model of the formulas. The model maps symbols of the formulas to concrete values (e.g., integer or rational values, or bitvector constants). The API provides functions to build and query models.

Figure 4.1 shows the top-level architecture of Yices 2, divided into the three main modules. Each context consists of two separate components: The *solver* employs a Boolean

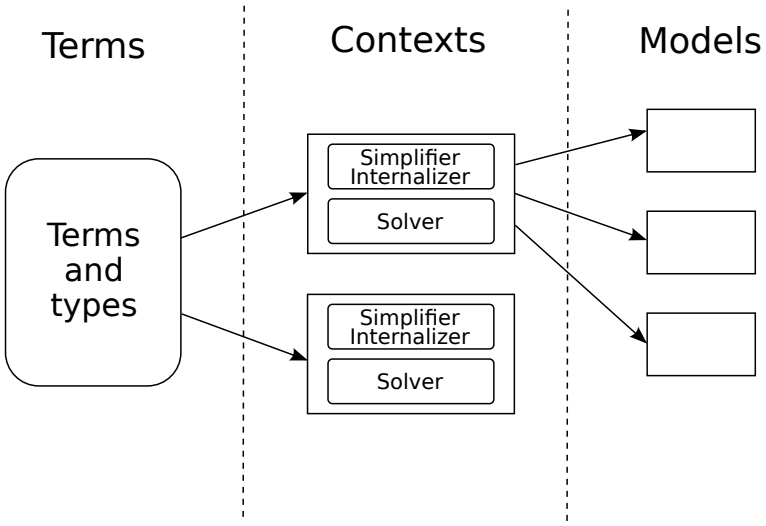

Figure 4.1: Top-level Yices 2 Architecture

satisfiability solver and decision procedures for determining whether the formulas asserted in the context are satisfiable. The *simplifier/internalizer* component converts the format used by the term database into the internal format used by the solver. In particular, the internalizer rewrites all formulas in conjunctive normal form, which is used by the internal SAT solver.

4.2 Solvers

In Yices 2, it is possible to select a different solver (or combination of solvers) for the problem of interest. Each context can thus be configured for a specific class of formulas. For example, you can use a solver specialized for linear arithmetic, or a solver that supports the full Yices 2 language. Figure 4.2 shows the architecture of the most general solver available in Yices 2. A major component of all solvers is a SAT solver based on the Conflict-Driven Clause Learning (CDCL) procedure. The SAT solver is coupled with one or more so-called *theory solvers*. Each theory solver implements a decision procedure for a particular theory. Currently, Yices 2 includes four main theory solvers:

- The *UF Solver* deals with the theory of uninterpreted functions with equality[1]. It implements a decision procedure based on computing congruence closures, similar to the Simplify system [DNS05], with other ideas borrowed from [NO07].

[1] UF stands for uninterpreted functions.

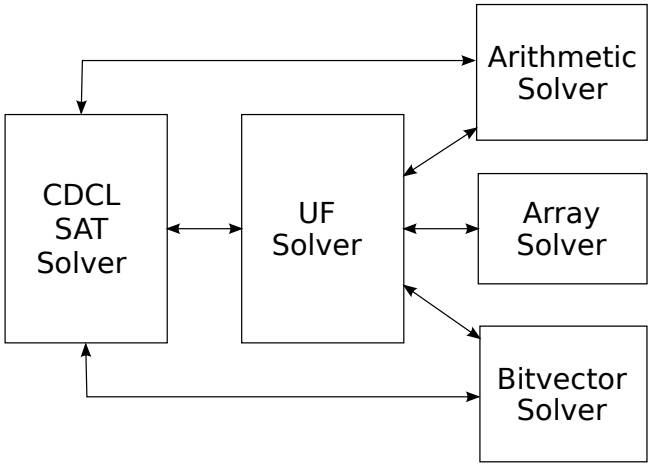

Figure 4.2: Solver Components

- The *Arithmetic Solver* deals with linear integer and real arithmetic. It implements a decision procedure based on the Simplex algorithm [DdM06a, DdM06b].
- The *Bitvector Solver* deals with the theory of bitvectors.
- The *Array Solver* implements a decision procedure for McCarthy's theory of arrays.

Two arithmetic solvers can be used in place of the Simplex-based solver for integer or real difference logic. These solvers implement a decision procedure based on the Floyd-Warshall algorithm. These solvers are more specialized and limited than the Simplex-based solver. They must be used standalone; they cannot be combined with the UF solver.

It is possible to remove some of the components of Figure 4.2 to build simpler and more efficient solvers that are specialized for classes of formulas. For example, a solver for pure arithmetic can be built by directly attaching the arithmetic solver to the CDCL SAT solver. Similarly, Yices 2 can be specialized for pure bitvector problems, or for problems combining uninterpreted functions, arrays, and bitvectors (by removing the arithmetic solver).

Yices 2 combines several theory solvers using the Nelson-Oppen method [NO79]. The UF solver is essential for this purpose; it coordinates the different theory solvers and ensures global consistency. The other solvers (for arithmetic, arrays, and bitvectors) communicate only with the central UF solver and never directly with each other. This property considerably simplifies the design and implementation of theory solvers. More details on the theory-combination method implemented by Yices are given in a tool paper [Dut14].

4.3 Context Configurations

A context can be configured to use different solvers and to support different usage scenarios. The basic operations on a context include:

- asserting one or more formulas
- checking satisfiability of the set of assertions
- building a model if the assertions are satisfiable

Optionally, a context can support addition and removal of assertions using a push/pop mechanism. In this case, the context maintains a stack of assertions organized in successive levels. The push operation starts a new level, and the pop operation removes all assertions at the top level. Thus, push can be thought as setting a backtracking point and pop restores the context state to a previous backtracking point.

Support for push and pop induces some overhead and may disable some preprocessing and simplification of assertions. In some cases, it is then desirable to use a context without support for push and pop, in order to get higher performance. Yices 2 allows users to control the set of features supported by a context by selecting a specific *operating mode*.

- The simplest mode is *one-shot*. In this mode, one can assert formulas then make a one call to the check operation. Assertions are not allowed after the call to check. This mode is the most efficient as Yices may apply powerful preprocessing and simplification (such as symmetry breaking [DFMWP11]).
- The next mode is *multi-checks*. In this mode, several calls to the check operation are allowed. One can assert formulas, call check, assert more formulas and call check again. This can be done as long as the context is satisfiable. Once check returns `unsat`, then no assertions can be added. This mode avoids the overhead of maintaining a stack of assertions.
- The default mode is *push-pop*. In this mode, a context supports the push and pop operations. Assertions are organized in a stack as explained previously.
- The last mode is *interactive*. This mode provides the same functionalities as *push-pop* but the context is configured to recover gracefully when a check operation times out or is interrupted.

4.4 MCSAT

Since version 2.4.0, Yices includes another solver that uses a different approach and architecture. This new solver is based on the Model Constructing Satisfiability Calculus

(MCSAT), and it is currently dedicated to quantifier-free nonlinear real arithmetic. The theory and implementation of MCSAT is discussed in several publications [JBdM13, dMJ13]. Currently, this solver can process input written in the SMT-LIB 2.0 or Yices notations. The MCSAT solver is required for nonlinear arithmetic, but it also supports other theories such as uninterpreted functions or bitvectors.

The MCSAT solver implements functions that can be used for constructing interpolants and it also supports a new form of SMT solving that we call *solving modulo a model*. An explanation of these concepts and application to interpolant construction for non-linear arithmetic are presented in a preprint [JD21].

4.5 Third-Party SAT Solvers

In Yices 2.6.2, we have added support for using third-party Boolean satisfiability solvers. Such solvers are optional but can provide significant performance improvements on bit-vector problems. Use of these SAT solvers is enabled by a command-line option and is currently restricted to non-incremental QF_BV problems. If an external solver is selected, Yices will perform "bit blasting," that is, convert the problem to an equisatisfiable SAT problem in conjunctive normal form (CNF) and use the third-party solver to check satisfiability of this CNF formula.

Chapter 5

Yices Tool

The Yices 2 distribution includes a tool for processing input written in the Yices 2 language. This tool is called `yices` (or `yices.exe` in the Windows and Cygwin distributions). The syntax and the set of commands supported by `yices` are explained in the file `doc/YICES-LANGUAGE` included in the distribution. Several example specifications are also included in the `examples` directory.

```
(define-type BV (bitvector 32))

(define a::BV)
(define b::BV)
(define c::BV (mk-bv 32 1008832))
(define d::BV)

(assert (= a (bv-or (bv-and (mk-bv 32 255)
                            (bv-not (bv-or b (bv-not c))))
                    (bv-and c (bv-xor d (mk-bv 32 1023))))))

(check)

(show-model)
(eval a)
(eval b)
(eval c)
(eval d)
```

Figure 5.1: Example Yices Script

5.1 Example

To illustrate the tool usage, consider file `examples/bv_test2.ys` shown in Figure 5.1. The first line defines a type called `BV`. In this case, `BV` is a synonym for bitvectors of size 32. Then four terms are declared of type `BV`. The three constants `a`, `b`, and `d` are uninterpreted, while `c` is defined as the bitvector representation of the integer 1008832. The next line of the file is an assertion expressing a constraint between `a`, `b`, `c`, and `d`. The command `(check)` checks whether the assertion is satisfiable. Since it is, command `(show-model)` asks for a satisfying model to be displayed. The next commands ask for the value of four terms in the model.

To run `yices` on this input file, just type

```
yices examples/bv_test2.ys
```

The tool will output something like this:

```
sat
(= d 0b00000000000000000000000000000000)
(= b 0b00000000000000000000000000000000)
(= a 0b00000000000000000000000011000000)

0b00000000000000000000000011000000
0b00000000000000000000000000000000
0b00000000000011110110010011000000
0b00000000000000000000000000000000
```

The result of the `(check)` command is shown on the first line (i.e., `sat` for satisfiable). The next three lines show the model as an assignment to the three uninterpreted terms `a`, `b`, and `d`. Then, the tool displays one bitvector constant for each of the `(eval ...)` command.

Since this example contains only terms and constructs from the bitvector theory, we could specify logic `QF_BV` on the command line as follows:

```
yices --logic=QF_BV examples/bv_test2.ys
```

Since the file does not use `push` and `pop`, and it contains only one call to `(check)`, we can select the mode `one-shot`:

```
yices --logic=QF_BV --mode=one-shot examples/bv_test2.ys
```

'To get a more detailed output, we can give a non-zero verbosity level:

```
yices --verbosity=4 examples/bv_test2.ys
```

```
(define x::real)

(assert
  (forall (y::real)
     (=> (and (< (* -1 y) 0) (< (+ -10 y) 0))
         (< (+ -7 (* -2 x) y) 0))))

(ef-solve)
(show-model)
```

Figure 5.2: Example Exists/Forall Problem

5.2 Exists/Forall Problems

Yices can solve a restricted class of quantified problems, known as *exists/forall problems*. As the name indicates, such problems are of the following general form:

$$\exists x_1, \ldots, x_n : \forall y_1, \ldots, y_m : P(x_1, \ldots, x_n, y_1, \ldots, y_m).$$

In many applications, the goal to find values $a_1, \ldots, a_n$ for the existentially quantified variables $x_1, \ldots, x_n$ such that the following formula

$$\forall y_1, \ldots, y_m : P(a_1, \ldots, a_n, y_1 \ldots, y_m)$$

is valid.

Yices can solve such problems when the quantified variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ either have finite type or are real variables. The algorithm implemented in Yices and an example application are described in [GSD+14].

Figure 5.2 shows how exists/forall problems are specified in the Yices language. Global declarations, such as the uninterpreted constant `x` in the figure, correspond to the existential variables. Constraints are then stated as assertions be of the form `(forall (y ...) P)` where `y ...` are universal variables. It is allowed to have several assertions of this form, as well as quantifier-free constraints on the global variables.

The command `(ef-solve)` invokes the exists/forall solver. This commands is similar to `(check)`. It reports `sat` if the problem is satisfiable, `unsat` if it is not, or `unknown` if the solver does not terminate within a fixed number of iterations. If `(ef-solve)` returns `sat`, then we can display the solution it has found using `(show-model)`. This is illustrated in Figure 5.2.

To run `yices` on this example, we must give option `--mode=ef` on the command line:

```
yices --mode=ef test.ys
```

This will produce the following output:

```
sat
(= x 2)
```

The first line is the result of `(ef-solve)`. The second line is the model, which just shows the value of the global variable `x`.

As previously, we can get more detailed output by increasing the verbosity:

```
yices --mode=ef --verbosity=5 test.ys
```

It is also possible to specify a logic on the command-line.

5.3 Unsat Cores

Since version 2.6.1, the Yices tool can produce unsat cores. This feature comes in two flavors, as in SMT-LIB 2.6.

5.3.1 Labeled Assertions

The first method is illustrated in Figure 5.3. One gives labels to assertions and command `(show-unsat-core)` displays an unsat core when a call to `(check)` returns `unsat`. The labels can be any symbols; there is a separate name space for these labels.

```
(define x::real)

(assert (>= x 0))   ;; regular assertion
(assert (> x 3) A)  ;; labeled assertion: A is the label
(assert (< x 3) B)  ;; labeled assertion
(assert (= x 3) C)  ;; another labeled assertion
(check)             ;; will return unsat
(show-unsat-core)   ;; display unsat core
```

Figure 5.3: Unsat cores using labeled assertions

5.3.2 Check With Assumptions

The second method uses command `(check-assuming ...)`, a variant of the `(check)` command that takes a list of assumptions as arguments. This is illustrated in Figure 5.4. An assumption is a restricted form of terms. It can be either `<name>` or `(not <name>)`, where `<name>` is the name of a Boolean term. If `(check-assuming)` returns `unsat`, then one can get an unsat core using `(show-unsat-assumptions)`. This command shows a subset of the assumptions that are inconsistent with the context.

```
(define x::real)
(define A::bool (> x 3))
(define B::bool (> x 2))
(define C::bool (> x 4))

(assert (and (>= x 0) <= x 5)) ;; regular assertion
(check-assuming A (not B) C)   ;; will return unsat
(show-unsat-assumptions)       ;; unsat core: (A (not B))
```

Figure 5.4: Check with assumptions

5.4 Tool Invocation

Yices is invoked on an input file by typing

```
yices [option] <filename>
```

If no `<filename>` is given, `yices` will run in interactive mode and will read the standard input. The following options are supported.

`--logic=<name>` Select an SMT-LIB logic.

The `<name>` must either be an SMT-LIB logic name such as `QF_UFLIA` or the special name `NONE`.

Yices recognizes the logics defined at `http://www.smtlib.org` (as of July 2014). Option `--logic=NONE` configures `yices` for propositional logic.

By default—that is, if no logic is given—`yices` includes all the theory solvers described in Section 4.2. In this default configuration, `yices` supports linear arithmetic, bitvectors, uninterpreted functions, and arrays. If a logic is specified, `yices` uses a specialized solver or combination of solvers that is appropriate for the given logic. Some of the search parameters will also be set to values that seem to work well for this logic (based on extensive benchmarking). All the search parameters can also be modified individually using the command `(set-param ...)`.

If option `--logic=NONE` is given, then `yices` includes no theory solvers at all. All assertions must be purely propositional (i.e., involve only Boolean terms).

If the selected logic includes nonlinear arithmetic (e.g., `--logic=QF_UFNRA`), then `yices` will automatically select the MCSAT solver. To force use of the MCSAT solver on logics that do not require it, use command-line option `--mcsat`.

`--arith-solver=<solver>` Select one of the possible arithmetic solvers.

`<solver>` must be one of `simplex`, `floyd-warshall`, or `auto`.

If the logic is QF_IDL (integer difference logic) or QF_RDL (real difference logic), then this option can be used to select the arithmetic solver: either the generic Simplex-based solver or a specialized solver based on the Floyd-Warshall algorithm. If option --arith-solver=auto is given, then the arithmetic solver is determined automatically; the default is auto.

This option has no effect for logics other than QF_IDL or QF_RDL.

--mode=<mode> Select solver features.

<mode> can be one-shot, multi-checks, push-pop, interactive, or ef.

The mode ef enables the exists/forall solver. In this mode, Yices can solve problems with universally quantified variables. The command (ef-solve) can be used for a single block of assertions. No assertions are allowed after the call to (ef-solve).

The other four modes select the set of functionalities supported by the solver as follows:

- one-shot: no assertions are allowed after the (check) command. In this mode, yices can check satisfiability of a single block of assertions and possibly build a model if the assertions are satisfiable.
- multi-checks: several calls to (assert) and (check) are allowed.
- push-pop: like multi-checks but with support for adding and retracting assertions via the commands (push) and (pop).
- interactive: supports the same features as the push-pop mode, but with a different behavior when (check) is interrupted.

In the first two modes, yices employs more aggressive simplifications when processing assertions; this can lead to better performance on some problems.

Unsat cores and checks with assumptions are not supported in mode one-shot.

In interactive mode, the solver context is saved before every call to (check) and it is restored if (check) is interrupted. This introduces some overhead, but the solver recovers gracefully if (check) is interrupted or times out. In the non-interactive modes, the solver exits after the first interruption or timeout.

The default mode is push-pop if a file name is given on the command line. If not input file is given, then the default mode is interactive and the solver reads standard input.

Mode one-shot is required to use the Floyd-Warshall solvers.

--mcsat Force use of the MCSAT solver.

This option forces yices to use the MCSAT solver instead of the default CDCL(T) solver. By default, MCSAT is used only if the logic includes non-linear arithmetic. Using option --mcsat selects the MCSAT solver on other logics. For example, this can be used to use the MCSAT solver on bitvector problems.

`--print-success` Print `ok` after every command that would otherwise execute silently.

Many commands are executed silently by `yices` (i.e., they produce no output). This can be a problem for tools that interact with `yices` via pipes or files. Option `--print-success` modifies this behavior. With this option, `yices` will print `ok` on its standard output when a command successfully executes.

`--version, -V` Display version information then exit.

This displays the Yices version number, the version of the GMP library linked with Yices, and information about build date and platform. For example, here is the output for Yices 2.2.0 built on MacOS X

```
Yices 2.2.0
Copyright SRI International.
Linked with GMP 5.1.3
Copyright Free Software Foundation, Inc.
Build date: 2013-12-21
Platform: x86_64-apple-darwin13.0.2 (release)
```

If you ever have to report a bug, please include this version information in your bug report.

`--help, -h` Print a summary of options

`--verbosity=<level>, -v <level>` Run in verbose mode.

As indicated in this list, some options can be given either in a long form (like `--verbosity=4`) or in an equivalent short from (like `-v 4`). In all cases the long and short forms are equivalent.

5.5 Input Language

The syntax of the Yices input language is summarized in Figures 5.5 to 5.8.

5.5.1 Lexical Elements

Comments

Input files may contain comments, which start with a semi-colon `';'` and extend to the end of the line.

```
<command>  ::=
             ( define-type <symbol> )
           | ( define-type <symbol> <typedef> )
           | ( define <symbol> :: <type> )
           | ( define <symbol> :: <type> <expression> )
           | ( assert <expression> )
           | ( assert <expression> <symbol> )
           | ( exit )
           | ( check )
           | ( check-assuming <assumption-list> )
           | ( push )
           | ( pop )
           | ( reset )
           | ( show-model )
           | ( eval <expression> )
           | ( echo <string> )
           | ( include <string> )
           | ( set-param <symbol> <immediate-value> )
           | ( show-param <symbol> )
           | ( show-params )
           | ( show-stats )
           | ( reset-stats )
           | ( set-timeout <number> )
           | ( show-timeout )
           | ( dump-context )
           | ( help )
           | ( help <symbol> )
           | ( help <string> )
           | ( ef-solve )
           | ( export-to-dimacs <string> )
           | ( show-implicant )
           | ( show-unsat-core )
           | ( show-unsat-assumptions )
           | ( show-reduced-model )
           | EOS
```

Figure 5.5: Yices Syntax: Commands

```
<immediate-value> ::=
           true
         | false
         | <number>
         | <symbol>

<number> ::= <rational> | <float>

<assumption-list> ::=
         |
         | <assumption> <assumption-list>

<assumption> ::=
         | <symbol>
         | ( not <symbol> )
```

Figure 5.6: Yices Syntax: Command Arguments

```
<typedef> ::=
           <type>
         | ( scalar <symbol> ... <symbol> )

<type> ::=
           <symbol>
         | ( tuple <type> ... <type> )
         | ( -> <type> ... <type> <type> )
         | ( bitvector <rational> )
         | int
         | bool
         | real
```

Figure 5.7: Yices Syntax: Types

```
<expr> ::=
           true
         | false
         | <symbol>
         | <rational>
         | <float>
         | <binary bv>
         | <hexa bv>
         | ( forall ( <var_decl> ... <var_decl> ) <expr> )
         | ( exists ( <var_decl> ... <var_decl> ) <expr> )
         | ( lambda ( <var_decl> ... <var_decl> ) <expr> )
         | ( let ( <binding> ... <binding> ) <expr> )
         | ( update <expr> ( <expr> ... <expr> ) <expr> )
         | ( <function> <expr> ... <expr> )

<function> ::=
           <function-keyword>
         | <expr>

<var_decl> ::= <symbol> :: <type>

<binding> ::= ( <symbol> <expr> )
```

Figure 5.8: Yices Syntax: Expressions

Strings

Strings are similar to strings in C. They are delimited by double quotes `"` and may contain escaped characters:

- The characters `\n` and `\t` are replaced by newline and tab, respectively.
- The character `\` followed by at most three octal digits (i.e., from `0` to `7`) is replaced by the character whose ASCII code is the octal number.
- In all other cases, `\<char>` is replaced by `<char>` (including if `<char>` is a newline or `\`).
- A newline cannot occur inside the string, unless preceded by `\`.

Numerical Constants

Numerical constants can be written as decimal integers (e.g., `44` or `-3`), rational (e.g., `-1/3`), or using a floating-point notation (e.g., `0.07` or `-1.2e+2`). Positive constants can start with an optional `+` sign. For example `+4` and `4` denote the same number.

Bitvector Constants

Bitvector constants can be written in a binary format using the prefix `0b` or in hexadecimal using the prefix `0x`. For example, the expressions `0b01010101` and `0x55` denote the same bitvector constant of eight bits.

Symbols

A symbol is any character string that's not a keyword (see Table 5.1) and doesn't start with a digit, a space, or one of the characters `(`, `)`, `;`, `:`, and `"`. If the first character is `+` or `-`, then it must not be followed by a digit. Symbols end by a space, or by any of the characters `(`, `)`, `;`, `:`, or `"`. Here are some examples:

```
a_symbol __another_one  X123  &&&  +z203  t\12
```

All the predefined keywords and symbols are listed in Table 5.1.

5.5.2 Declarations

A declaration either introduces a new type or term or gives a name to an existing type or term. Yices uses different name spaces for types and terms. It is then permitted to use the same name for a type and for a term.

*	+	-
->	/	/=
<	<=	<=>
=	=>	>
>=	^	abs
and	assert	bit
bitvector	bool	bool-to-bv
bv-add	bv-and	bv-ashift-right
bv-ashr	bv-comp	bv-concat
bv-div	bv-extract	bv-ge
bv-gt	bv-le	bv-lshr
bv-lt	bv-mul	bv-nand
bv-neg	bv-nor	bv-not
bv-or	bv-pow	bv-redand
bv-redor	bv-rem	bv-repeat
bv-rotate-left	bv-rotate-right	bv-sdiv
bv-sge	bv-sgt	bv-shift-left0
bv-shift-left1	bv-shift-right0	bv-shift-right1
bv-shl	bv-sign-extend	bv-sle
bv-slt	bv-smod	bv-srem
bv-sub	bv-xnor	bv-xor
bv-zero-extend	ceil	check
define	define-type	distinct
div	divides	dump-context
echo	ef-solve	eval
exists	exit	export-to-dimacs
false	floor	forall
help	if	include
int	is-int	ite
lambda	let	mk-bv
mk-tuple	mod	not
or	pop	push
real	reset	reset-stats
scalar	select	set-param
set-timeout	show-implicant	show-model
show-param	show-params	show-reduced-model
show-stats	show-unsat-core	show-unsat-assumptions
true	tuple	tuple-update
update	xor	

Table 5.1: Keywords and predefined symbols

Type Declaration

A type declaration is a command of the following two forms.

```
(define-type name)
(define-type name type)
```

The fist form creates a new uninterpreted type called `name`. The second form gives a `name` to an existing `type`. After this definition, every occurrence of `name` refers to `type`. A variant of this second form is used to define scalar types. In these two commands, `name` must be a symbol that's not already used as a type name.

Term Declaration

A term is declared using one for the following two commands.

```
(define name :: type)
(define name :: type term)
```

The first form declares a new uninterpreted term of the given `type`. The second form assigns a `name` to the given `term`, which must be of type `type`. The `name` must be a symbol that's not already used as a term name.

5.5.3 Types

Yices includes a few predefined types for arithmetic and bitvectors. One can extend the set of atomic types by creating uninterpreted and scalar types. In addition to the atomic types, Yices provides constructors for tuple and function types. More details about types and subtyping are given in Section 3.1.

Predefined Types

The predefined types are `bool`, `int`, `real`, and `(bitvector k)` where k is a positive integer. For example a bit-vector variable `b` of 32 bits is declared using the command

```
(define b::(bitvector 32))
```

The number of bits must be positive so `(bitvector 0)` is not a valid type. There is also a hard-coded limit on the size of bitvectors (namely, $2^{28} - 1$). Of course, this is a theoretical limit; the solver will most likely run out of memory if you attempt to use bitvectors that are that large.

Uninterpreted Types

A new uninterpreted type T can be introduced using the command

```
(define-type T)
```

This command will succeed provided `T` is a fresh type name, that is, if there is no existing type called `T`. As explained in Section 3.1, an uninterpreted type denotes a nonempty collection of objects. There is no cardinality constraint on `T`, except that `T` is not empty.

Scalar Type

A scalar type is defined by enumerating its elements. For example, the following declaration

```
(define-type P (scalar A B C))
```

defines a new scalar type called `P` that contains the three distinct constants `A`, `B`, and `C`. Such a declaration is valid provided `P` is a fresh type name and `A`, `B`, and `C` are all fresh term names.

The enumeration must include at least one element, but singleton types are allowed. For example, the following declaration is valid.

```
(define-type Unit (scalar One))
```

It introduces a new type `Unit` of cardinality one, and which contains `One` as its unique element. Thus, any term of type `Unit` is known to be equal to `One`.

Tuple Types

A tuple type is written `(tuple tau_1 ... tau_n)` where `tau_i` is a type. For example, the type of pairs of integer can be declared as follows:

```
(define-type Pairs (tuple int int))
```

Then one can declare an uninterpreted constant `x` of this type as follows

```
(define x::Pairs)
```

This is equivalent to the declaration

```
(define x::(tuple int int))
```

Tuple types with a single component are allowed. For example, the following declaration is legal.

```
(define-type T (tuple bool))
```

Function Types

A function type is written `(-> tau_1 ... tau_n sigma)`, where n is positive, and the `tau_i`s and `sigma` are types. The types `tau_1`, ..., `tau_n` define the domain of the function type, and `sigma` is the range. For example, a function defined over the integers and that returns a Boolean can be declared as follows:

```
(define f::(-> int bool))
```

Yices does not have a distinct type construct for arrays. In Yices, arrays are the same as functions.

5.5.4 Terms

Yices uses a Lisp-like syntax. For example, the polynomial $x + 3y + z$ is written

```
(+ x (* 3 y) z)
```

In general, all associative operations can take one, two, or more arguments. For example, one can write

```
(or A)    (or A B)    (or A B C D)
```

since `or` is associative.

If-Then-Else

Yices provides an if-then-else construct that applies to any type. An if-then-else term can be written using either one of the two following forms

```
(ite c t1 t2)     (if  c t1 t2)
```

Both forms are equivalent and just mean "if `c` then `t1` else `t2`." The condition `c` must be a Boolean term, and the two terms `t1` and `t2` must have compatible types. If `t1` and `t2` have the same type τ then `(ite c t1 t2)` also has type τ. Otherwise, as explained in Section 3.1, the type of `(if c t1 t2)` is the minimal supertype of `t1` and `t2`. For example, if `t1` has type `int` and `t2` has type `real`, then `(ite c t1 t2)` has type `real`.

Equalities and Disequalities

Equalities and disequalities are written as follows

```
(=  t1 t2)        (/= t1 t2)
```

where `t1` and `t2` are two terms of compatible types. These operators are binary. Unlike SMT-LIB 2, Yices does not support constraints such as `(= x y z t u)`. On the other hand, Yices includes an n-ary `distinct` operator that generalizes disequality.

The Boolean term

```
(distinct t_1 .... t_n )
```

is true if `t_1`, ..., `t_n` are all different from each other. The terms `t_1` to `t_n` must all have compatible types. There must be at least two arguments. The expression `(distinct a b)` means the same thing as `(/= a b)`.

Boolean Operators

`true`	`false`
`and`	`or`
`not`	`xor`
`<=>`	`=>`

Table 5.2: Boolean Constants and Operators

The usual Boolean constants and functions are available. They are listed in Table 5.2. The associative and commutative operators `or`, `and`, and `xor` can take any number of arguments. The equivalence (`<=>`) and implication (`=>`) operators take exactly two arguments.

One can also use the equality and disequality operators with Boolean terms. If `t1` and `t2` are Boolean then `(= t1 t2)` is the same as `(<=> t1 t2)`, and `(/= t1 t2)` is the same as `(xor t1 t2)`.

Basic Arithmetic

Syntax	Meaning	
`(+ a1 ... a_n)`	sum	$a_1 + \ldots + a_n$
`(* a1 ... a_n)`	product	$a_1 \times \ldots \times a_n$
`(- a)`	opposite	$-a$
`(- a1 a2 ... a_n)`	difference	$a_1 - a_2 - \ldots - a_n$
`(^ a k)`	exponentiation	a^k
`(/ a c)`	division	a/c
`(<= a1 a2)`	inequality	$a_1 \leq a_2$
`(>= a1 a2)`	inequality	$a_1 \geq a_2$
`(< a1 a2)`	strict inequality	$a_1 < a_2$
`(> a1 a2)`	strict inequality	$a_1 > a_2$

Table 5.3: Arithmetic Operations

Arithmetic constants can be written in decimal, as rationals, or using the floating point notation. Internally, Yices uses exact rational arithmetic and it represents all arithmetic constants as rationals.

The usual arithmetic operations and comparison operators are summarized in Table 5.3. One can freely mix terms of real and integer types in all operations. The exponent k in `(^ a k)` must be a non-negative integer constant. The divisor `c` in `(/ a c)` must be a non-zero constant.

The Yices language includes more than linear arithmetic, but this is for future extensions. Currently, Yices does not include solvers for non-linear arithmetic (cf. Section 4.2).

Arithmetic Functions

Other arithmetic operations defined in Yices are listed in Table 5.4. The operations `abs`, `floor`, and `ceil` have the usual meaning:

- `(abs x)` is the absolute value of `x`.
- `(floor x)` is the largest integer less than or equal to `x`.
- `(ceil x)` is the smallest integer larger than or equal to `x`.

For integer division and modulo, Yices uses the SMT-LIB conventions (see Section 3.3.1), except that the divider `k` must be a non-zero constant and that both `div` and `mod` are defined over the reals, not just the integers. Division by a non-constant term is not supported.

In the divisibility test `(divides k x)`, the divider `k` must be a rational constant but it can be zero. The term `x` can be any real. The atom `(divides k x)` is true if there exists an integer $n \in \mathbb{Z}$ such that $\mathtt{x} = n\mathtt{k}$.

Syntax	Meaning
`(abs x)`	absolute value
`(floor x)`	floor
`(ceil x)`	ceiling
`(div x k)`	integer division
`(mod x k)`	modulo
`(divides k x)`	divisibility test
`(is-int x)`	integrality test

Table 5.4: Arithmetic Functions

Bitvectors Constants

A bitvector constant can be written in binary or hexadecimal notation, as follows

```
0b0    0b1    0xFFFF     0xaaaa     0xC0C0D0D0
```

In the binary notation, the number of bits in the constant is equal to be number of binary digits. For example, the three terms

Syntax	Meaning
`(bv-add u1 ... u_n)`	sum
`(bv-mul u1 ... u_n)`	product
`(bv-sub u1 ... u_n)`	subtraction
`(bv-neg u)`	2s-complement
`(bv-pow u k)`	exponentiation
`(bv-not u)`	bitwise complement
`(bv-and u1 ... u_n)`	bitwise and
`(bv-or u1 ... u_n)`	bitwise or
`(bv-xor u1 ... u_n)`	bitwise xor
`(bv-nand u1 ... u_n)`	bitwise nand
`(bv-nor u1 ... u_n)`	bitwise nor
`(bv-xnor u1 ... u_n)`	bitwise xnor

Table 5.5: Bitvector Operations (Arithmetic and Bitwise Logic)

```
0b1   0b0001   0b00001
```

denote distinct bitvector constants, of one, four, and five bits, respectively. In the hexadecimal notation, the number of bits is equal to four times the number of hexadecimal digit.

One can also construct a bitvector constant using the expression:

```
(mk-bv size value)
```

In this expression, both `size` and `value` must be integer constants; `size` is the number of bits in the bitvector constant and `value` is the decimal value of the constant interpreted as a non-negative integer. The `size` must then be positive, and the `value` must be non-negative. If `value` is more than $2^{\texttt{size}}$, only the residue of `value` modulo $2^{\texttt{size}}$ is taken into account. For example, the expressions

```
(mk-bv 3 6)     (mk-bv 3 22)
```

construct the same bitvector constant (whose binary representation is `0b110`).

Bitvector Arithmetic

Table 5.5 lists all the arithmetic and bitwise operators. All operators in this table take arguments that have the same size and return a result of that size. As usual, the associative operators can take one, two, or more arguments. The `bv-sub` operator takes at least two arguments. In `(bv-pow u k)`, the power k must be a non-negative integer constant.

The expression `(bv-xnor u1 ... u_n)` is the same as `(bv-not (bv-xor u1 ... u_n))`.

Syntax	Meaning
`(bv-shift-left0 u k)`	left shift, padding with 0
`(bv-shift-left1 u k)`	left shift, padding with 1
`(bv-shift-right0 u k)`	right shift, padding with 0
`(bv-shift-right1 u k)`	right shift, padding with 1
`(bv-ashift-right x k)`	arithmetic shift by k bits
`(bv-rotate-left x k)`	rotate by k bits to the left
`(bv-rotate-right x k)`	rotate by k bits to the right
`(bv-shl u v)`	left shift (padding with 0)
`(bv-lshr u v)`	logical right shift (padding with 0)
`(bv-ashr u v)`	arithmetic shift (padding with the sign bit)

Table 5.6: Bitvector Operations (Shift and Rotate)

Syntax	Meaning
`(bv-extract i j u)`	subvector extraction
`(bv-concat u1 ... u_n)`	concatenation
`(bv-repeat u k)`	repeated concatenation
`(bv-sign-extend u k)`	sign extension
`(bv-zero-extend u k)`	zero extension
`(bv-redor u)`	or-reduction
`(bv-redand u)`	and-reduction
`(bv-redcomp u v)`	equality reduction

Table 5.7: Bitvector Operations (Structural Operators)

Bitvector Shift and Rotate

Table 5.6 lists the shift and rotate operations. The operations in the first seven rows shift a bitvector `u` by a fixed number of bits k. If `u` is a bitvector of n bits, then k must be an integer constant such that $0 \leq k \leq n$. The `bv-shl`, `bv-lshr`, and `bv-ashr` operators (last three rows of Table 5.6) take two bitvector arguments `u` and `v`, which must be bitvectors of the same size n. The shift operation is applied to `u` and the value of `v`, interpreted as an unsigned integer in the range $[0, 2^n - 1]$, defines the shift amount. The semantics follows the SMT-LIB standards: if `v`'s value is more than n then the padding bit is copied n times.

Bitvector Structural Operations

The operators in Table 5.7 perform extraction, concatenation, and other structural operations. The expression `(bv-extract i j u)` is the segment of bitvector `u` formed by taking bits $j, j+1, \ldots, i$. If `u` is a bitvector of n bits then the constants i and j must satisfy $0 \leq j \leq i \leq n-1$, and the result is a bitvector of $(i - j + 1)$ bits. For example, we have

```
(bv-extract 7 2 0b110110100) = 0b101101.
```

Syntax	Meaning
`(bv-div u v)`	quotient in unsigned division
`(bv-rem u v)`	remainder in unsigned division
`(bv-sdiv u v)`	quotient in signed division
`(bv-srem u v)`	remainder in signed division
`(bv-smod u v)`	remainder in signed division (rounding to $-\infty$)

Table 5.8: Bitvector Operations (Divisions)

In `(bv-repeat u k)`, bitvector `u` is concatenated with itself k times. The integer constant k must be positive. In the sign and zero extension operators, vector `u` is extended by adding k bits (either zero or `u`'s sign bit copied k times). In these two operations, k must be non-negative.

The `bv-redor`, `bv-redand`, and `bv-redcomp` operators produce a one-bit vector. The term `(bv-redor u)` is the or of `u`'s bits; it is equal to `0b0` if all bits of `u` are zero, and to `0b1` otherwise. Similarly, `(bv-redand u)` is the and of `u`'s bit; it is equal to `0b1` if all bits of `u` are one and to `0b0` otherwise. In `(bv-redcomp u v)`, the arguments `u` and `v` must be two bitvectors of the same size. The operator performs a one-to-one comparison of the bits of `u` and `v` and returns either `0b1`, if `u` and `v` are equal, or `0b0`, if `u` and `v` are distinct.

Bitvector Division

Table 5.8 lists the division and remainder operators. In this table, `u` and `v` must be two bitvectors of the same size n.

In the unsigned division and quotient operations, `u` and `v` are interpreted as integers in the interval $[0, 2^n - 1]$. As explained in section 3.3.2, `(bv-div u v)` is the largest integer that can be represented using n bits and is smaller than or equal to u/v. The unsigned remainder `(bv-rem u v)` satisfies the identity

```
u = (bv-add (bv-mul (bv-div u v) v) (bv-rem u v)).
```

In the signed division and quotient, `u` and `v` are interpreted as integers in the interval $[-2^{n-1}, 2^{n-1} - 1]$ (in 2s-complement representation), and the division is done with rounding to zero.

- If u/v is non-negative, then `(bv-sdiv u v)` is the largest integer q in $[0, 2^{n-1} - 1]$ such that $0 \leq q \leq u/v$.
- If u/v is negative then `(bv-sdiv u v)` is the smallest integer q in $[-2^{n-1}, 0]$ such that $u/v \leq q \leq 0$.

The signed remainder operation satisfies the identity

Syntax	Meaning
`(bv-ge u v)`	$u \geq v$ unsigned
`(bv-gt u v)`	$u > v$ unsigned
`(bv-le u v)`	$u \leq v$ unsigned
`(bv-lt u v)`	$u < v$ unsigned
`(bv-sge u v)`	$u \geq v$ signed
`(bv-sgt u v)`	$u > v$ signed
`(bv-sle u v)`	$u \leq v$ signed
`(bv-slt u v)`	$u < v$ signed

Table 5.9: Bitvector Operations (Comparison)

```
u = (bv-add (bv-mul (bv-sdiv u v) v) (bv-srem u v)).
```

The last operator in Table 5.8 is the remainder in the signed division of `u` by `v` with rounding to $-\infty$. In this operation, `u` and `v` are interpreted as signed integers in the interval $[-2^{n-1}, 2^{n-1} - 1]$; the quotient is $\lfloor u/v \rfloor$ (i.e., the largest integer q such that $q \leq u/v$); and the remainder is $u - qv$. The special case $v = 0$ is explained in Section 3.3.2.

Bitvector Inequalities

Table 5.9 lists the inequality comparison operators for bitvectors. In the table, `u` and `v` must be two bitvector terms of the same size. Depending on the operation, both are interpreted as unsigned integers or as signed integers (using 2s-complement representation). All operators return a Boolean. As usual, one can also apply the equality and disequality operators to two bitvectors of the same size.

Conversions Between Booleans and Bitvectors

Two operations, listed in Table 5.10, convert Booleans to bitvectors and conversely.

Syntax	Meaning
`(bool-to-bv b1 ... bn)`	Booleans to bitvector
`(bit u i)`	Bit extraction

Table 5.10: Bitvector Operators (Conversions)

Operator `bool-to-bv` builds a bitvector from n Boolean terms $b_1, \ldots, b_n$. The result is a bitvector of n bits equal to the concatenation of $b_1, \ldots, b_n$. The high-order bit is b_1 and the low-order bit is b_n. For example,

```
(bool-to-bv true true false false)
```

is equal to the bitvector constant `0b1100`.

Expression `(bit u i)` is the i-th bit of bitvector `u` as a Boolean. If `u` is a bitvector of n bits then the index `i` must be an integer constant between 0 and $n-1$. The lower-order bit has index 0 and the high-order bit has index $n-1$. For example, we have

```
(bit 0b1100 3) = true
(bit 0b1100 2) = true
(bit 0b1100 1) = false
(bit 0b1100 0) = false
```

Tuples

A tuple term can be constructed using `(mk-tuple t1 ... t_n)` where $n \geq 1$ and `t1`, ..., `t_n` are arbitrary terms. For example, a pair of integers can be constructed using

```
(mk-tuple -1 1)
```

The projection operation extracts the i-th component of a tuple. It is denoted by `(select t i)` where `t` is a term of tuple type and i is an integer constant. If the tuple has n components, then i must be between 1 and n. The components are indexed from 1 to n starting from the left. For example, we have

```
(select (mk-tuple -1 1) 1) = -1
(select (mk-tuple -1 1) 2) = 1
```

Yices includes a tuple-update operator. The expression `(tuple-update t i v)` is equal to tuple `t` with its i-th component replaced by `v`. The type of `v` must be a subtype of the i-th component of `t`.

Function Updates

Array or function update is written `(update a (i_1 ... i_n) v)`. In this expression, `a` must be a term with a function type and n is the arity of `a`. The expression constructs a function `b` that is equal to `a`, except that it maps `i_1`,...,`i_n` to `v`. The semantics and typechecking rules of this operator are explained in Section 3.2.

5.5.5 Commands

The Yices commands allow one to declare types and terms, build a set of assertions, check their satisfiability, and query models. Other commands set parameters that control preprocessing and heuristics used by the different solvers.

Declarations

As presented in Section 5.5.2, a type declaration has one of the following forms

```
(define-type name)
(define-type name type)
```

A term declaration is similar:

```
(define name :: type)
(define name :: type term)
```

To define a function, one can use the `lambda` notation. Here is an example:

```
(define max::(-> real real real)
   (lambda (x::real y::real) (if (< x y) y x)))
```

This defines the function `max` that computes the maximum of two real numbers. Note that such a function definition acts like a macro. A term of the form `(max a b)` is eagerly replaced by the "function body", that is, by the term `(if (< a b) b a)`. The ability to define function is useful to abbreviate specifications, but it must be used with care. Since the substitution is performed eagerly, the expanded terms may grow quickly, especially if they contain nested function applications.

All declarations have global scope and are permanent. They are not affected by commands such as push, pop, or reset. Also, as discussed previously, Yices uses separate name spaces for terms and for types.

Assertions

The following command adds an assertion to the current context.

```
(assert formula)
```

In this command, the `formula` must be a Boolean term.

In the mode `one-shot`, assertions are stored internally and are not processed immediately. Processing of assertions is delayed, and all assertions are processed and simplified on the first call to `(check)`.

In all other modes, the assertions are processed and simplified immediately and are added to the context. As a result, `yices` may detect and report that the current set of assertions is inconsistent after an `assert` command. This happens when the context is seen to be unsatisfiable by simplification only. The most trivial example is:

```
(assert false)
```

Once the context is unsatisfiable, any new assertion is treated as an error.

Labeled Assertions

An optional label can be provided for assertions as follows:

```
(assert formula label)
```

In this command, the `formula` must be a Boolean term and the `label` must be a symbol. Yices uses a separate name space for storing these labels, so the same symbol can be used as a type name or a term name.

The label identifies the assertion as relevant to unsat core If a call to `(check)` returns `unsat`, then a corresponding unsat core is a subset of all the labeled assertions that is inconsistent with the context. Command `(show-unsat-core)` displays this unsat core by listing the labels of all the assertions in the core.

The labeled assertions must have distinct labels. Yices will complain if the label provided is already used. The set of labeled assertions is maintained in a stack like regular assertions. For example, labeled assertions may be removed by a call to `(pop)` or `(reset)`.

Check

The command

```
(check)
```

checks whether the current set of assertions is satisfiable.

If the context's current status is already known, then the command returns immediately and prints the status as either `sat` or `unsat`. This happens, for example, in the following situation:

```
(assert ...)
(check)
(check)
```

The context status is known after the first `(check)` command (provided this command does not timeout or otherwise fails). Then the second `(check)` does nothing and just prints the current status.

If the context's status is unknown, then `(check)` invokes the SMT solver to establish whether the assertions are satisfiable. As discussed previously, the actual solver or solver combination is dependent on command-line options given to the `yices` tool. In particular, the `--logic` option allows one to select a solver architecture that is specialized for a particular logic. For best performance, it is usually better to specify the logic if it is known in advance.

Several parameters also control the heuristics employed by the solver. Yices uses default settings based on the specified logic (or global defaults if no logic is given). All these parameters can be examined and modified, using the command `show-params` and `set-param` described in a subsequent section.

One can also provide a timeout before calling `(check)`. If the timeout is reached or the search is interrupted (by CTRL-C), then the result will be displayed as `interrupted`.

Check With Assumptions

The command

```
(check-assuming <assumption> ... <assumption>)
```

checks whether the current set of assertions together with a list of assumptions is satisfiable. If it is not, command `(show-unsat-assumptions)` will provide an unsat core (i.e., a subset of the assumptions that are inconsistent with the context).

Each assumption must either be the name of a Boolean term or the negation of a named Boolean term. For example, after the following definitions:

```
(define A::bool <expression>)
```

then both `A` and `(not A)` can be used as assumptions.

It is allowed for the same name to occur several times in the assumption list. It is also allowed to give an empty list of assumptions.

Mixing assumptions and labeled assertions in a context is not supported. Command `(check-assuming ...)` will fail if there are labeled assertions.

Push, Pop, Reset

Command `(push)`, `(pop)`, and `(reset)` allows one to manipulate the set of assertions.

The command `(reset)` clears all assertions. The current context is then returned to its initial state, where the set of assertions is empty. This command can be used in all modes.

The push and pop commands are supported by `yices` if it is run in mode `push-pop` or `interactive`. In these modes, the context maintains a stack of assertions organized in successive levels. The `(push)` command starts a new assertion level in this stack, and `(pop)` removes all assertions at the current level. The command `(assert f)` adds an assertion `f` to the current level. This assertion will be part of the context until this current level is exited by either a call to `(pop)` or a call to `(reset)`. Thus, a call to `(pop)` retracts all assertions entered since the matching `(push)`. The initial assertion level includes all formulas that are asserted before the first `(push)` command. Such assertions cannot be retracted by `(pop)`. They remain in the context until `(reset)` is called.

The commands `(reset)` and `(pop)` modify the set of assertions in the context, but they do not affect term and type declarations. For example, the following sequence of commands is valid.

```
(push)
(define A::bool)
(assert A)
(check)
(pop)
(assert (not A))
(check)
```

The term `A` is declared after the `(push)` command. The `(pop)` command removes the first assertion but its does not remove the declaration. Thus, `A` remains declared as a Boolean term after the `(pop)` command. The second assertion is then valid. Both calls to `(check)` return `sat`.

Model

If a call to `(check)` returns `sat`, then the set of assertions in the context is satisfiable. One can request `yices` to construct and display a model for the assertions. One can also evaluate the value of arbitrary terms in this model.

The command

```
(show-model)
```

displays the current model (and constructs it if necessary). An error is reported if the context's status is unknown or if the context is not satisfiable. Otherwise, the model is displayed in the format illustrated in Figure 5.9. The model is displayed as a list of assignments, possibly followed by a list of function definitions. An assignment has the form

```
(= name value)
```

where `name` is an uninterpreted constant and `value` is a constant, that is, the value mapped to `name` in the model. This format is used for all terms of atomic types (Boolean, integer and real, bitvector, scalar, and uninterpreted types). It is also used to display the value of terms that have tuple type. The value of an uninterpreted functions f is displayed as shown on the right column of Figure 5.9. For each uninterpreted function, `yices` displays the type of the function, a finite list of assignments, and the function's default value. For example, in Figure 5.9, one can see that `yices` has constructed a model where `(b 0)` and `(b 1)` are `true`, and the default value for `b` is `false`. This means that `(b x)` is `false` for any `x` different from `0` and `1`.

The command

```
(show-reduced-model)
```

was introduced with Yices-2.6.2. It is similar to `get-model` in that it displays a model as a list of assignments, but it tries to first simplify the model by computing a set of relevant variables. The values assigned to these variables are sufficient to ensure that all the current assertions are true. Variables that are not displayed are not necessary to satisfy the assumptions. For example, if you call `yices --mcsat` on the following input:

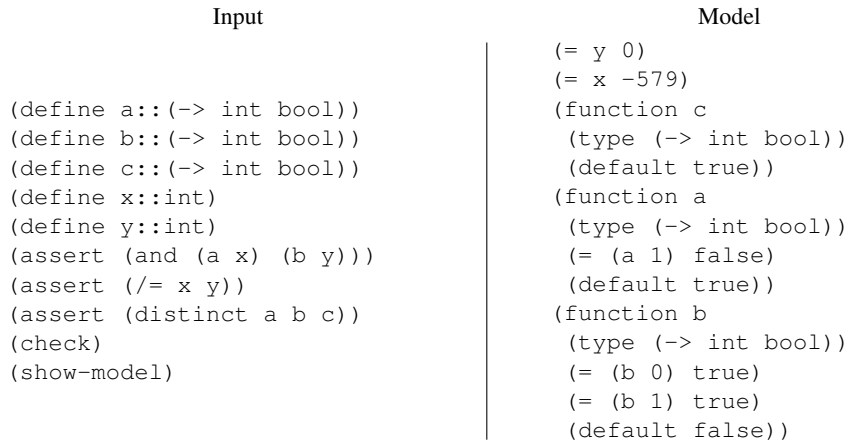

Figure 5.9: Model Display Format

```
(define x::real)
(define y::real)
(assert (= (* x y) 0))
(check)
```

then `(show-model)` will display the value of both `x` and `y`:

```
(= x 0)
(= y 0)
```

whereas `(show-reduced-model)` will display only one of them

```
(= x 0).
```

This assignment is sufficient to ensure the assertion `(= (* x y) 0)` is true, independent of the value assigned to `y`.

Command

```
(eval term)
```

computes the value assigned to `term` in the current model, and displays this value. For example, assuming the model shown in Figure 5.9, one can type

```
(eval (a y))
```

and the result will be `true`. It is also possible to ask for the value of a function term, as in

```
(eval (update a (y) false))
```

The result is displayed as a function specification such as:

```
(function fun!17
  (type (-> int bool))
  (= (fun!17 1) false)
  (= (fun!17 0) false)
  (default true))
```

Yices creates an internal name of the form `fun!<number>` to display the function value.

Unsat Cores

If a call to `(check)` returns `unsat`, then one can get an unsat core by using command

```
(show-unsat-core)
```

The unsat core is a subset of all the labeled assertions in the current context. The command will report an error if there are no labeled assertions in the context. Otherwise, it will list the labels of all the assertions in the core.

Unsat Assumptions

If a check with assumptions returns `unsat`, the following command lists an unsatisfiable subset of the assumptions:

```
(show-unsat-assumptions)
```

The unsatisfiable subset is displayed as a list of assumptions. It is possible for this subset to be empty, if the context is unsatisfiable on its own.

Implicants

If a set of assertions is satisfiable, one can construct an implicant for them. The implicant is a set of literals $l_1, \ldots, l_n$ (i.e., atoms or negations of atoms) such that the conjunct $l_1 \wedge \ldots \wedge l_n$ is satisfiable and implies the assertions. To compute such an implicant, Yices first constructs a model M of the assertions then builds the implicant from the model: all the literals l_i are true in M.

The command to display an implicant is `(show-implicant)`. It can be used only when Yices is executed in mode `one-shot`. Like `(show-model)`, it can be used after a call to `(check)` that returns `sat`. The implicant is displayed as a list of literals, one per line. Figure 5.10 shows an example. The assertion `(distinct x y z)` is not considered atomic in this case. The implicant includes two literals equivalent to $z < y \wedge y < x$, which implies that x, y, and z are distinct.

Input

```
(define x::int)
(define y::int)
(define z::int)
(assert (distinct x y z))
(assert (or (> x (+ z (* 2 y)))
            (< x (- z (* 2 y)))))
(check)
(show-implicant)
```

Implicant

```
(< (+ (* -1 y) z) 0)
(< (+ (* -1 x) (* 2 y) z) 0)
(< (+ (* -1 x) y) 0)
```

Figure 5.10: Implicant

Exists/Forall Solver

The command

```
(ef-solve)
```

checks satisfiability of an exists/forall problem. This command is available when `yices` is run with option `--mode=ef`.

Parameters

A number of parameters controls the preprocessing and simplifications applied by Yices, and the heuristics used by the CDCL SAT solver and the theory solvers. Several commands allow one to examine and modify these parameters.

To see the list of all available parameters, and their current values, type

```
(show-params)
```

If you want to see the value of a specific parameter, type

```
(show-param name)
```

where `name` is the parameter name. To set a parameter value, use

```
(set-param name value)
```

For example, the CDCL solver can use different branching heuristics. This is controlled by the `branching` parameter. To see its current value, type the command

```
(show-param branching)
```

To select a branching heuristic, use a command like

```
(set-param branching negative)
```

Input

```
(define a::(bitvector 4))
(define b::(bitvector 4))
(assert (bv-ge a b))
(export-to-dimacs "test.cnf")
(exit)
```

Result

```
c Autogenerated by Yices
c
c   a --> [6 7 8]
c   b --> [3 4 5]
c
p cnf 10 14
1 0
-2 0
-3 9 0
-5 8 0
-5 -10 0
6 9 0
8 -10 0
-9 -6 3 0
-10 -7 4 0
-4 7 10 0
-10 9 4 0
-4 -9 10 0
-7 9 -10 0
7 10 -9 0
```

Figure 5.11: Export to DIMACS

There are many search and preprocessing parameters. The full list is described in the file `doc/YICES-LANGUAGE` included in the distribution. You can also get on-line help on the parameter using

```
(help params)
```

You can also get on-line help on a specific parameter. For example, the command

```
(help branching)
```

will print a short description of the parameter `branching` and list its possible values.

Conversion to DIMACS

Command

```
(export-to-dimacs file)
```

converts Boolean and bitvector problems to the DIMACS format. This command is supported if `yices` is run with option `--logic=NONE` or `--logic=QF_BV`.

The argument must be the name of a file to store the result. It must be given as a string. The command collects all the assertions and converts them to CNF, then it writes the result into `file`. A mapping from Yices terms to the DIMACS literals is included.

Figure 5.11 shows a small example. The left-hand side is a small bitvector problem. The right-hand side shows the DIMACS file produced by `yices`. The comments shows how the two bitvector variables `a` and `b` are converted to arrays of DIMACS literals. To produce this file, `yices` must be run with option `--logic=QF_BV`.

Timeout

By default, `yices` does not use a timeout. So a call to `(check)` may take a very long time to terminate. To limit the runtime of `(check)`, one can give a timeout is seconds. For example, to limit the runtime to 2 minutes:

```
(set-timeout 120)
```

This timeout will apply to the next call to `(check)`, but not to the one after that. After every call to `(check)`, the timeout is reset to 0 (which means no timeout). One can also clear the timeout explicitly by setting it to 0:

```
(set-timeout 0)
```

To see the current value of the timeout, one can use the command

```
(show-timeout)
```

Echo

The `echo` command can be use to print a string on the standard output. It can be useful in Yices scripts to help display results. An example in Figure 5.12 illustrates its use.

Include

It is possible to include a Yices script using the following command:

```
(include filename)
```

where `filename` is the name of an input file given as a string. For example, to include the file `example.ys`, type

```
(include "example.ys")
```

This command will read and execute all commands contained in the given file.

Help

The `yices` tool has on-line help, which can be obtained using one of the following commands:

```
(help)
(help topic)
```

```
(define a::bool)
(define b::bool)
(define c::bool)
(define d::bool)
(define e::bool)

(assert (= a (or b c)))
(assert (= d (and b c)))
(assert (= a d))
(echo "First check: should be sat\n")
(check)
(show-model)

(assert (= e (xor b c)))
(assert (= e d))
(echo "\nSecond check: should be sat\n")
(check)
(show-model)

(assert d)
(echo "\nThird check: should be unsat\n")
(check)
```

Figure 5.12: Example Use of the `echo` Command

Without argument, `(help)` prints a summary of the main Yices commands. With an argument, `(help topic)` gives help on the specified `topic`. The argument can be a command name, one of the built-in type or term constructor, or the name of a parameter. The argument can be given as a string or as a symbol. For example, to get some information on the search parameter `var-elim`, you can type either

```
(help "var-elim")
```

or just

```
(help var-elim).
```

On-line help is available for other topics such as the syntax. To get a list of all topics, type

```
(help index)
```

Statistics

The solver keeps track of various statistics concerning the search algorithms (e.g., the number of decisions and conflicts in the CDCL solver). The following command prints all the internal statistics

```
(show-stats)
```

As part of these statistics, `yices` keeps track of the cumulative CPU time spent in calls to the `check` command. To get time measurement for a specific call to `(check)` (rather than the total amount of time spent in all calls to `(check)` so far), one can reset the global time counter to zero using command `(reset-stats)`. To get the runtime and other statistics about a specific `(check)`, type the following commands:

```
(reset-stats)
(check)
(show-stats)
```

Exit

At any time, one can exit the solver using the command

```
(exit)
```

If this command is part of a Yices script file, then `yices` exits immediately after this command, without parsing or processing the rest of the file.

Chapter 6

Support for SMT-LIB

The `yices` tool described in the previous chapter processes input given in the Yices 2 language. The distribution includes two other tools that can process input in the SMT-LIB 2.x and the older SMT-LIB 1.2 notations.

6.1 SMT-LIB 2.x

To process SMT-LIB 2.x input, use the `yices-smt2` solver instead of `yices`. This tool is included in the `bin` directory in the distribution. In the Windows or Cygwin distribution, it is called `yices-smt2.exe`.

The SMT-LIB 2.6 language is defined in [BFT17]. More information about the various logics defined in SMT-LIB is available at the SMT-LIB website: `http://www.smtlib.org`. David Cok's tutorial covers all aspects of the language in detail [Cok13].

6.1.1 Tool Invocation

To run `yices-smt2` on an input file, type

```
yices-smt2 <input-file>
```

Since `yices-smt2` runs in mode `one-shot` by default, this will work fine as long as the `<input-file>` does not use the commands `push` and `pop` of SMT-LIB 2 (cf. Section 4.3).

To enable support for `push` and `pop`, give the command-line option `--incremental`. This option configures `yices-smt2` to work in the mode `push-pop`. This flag is also required if the input files contains several blocks of assertions and multiple calls to the command `(check-sat)`.

If no input file is given, `yices-smt2` will read commands from the standard input.

Optionally, one can also run the solver with the following option:

```
yices-smt2 --interactive
```

When invoked in this manner, `yices-smt2` will print a prompt before accepting commands from standard input. In addition, option `:print-success` is set to `true`. This causes `yices-smt2` to report `success` after various commands that would otherwise be executed silently (as required by [BFT15]).

Model Format

Yices has always provided a command `get-model` for displaying a model when assertions are satisfiable. This command was added to SMT-LIB with version 2.5. By default, Yices displays models in a format similar to the one illustrated in Figure 5.9, except that constants are printed with the SMT-LIB 2 Syntax.

Since version 2.6.2, Yices can display models in a different format defined by SMT-LIB (see [BFT17]). To use this format, you must invoke `yices-smt2` as follows

```
yices-smt2 --smt2-model-format
```

Another option controls how bit-vector constants are displayed in `(get-model)` and `(get-value ...)`. By default, `yices-smt2` prints bit-vector values as binary constants (e.g., in the format `#b0110` for a four-bit value). If you prefer to get bit-vector constants converted to integers, use the following option:

```
yices-smt2 --bvconst-in-decimal
```

With this option the solver will use the the SMT-LIB 2 decimal syntax for bit-vector, such as, `(_ bv6 4)`. In general, a bit-vector constant will be displayed as

```
(_ bv<number> <size>)
```

where `<size>` is the number of bits in the constant, and `<number>` is an integer between 0 and $2^{\texttt{<size>}} - 1$.

Selecting a Back-end SAT Solver

For problems in the QF_BV logic, `yices-smt2` relies on so-called *bit blasting* by default. This amounts to converting a bit-vector formula in the `QF_BV` logic into an equivalent Boolean problems in CNF. To solve the resulting CNF formula, Yices will use its internal SAT solver by default.

Since Yices 2.6.2, it is possible to select an alternative Boolean SAT solver as follows:

```
yices-smt2 --delegate=<solver-name>
```

where <solver-name> is either cadical, cryptominisat, kissat, or y2sat. The first three variants select either CaDiCaL, CryptoMiniSat5, or Kissat as backend solver. Support for these third-party solvers must be enabled at compilation time as explained in Chapter 2. The y2sat solver is an experimental SAT solver that is part of the Yices code. It is always available.

For example, to use CaDiCaL on problem bvadd_12000.smt2, type

```
yices-smt2 --delegate=cadical bvadd_12000.smt2
```

This will use CaDiCaL in quiet mode. To get more output from CaDiCaL, increase verbosity:

```
yices-smt2 --delegate=cadical --verbosity=2 bvadd_12000.smt2
```

Note: yices-smt2 can sometimes solve bit-vector problems by preprocessing and simplification, without producing a CNF formula. In such cases, the delegate option is irrelevant.

Limitation: Currently, use of the delegates is restricted to non-incremental problems.

Exporting to Dimacs

In addition to using external SAT solver, you can use yices-smt2 to export the result of bit blasting to a file in the DIMACS CNF format, You can do this as follows:

```
yices-smt2 --dimacs=<outputfile> <inputfile>
```

For example, command

```
yices-smt2 --dimacs=bvadd_12000.cnf bvadd_12000.smt2
```

will process bit-vector problem in file bvadd_12000.smt2, convert it to CNF, and export the result in file bvadd_12000.cnf. This DIMACS file can then be processed by any modern Boolean SAT solver.

You can also use the y2sat SAT solver to simplify the CNF formula before exporting it. This is done by passing an extra command-line option:

```
yices-smt2 --dimacs=<outputfile> --delegate=y2sat <inputfile>
```

In this mode, yices-smt2 will first produce a CNF formula by bit blasting. This formula will then be simplified by applying CNF preprocessing functions implemented by the y2sat SAT solver. The result of this simplification is then exported in DIMACS format.

Note: If `yices-smt2` can solve the problem by preprocessing and simplification, it will report that the instance is either `sat` or `unsat` and it will *not* generate a DIMACS file. Similarly, `y2sat` simplification may solve the CNF formula that results from bit blasting on its own. In such a case, no DIMACS file is produce either.

Limitation: Conversion to DIMACS is not supported for incremental problems.

All Command-line Options

Here is the full list of command-line options supported by `yices-smt2`.

`--verbosity=<level>, -v <level>` Set the initial verbosity level.

By default, `yices-smt2` runs with verbosity level 0. This can be changed by using the SMT command `(set-option :verbosity <level>)`. Calling `yices-smt2 --verbosity=<level>` has the same effect.

`--incremental` Enable support for `push`, `pop`, and multiple calls to `check-sat`.

`--interactive` Run in interactive mode.

This flag has no effect if `yices-smt2` is called with an `input-file`. Otherwise, this flag sets the `:print-success` option to `true`.

`--timeout=<timeout>, -t <timeout>` Give a timeout in seconds.

This sets a timeout for the SMT-LIB command `(check-sat)`. If the timeout is reached, the command returns `unknown`. In non-interactive mode, there can be only one such command. In interactive mode, the same timeout applies to every `(check-sat)`.

`--mcsat` Use the MCSAT solver.

This flag selects the MCSAT solver of Yices instead of the default CDCL-based solver.

`--stats, -s` Display statistics on exit.

If this option is given, `yices-smt2` will print statistics after all commands have been executed (i.e., after reaching the command `(exit)` or the end of the input file).

`--smt2-model-format` Display models in the SMT-LIB 2 format.

`--bvconst-in-decimal` Prints bit-vector constants as numbers (using the SMT-LIB 2 decimal syntax), instead of constants in binary notation.

`--delegate=<solver>` Select an external SAT solver for bit-vector problems,

The `<solver>` can be either `cadical`, or `cryptominisat`, or `y2sat`.

`--dimacs=<filename>` Bitblast then export the CNF to a file (in DIMACS format).

`--version, -V` Print version and exit.

`--help, -h` Show a summary of command-line options and exit.

`--mcsat-help` Show extra options used only by the MCSAT solver.

`--ef-help` Show extra options used only by the Exists-Forall solver.

6.1.2 SMT-LIB 2.6 Compliance

Yices follows the SMT-LIB 2.6 specifications as much as possible. In this section, we list the few special cases where Yices may not adhere to the standard.

Arithmetic

Because Yices uses a more liberal type system than SMT-LIB 2.0, it will accept input that is not strictly compliant with SMT-LIB 2.0. The difference occurs in arithmetic problems. Yices allows formulas to freely mix real and integer terms. In SMT-LIB 2.0, the types `Int` and `Real` are disjoint and cannot be mixed in arithmetic expression. This should not be a problem, as any properly typed SMT-LIB 2.0 arithmetic expression is also type-correct for Yices.

Bitvectors

As mentioned previously, Yices follows the SMT-LIB standard definition for all bit-vector operators except division by zero. The conventions used by Yices are explained in Section 3.3.2.

Unsupported Commands

Some commands defined in SMT-LIB 2.6 are optional. This version of Yices supports the basic commands for declaration and definition of sorts and terms, assertions, satisfiability checking, unsat cores and assumptions. Yices does not support the optional commands `get-assertions` and `get-proof`.

Ignored Options

The standard requires option `:produce-assignments` to be set to `true` before the command `get-assignment` can be issued. It also requires option `:produce-models` to be set to `true` before using the command `get-value`. Yices does not enforce these rules. It supports both commands `get-assignment` and `get-value` even if the corresponding option is `false`.

Unsat Cores and Assumptions

Command `get-unsat-core` and `get-unsat-assumptions` are supported by Yices since version 2.6.0. To activate these commands, options `:produce-unsat-cores` or `:produce-unsat-assumptions` must be set. Yices does not allow both options to be true at the same time.

In-line Definitions

In SMT-LIB 2.x, one can attach annotations to any term. In particular, one can give a name to a term using the syntax

```
(! <term> :named <symbol>)
```

The `<symbol>` is a label attached to `<term>` and marks it as important for the command `get-assignment` and `get-unsat-core`. The standard also requires such an annotation to be treated as an in-line definition. When an annotated subterm `(! <term> :named <name>)` is encountered while parsing a larger term `t0`, then the annotation must be treated as if one had written

```
(define-fun <name> () <sort> <term>)
```

before the term `t0`. This unfortunate decision breaks well-established, common-sense rules about the scope of identifiers. It also means that removing annotations can turn a syntactically correct formula into an incorrect one. It forces SMT-LIB solver to process annotations even if they do not support the commands `get-unsat-core` and `get-assignment`, which were the reason for attaching labels to terms in the first place. Other undesirable consequences include the fact that simple syntactic transformations, for example, rewriting `(or a b)` to `(or b a)`, may be incorrect if `a` contains named annotations. In short, this decision complicates implementation while providing little, if any, benefit.

Still, Yices supports in-line definitions, provided the `<name>` occurring in the annotation is globally fresh. That is, the `<name>` must not be assigned via a previous global definition or by a local `let`. For example, the following monstrosity will cause Yices to complain

```
(assert (let ((x (+ y 1))) (! (P (* 2 x))) :named x)))
```

because the symbol `x` is bound by the enclosing `let` when the annotated term is processed.

Miscellaneous Issues

The SMT-LIB 2.0 document states that option `:print-success` should be `true` by default. This setting requires SMT solvers to report success after any command in a script. This is fine for interactive use, but impractical when reading large input files (such as the SMT-LIB benchmarks at http://www.smtlib.org). These input files typically contain long sequences of declarations and definitions, and printing `success` after each of

them is not useful or informative, and can generate hundreds of thousands if not millions of lines of output. Like other solvers, Yices avoids these issues by setting `:print-success` to `false` by default, unless command-line option `--interactive` is given.

SMT-LIB 2.0 includes two options for directing output and diagnostic information to other channels than the default `stdout` and `stderr`. To send output to a file, you can use the command

```
(set-option :regular-output-channel <filename>)
```

SMT-LIB 2.0 states that `<filename>` should follow the POSIX standard. Yices does not check or enforce this requirement. You can use any character string that can be interpreted as a file name by the underlying operating system.

6.1.3 Solving Modulo a Model

Yices provides a few commands that are not defined in SMT-LIB 2.0 but should be useful. These commands include support for solving modulo a model and generating a model interpolant [JD21]. Figure 6.1 illustrates the use of two non-standard commands:

- `check-sat-assuming-model`: check whether a partial model M can be extended into a full model M' that satisfies all the assertions so far.

  This command takes two arguments. The first argument must be a list of distinct variables and the second argument must be a list of constants. Both lists must have the same length and they define the assumed partial model M.

  For example, the command

  ```
  (check-sat-assuming-model (a b c) (1 1 1))
  ```

  means "check satisfibiliy assuming the model $a = 1$, $b = 1$, and $c = 1$."

- `get-unsat-model-interpolant`: if a partial model M cannot be extended to into a full model of the assertions, this command produces a *model interpolant*. This interpolant is a formula implied by the assertion and is false in M. This formula contains only variables that occur in M and it explains why M is not a good model for the assertions.

  Command `get-unsat-model-interpolant` takes no arguments. It must be called after a call to `check-sat-assuming-model` that returned `unsat`.

The first line of the script in Figure 6.1 sets an option that is enables support for solving modulo a model. This option must occur before the usual `(set-logic ...)` command. Solving modulo a model is currently relies on the MCSAT solver of Yices, so the logic

```
(set-option :produce-unsat-model-interpolants true)

(set-logic QF_NRA)

(declare-const x Real)
(declare-const a Real)
(declare-const b Real)
(declare-const c Real)

;; quadratic equation
(assert (= (+ (* a x x) (* b x) c) 0))

;; check satisfiability assuming a=1, b=1, c=1
(check-sat-assuming-model (a b c) (1 1 1))
(get-unsat-model-interpolant)

;; check satisfiability assuming a=-1, b=1, c=-1
(check-sat-assuming-model (a b c) ((- 1) 1 (- 1)))
(get-unsat-model-interpolant)

;; check satisfibility assuming a=0, b=0, c=1
(check-sat-assuming-model (a b c) (0 0 1))
(get-unsat-model-interpolant)
```

Figure 6.1: Satisfiability Assuming a Model and Model Interpolants

selected must be supported by MCSAT. In the example, the logic is quantifier-free non-linear real arithmetic.

The example contains a single assertion, namely, the quadratic equation

$$ax^2 + bx + c \;=\; 0$$

where variables a, b, c, and x are all reals. Solving assuming a model allows one to discover conditions on a, b, and c under which this quadratic equation has no solution.

For example, the first call to `check-sat-assuming-model` assumes that $a = 1$, $b = 1$, and $c = 1$. Under these conditions, the equation has no solution, so the first `check-sat-assuming-model` will produce `unsat`.

The subsequent call to `get-unsat-model-interpolant` will produce the following formula:

```
(or (>= (* -1 a) 0) (>= (+ (* -4 a c) (^ b 2)) 0))
```

This formula is implied by the assertion and it is false when $a = 1$, $b = 1$ and $c = 1$. It is a symbolic explanation of why the equation has no solutions under our assumptions. The formula rules out the model $a = 1, b = 1, c = 1$ but it generalizes to other models. In more standard mathematical notations, one can read the model interpolant as the well-known necessary condition on the discriminant for solutions to exist:

$$ax^2 + bx + c = 0 \wedge a > 0 \Rightarrow b^2 - 4ac \geq 0.$$

We note that interpolant construction does not always produce the most general explanation; the following would rule out more models

$$ax^2 + bx + c = 0 \wedge a \neq 0 \Rightarrow b^2 - 4ac \geq 0.$$

The second call to `check-sat-assuming-model` is a case where both a and c are negative. The model interpolant in this case is

```
(or (>= (+ (* -4 a c) (^ b 2)) 0) (>= a 0))
```

which can be read as:

$$ax^2 + bx + c = 0 \wedge a < 0 \Rightarrow b^2 - 4ac \geq 0.$$

The two interpolants in conjunction give a necessary condition on the discrimant that holds when the equation is quadratic, that is, when a is not zero.

Finally, the last call to `check-sat-assuming-model` covers a degenerate case, where a and b are both null and c is positive. This time, the model interpolant is

```
(or (>= (* -1 c) 0) (/= b 0) (/= a 0))
```

which is equivalent to

$$ax^2 + bx + c = 0 \wedge a = 0 \wedge b = 0 \Rightarrow c \leq 0.$$

Again, this is not the most general explanation but it rules out the model we started from, namely, $a = 0$, $b = 0$, $c = 1$.

6.2 SMT-LIB 1.2

Another tool included in the distribution can process input written in the SMT-LIB 1.2 notation. This tool is called `yices-smt` (or `yices-smt.exe` on Windows or Cygwin). It is included in the `bin` directory. This tool can process SMT problems written in version 1.2 of SMT-LIB, which is documented in [RT06]. This version of SMT-LIB was used in the SMT competitions before 2010. Since 2010, the competitions have used SMT-LIB 2.0.

6.2.1 Tool Usage

To execute this solver on an input file in the SMT-LIB 1.2 format, just type:

```
yices-smt <input-file>
```

The solver will check satisfiability of the constraints in `input-file` and report either `sat` or `unsat`. The input file must contain a specification in the SMT-LIB *benchmark language* (cf. [RT06]). The standard also defines a *theory language* that is not supported by `yices-smt`. If no input file is given, `yices-smt` will read standard input.

6.2.2 Command-Line Options

The following command-line options can be given to `yices-smt`.

`--model, -m` If this option is given, and the benchmark is satisfiable, `yices-smt` will display a model.

This model may be partial. Some variables of the input benchmark may be eliminated by preprocessing and formula simplification. The value of these variables is not displayed in the model.

`--full-model, -f` Print a full model.

This causes `yices-smt` to display a model if the benchmark is satisfiable. Unlike option `--model`, this option forces Yices to display a complete model. The value of all variables declared in the input benchmark is displayed, even for variables that are eliminated during preprocessing.

`--verbose, -v` Run in verbose mode.

The tool will print various statistics during the search.

`--stats, -s` Show statistics.

This causes `yices-smt` to display statistics about the search, including search time, number of decisions and conflicts, and so forth.

`--timeout=<int>, -t <int>` Give a timeout in seconds.

For example, to run `yices-smt` with a 20 s timeout, use:

```
    yices-smt --timeout=20 ...
```

`--version, -V` Display the version and exit.

`--help, -h` Show a summary of all options and exit.

Chapter 7

Yices API

You can use Yices in your software via a C-API. The main header file is `yices.h` which includes all the API. The API functions are documented in this header file and at https://yices.csl.sri.com/. Since Yices 2.6.0, we also provide Python bindings to this API. These bindings are included in the source distributions in directory `src/bindings`.

As sketched in Figure 4.1, the API provides three main classes of functions:

- Type and term constructors
- Operations on contexts
- Operations on models

The API also includes functions related to error reporting and diagnosis, global initialization and cleanup, and garbage collection.

In the API, types and terms are identified by 32bit signed integers (the types `type_t` and `term_t` are aliases for `int32_t`, as defined in file `yices_types.h`). Other data structures internal to Yices are accessed via opaque pointers. For example, a context is an object of the following type

```
typedef struct context_s context_t;
```

and all functions that operate on contexts take an argument of type `context_t *`.

When an API function fails, it returns a special code. Term constructors return the constant `NULL_TERM`; type constructors return `NULL_TYPE`. Other functions either return a negative integer or the `NULL` pointer. In addition, diagnostic information is stored in a global data structure of type `error_report_t` (defined in `yices_types.h`). The API provides functions to help diagnosis by printing error messages or consulting the error report structure.

```c
#include <stdio.h>
#include <yices.h>

int main(void) {
   printf("Testing Yices %s (%s, %s)\n", yices_version,
          yices_build_arch, yices_build_mode);
   return 0;
}
```

Figure 7.1: Minimal Example

7.1 A Minimal Example

The distribution includes four header files:

- `yices_types.h` defines all types that are part of the API, including a data structure used for error reporting and a set of error codes.
- `yices_limits.h` defines a few constants that set hard limits on the sizes of various constructs. For example, this file defined the maximal arity of functions and the maximal size of bitvector types supported by Yices.
- `yices.h` contains the declaration of all the API functions.
- `yices_exit_codes.h` lists the exit codes that can be returned by the Yices executables (via an `exit` system call).

To use the library, it is enough to include `yices.h` in your code. This will also include `yices_types.h` and `yices_limits.h`.

A minimal example is shown in Figure 7.1. Assuming the Yices library and header files are in standard directories such as `/usr/local/lib` and `/usr/local/include`, this code should compile with the following command:

```
gcc minimal.c -o minimal -lyices
```

Other compilers than GCC can be used. If Yices is installed in a non-standard location, then give appropriate flags to the compilation command. For example, if Yices is installed in your home directory:

```
gcc minimal.c -o minimal -I${HOME}/yices-2.2.0/include \
     -L${HOME}/yices-2.2.0/lib -lyices
```

Running the program should print something like this:

```
Testing Yices 2.2.0 (x86_64-unknown-linux-gnu, release)
```

If you have built a version of Yices that's dynamically linked against GMP, make sure to install GMP on your system. If the Yices library is installed in a non-standard location, you may also need to set environment variable `LD_LIBRARY_PATH` (or `DYLD_LIBRARY_PATH` on Mac OS X).

7.2 Basic API Usage

The distribution includes a few simple examples that illustrate basic use of the Yices library. The code fragments shown in this section come from file `examples/example1.c` included in the distribution.

Global Initialization

Before doing anything with Yices, make sure to initialize all internal data structures by calling function `yices_init`. To avoid memory leaks, you should also call `yices_exit` at the end of your code to free all the memory that Yices has allocated internally.

Term Construction

Figure 7.2 shows code that builds two uninterpreted terms `x` and `y` of type `int`, then constructs the formula

```
(and (>= x 0) (>= y 0) (= (+ x y) 100))
```

This code fragment comes from file `example1.c` that is included in the distribution.

Pretty Printing

Once a term is constructed, we can print it as shown in Figure 7.3. This uses the pretty-printing function `yices_pp_term`. The first argument to this function is the output file to use (in this case, `stdout`). The second argument is the term to print. The other three arguments define the pretty-printing area (in this case, a rectangle of 80 columns and 70 lines). The figure also shows how one checks for errors and prints an error message.

Building a Context and Checking Satisfiability

To check whether formula `f` constructed previously is satisfiable, we create a fresh context, assert formula `f` in this context, then call function `yices_check_context`. This is illustrated in Figure 7.4.

```
// Create two uninterpreted terms of type int.
type_t int_type = yices_int_type();
term_t x = yices_new_uninterpreted_term(int_type);
term_t y = yices_new_uninterpreted_term(int_type);

// Assign names "x" and "y" to these terms.
// This is optional, but we need the names in yices_parse_term
// and it makes pretty printing nicer.
yices_set_term_name(x, "x");
yices_set_term_name(y, "y");

// Build the formula (and (>= x 0) (>= y 0) (= (+ x y) 100))
term_t f = yices_and3(yices_arith_geq0_atom(x),
                      yices_arith_geq0_atom(y),
                      yices_arith_eq_atom(yices_add(x, y),
                                          yices_int32(100)));

// Another way to do it
term_t f_var =
  yices_parse_term("(and (>= x 0) (>= y 0) (= (+ x y) 100))");
```

Figure 7.2: Term Construction using the API

```
static void print_term(term_t term) {
  int32_t code;

  code = yices_pp_term(stdout, term, 80, 20, 0);
  if (code < 0) {
    // An error occurred
    fprintf(stderr, "Error in print_term: ");
    yices_print_error(stderr);
    exit(1);
  }
}

...

// print f and f_var: they should be identical
printf("Formula f\n");
print_term(f);
printf("Formula f_var\n");
print_term(f_var);
```

Figure 7.3: Pretty Printing a Term

```
context_t *ctx = yices_new_context(NULL);
code = yices_assert_formula(ctx, f);
if (code < 0) {
  fprintf(stderr, "Assert failed: code = %"PRId32", error = %"PRId32"\n",
          code, yices_error_code());
  yices_print_error(stderr);
}

switch (yices_check_context(ctx, NULL)) {
case STATUS_SAT:
  printf("The formula is satisfiable\n");
  ...
  break;

 case STATUS_UNSAT:
   printf("The formula is not satisfiable\n");
   break;

 case STATUS_UNKNOWN:
   printf("The status is unknown\n");
   break;

 case STATUS_IDLE:
 case STATUS_SEARCHING:
 case STATUS_INTERRUPTED:
 case STATUS_ERROR:
   fprintf(stderr, "Error in check_context\n");
   yices_print_error(stderr);
   break;
 }
 yices_free_context(ctx);
```

Figure 7.4: Checking Satisfiability

```
model_t *model = yices_get_model(ctx, true);  // get the model
if (model == NULL) {
  fprintf(stderr, "Error in get_model\n");
  yices_print_error(stderr);
} else {
  printf("Model\n");
  code = yices_pp_model(stdout, model, 80, 4, 0); // print the model

  int32_t v;
  // get the value of x, we know it fits int32
  code = yices_get_int32_value(model, x, &v);
  if (code < 0) {
    printf(stderr, "Error in get_int32_value for 'x'\n");
    yices_print_error(stderr);
  } else {
    printf("Value of x = %"PRId32"\n", v);
  }

  // get the value of y
  code = yices_get_int32_value(model, y, &v);
  if (code < 0) {
    fprintf(stderr, "Error in get_int32_value for 'y'\n");
    yices_print_error(stderr);
  } else {
    printf("Value of y = %"PRId32"\n", v);
  }

  yices_free_model(model); // clean up: delete the model
}
```

Figure 7.5: Building and Querying a Model

Building and Querying a Model

If `yices_check_context` returns `STATUS_SAT` (or `STATUS_UNKNOWN`), then we can construct a model of the asserted formulas as shown in Figure 7.5. The code also shows how to print the model and how to evaluate the value of terms in a model.

7.3 Full API

The main header file `yices.h` includes documentation about all API functions. We provide more documentation on the Yices website: https://yices.csl.sri.com/.

Bibliography

[BBL08] R. Brummayer, A. Biere, and F. Lonsing. BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking. In *First International Workshop on Bit-Precise Reasoning*, pages 53–64, 2008. Available at http://fmv.jku.at/BrummayerBiereLonsing-BPR08.pdf.

[BFFH20] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[BFT15] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, SMT-LIB Initiative, 2015. Available at http://www.smtlib.org.

[BFT17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, SMT-LIB Initiative, 2017. Available at http://www.smtlib.org.

[Bie19] Armin Biere. CaDiCaL at the SAT Race 2019. In *Proceedings of SAT Race 2019: Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications*, pages 8–9. University of Helsinki, 2019. https://helda.helsinki.fi//bitstream/handle/10138/308034/sr2019_proceedings.pdf.

[BST12] Clark Barrett, Aaron Sump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, SMT-LIB Initiative, 2012. Available at http://www.smtlib.org.

[Cok13] David R. Cok. The SMT-LIBv2 Language and Tools: A Tutorial. Technical report, GrammaTech, Inc., March 2013. Available at http://www.grammatech.com/resource/smt/SMTLIBTutorial.pdf.

[DdM06a] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer-Aided Verification (CAV'2006)*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer Verlag, August 2006.

[DdM06b] Bruno Dutertre and Leonardo de Moura. Integrating Simplex with DPLL(T). Technical Report SRI-CSL-06-01, Computer Science Laboratory, SRI International, May 2006. Available at http://yices.csl.sri.com/sri-csl-06-01.pdf.

[DFMWP11] David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paelo. Expoiting symmetry in SMT problems. In *Automated Deduction – CADE 23*, volume 6803 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2011.

[dMJ12] Leonardo de Moura and Dejan Jovanović. Solving non-linear arithmetic. In *International Joint Conference on Automated Deduction (IJCAR 2012)*, volume 7364 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2012.

[dMJ13] Leonardo de Moura and Dejan Jovanović. A model-constructing satisfiability calculus. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2013)*, volume 7737 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2013.

[DNS05] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a Theorem Prover for Program Checking. *Journal of the ACM*, 52(3):365–473, May 2005.

[Dut14] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[GSD+14] Adrià Gascón, Pramod Subramanyan, Bruno Dutertre, Ashish Tiwari, Dejan Jovanović, and Sharad Malik. Template-based circuit understanding. In Koen Claessen and Viktor Kuncak, editors, *Formal Methods in Computer-Aided Design (FMCAD 2014)*, pages 83–90, October 2014. Available at http://www.cs.utexas.edu/users/hunt/FMCAD/FMCAD14/proceedings/17_gascon.pdf.

[JBdM13] Dejan Jovanović, Clark Barrett, and Leonardo de Moura. The design and implementation of the model constructing satisfiability calculus. In Barbara Jobstmann and Sandeep Ray, editors, *Formal Methods in Computer-Aided Design (FMCAD 2013)*, pages 173–180, October 2013.

[JD17] Dejan Jovanović and Bruno Dutertre. LIBPOLY: A Library for Reasoning about Polynomials. In *Proceedings of the 15th International Workshop on Satisfiability Modulo Theories (SMT 2017)*, 2017.

[JD21] Dejan Jovanović and Bruno Dutertre. Interpolation and model checking for nonlinear arithmetic, 2021.

[NO79] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[NO07] Robert Neuwenhuis and Albert Oliveras. Fast Congruence Closure and Extensions. *Information and Computation*, 205(4):557–580, April 2007.

[RT06] Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, SMT-LIB Initiative, 2006. Available at http://www.smtlib.org.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing (SAT 2009)*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.

[Som98] Fabio Somenzi. *CUDD: CU Decision Diagram Package*. University of Colorado at Boulder, 1998.

Appendix A

License Terms

GNU GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be

marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

   (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   (d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

(a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

(b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

(c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

(d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

(e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work

in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

   Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   (a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

   (b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

   (c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

   (d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

    In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

    If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <https://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `https://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `https://www.gnu.org/licenses/why-not-lgpl.html`.