SRI International

February 10, 2014

Yices 2 Manual

Bruno Dutertre
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Computer Science Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 326-6200 • 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 in SRI International's Computer Science Laboratory and is distributed free-of-charge for personal use, under the terms of the Yices License (reproduced in Chapter 7 of this manual). To discuss alternative license terms, please contact us at `fm-license@csl.sri.com`.

1.1 Download and Installation

The latest version of Yices 2 can be downloaded at `http://yices.csl.sri.com/download-yices2.shtml`. We provide binary distributions for the platforms and operating systems listed in Table 1.1. To download Yices 2, go to `http://yices.csl.sri.com/download-yices2.shtml` and select the distribution that you want to install. This will open a web page showing the license terms. If you agree to the terms, click on the "accept" button to download a tarfile or zip file. Untar or unzip the file and follow the instructions in the included README file.

OS/Hardware	**Notes**
Linux (32 and 64bit)	Requires Kernel 2.6.8 or more recent
Mac OS X (Intel 32 and 64bit)	For MacOS X 10.5 (Leopard) and more recent
Windows (32bits and 64bit)	Compatible with Windows XP, Vista, 7 and 8
Cygwin (32bit Intel)	
FreeBSD 9.0 (32 and 64bits)	
Solaris 2.10 (Sparc 64bits)	

Table 1.1: Supported Platforms

1.2 Content of the Distribution

The distribution includes the Yices executables, the Yices library and header files, and examples and documentation. Four solvers are currently included in the distribution:

- `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 4.
- `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 2.0 notation [BST12].
- `yices-sat` is a Boolean satisfiability solver that can read input in the DIMACS CNF format.

The Yices library and header files allows you to use Yices via its API, as explained in Chapter 6.

1.3 Library Dependencies

Yices uses the GNU Multiple Precision Arithmetic Library (GMP). We recommend most people to download a Yices distribution that is statically linked against GMP. However, we also provide Yices builds that are linked dynamically against GMP. To use such distributions, you have to install a compatible version of GMP on your machine. GMP-5.0.x or more recent should work. To see the exact GMP version used by Yices, check the README file. The GMP library can be installed using common package managers in most Linux distributions. It can also be built and installed from source. For more information, please visit the GMP website `http://gmplib.org`.

1.4 Supported Logics

The current Yices 2 release supports quantifier-free combinations of linear integer and real arithmetic, uninterpreted function, arrays, and bitvectors. Currently, Yices 2 supports all SMT-LIB logics that do not involve quantifiers or nonlinear arithmetic as summarized in Table 1.2. 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
AUFLIA	Arrays, Linear Integer Arithmetic Quantifiers, Uninterpreted Functions	no
AUFLIRA	Arrays, Mixed Linear Arithmetic Quantifiers, Uninterpreted Functions	no
AUFNIRA	Arrays, Nonlinear Integer Arithmetic Quantifiers, Uninterpreted Functions	no
LIA	Linear Integer Arithmetic, Quantifiers	no
LRA	Linear Real Arithmetic, Quantifiers	no
QF_A	Arrays (without extensionality)	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_LIA	Linear Real Arithmetic	yes
QF_NIA	Nonlinear Integer Arithmetic	no
QF_RDL	Real Difference Logic	yes
QF_UF	Uninterpreted Functions	yes
QF_UFIDL	Uninterpreted Functions, Integer Difference Logic	yes
QF_UFBV	Uninterpreted Functions, Bitvectors	yes
QF_UFLIA	Uninterpreted Functions, Linear Integer Arithmetic	yes
QF_UFLRA	Uninterpreted Functions, Linear Real Arithmetic	yes
QF_UFNRA	Uninterpreted Functions, Nonlinear Real Arithmetic	yes
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. For bug reports and questions about Yices, please 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.

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

  Please include enough information in your bug 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 good bug report 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

Chapter 2

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.

2.1 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.

2.2 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 2.1, 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 does not have a solver for quantified formulas at this point.

2.3 Theories

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

2.3.1 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) with the exception that it does not allow division by a non constant, to avoid issues related to division by zero. For example, the expression $(x + 4y)/3$ is allowed, but

[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 2.1: Primitive Operators and Type Checking

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

The language allows nonlinear polynomials but this is not fully supported by the tool at this time. Yices 2 can solve problems involving real and integer linear arithmetic, but it does not yet include a solver for nonlinear arithmetic.

2.3.2 Bitvectors

Yices supports all the bitvector operators defined in the SMT-LIB standards [RT06,BST12]. The most commonly used operators are listed in Table 2.1. 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`); all of them are supported by Yices 2.

The semantics of all the bitvector operators is defined in the SMT-LIB 1.2 standard. Yices 2 follows the standard except for the case of division by zero. In SMT-LIB, the result of a division by zero is an unspecified value, but one must ensure that the division operators are functional. In other words, SMT-LIB does not specify the result of $(\texttt{bvudiv}\ a\ b)$ if b is the zero vector, but $(\texttt{bvudiv}\ a\ b)$ and $(\texttt{bvudiv}\ c\ b)$ must be equal whenever $a = c$, even if b is the zero vector. Yices 2 uses a simpler semantics (inspired by the BTOR format [BBL08]):

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

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

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

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

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

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

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 form a bitvector of size n

Table 2.1: Bitvector Operators

Chapter 3

Yices 2 Architecture

Yices 2 is based on a modular architecture. Users can select a specific combination of theory solvers for their needs using the API or the `yices` executable. With the API, users can maintain several independent contexts in parallel, possibly each using different solvers and settings.

3.1 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.

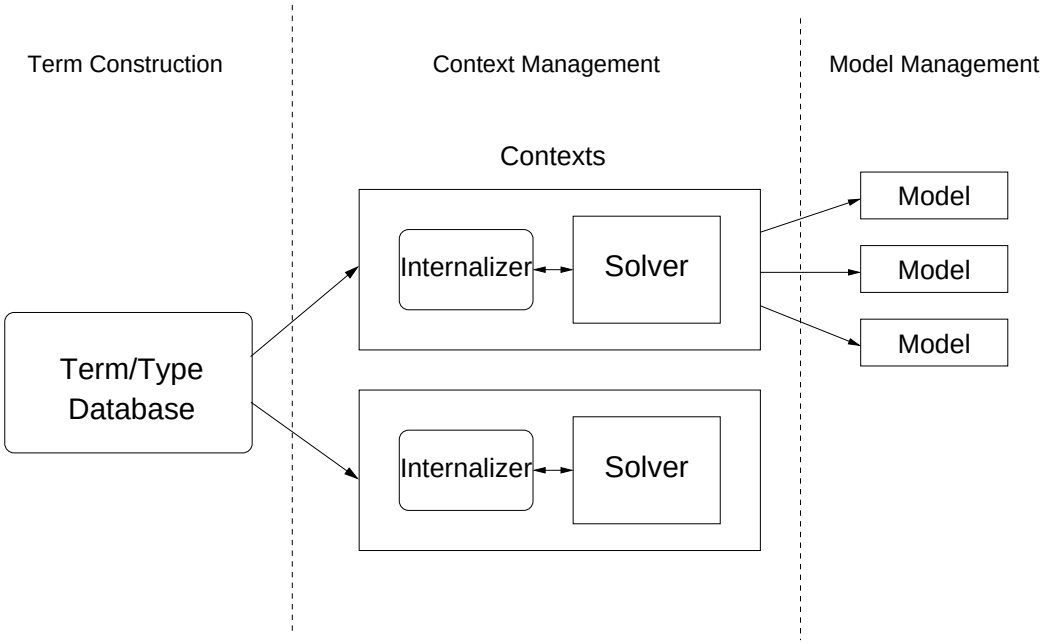

Figure 3.1: Top-level Yices 2 Architecture

Figure 3.1 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 satisfiability solver and decision procedures for determining whether the formulas asserted in the context are satisfiable. The *internalizer* 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.

3.2 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, one can use a solver specialized for linear arithmetic, or use a solver that supports the full Yices 2 language. Figure 3.2 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

[1] UF stands for uninterpreted functions.

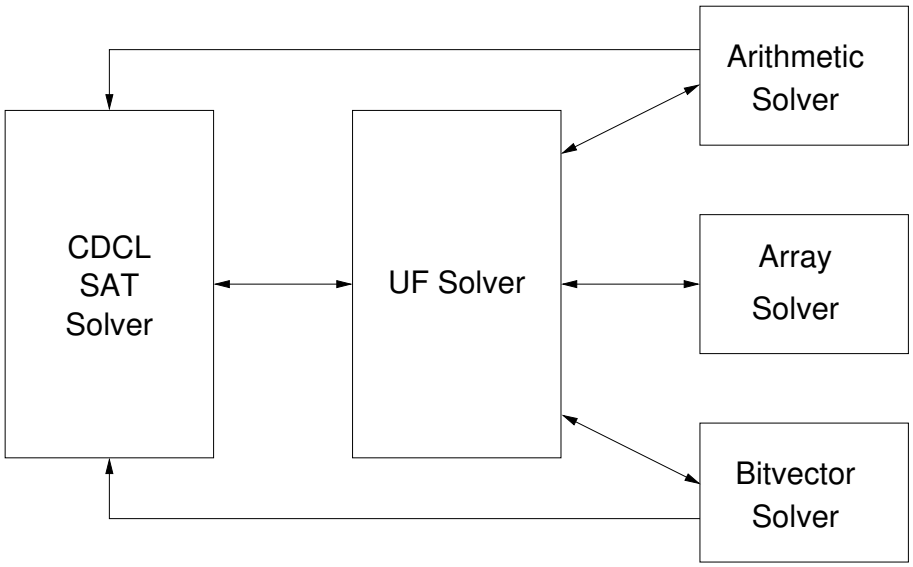

Figure 3.2: Solver Components

implements a decision procedure based on computing congruence closures, similar to the Simplify system [DNS05], with other ideas borrowed from [NO07].

- 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.

In addition, 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 3.2 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 solver 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.

3.3 Context Configurations

In addition to the set of solvers it contains, a context can be configured to support different usage scenarios. The basic operations supported by a context include:

- Asserting one or more formulas
- Checking satisfiability of the set of assertions
- Building a model if the assertions are satisfiable

In addition, a context can be configured to 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 pop 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*. In addition, the context is configured to recover gracefully when a check operation times out or is interrupted.

Chapter 4

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 4.1: Example Yices Script

4.1 Example

To illustrate the tool usage, consider file `examples/bv_test2.ys` shown in Figure 4.1. 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)`, one can select the mode `one-shot`:

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

To get a more detailed output, give option `--verbose`:

```
yices --verbose examples/bv_test2.ys
```

4.2 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 December 2012). 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 3.2. 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).

`--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`, or `interactive`.

This option selects 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.

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.

`--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

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

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

4.3 Input Language

The syntax of the Yices input language is summarized in Figures 4.2, 4.3, and 4.4.

```
<command>  ::=
            ( define-type <symbol> )
          | ( define-type <symbol> <typedef> )
          | ( define <symbol> :: <type> )
          | ( define <symbol> :: <type> <expression> )
          | ( assert <expression> )
          | ( exit )
          | ( check )
          | ( 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> )
          | EOS

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

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

Figure 4.2: Yices Syntax: Commands

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

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

Figure 4.3: 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 4.4: Yices Syntax: Expressions

4.3.1 Lexical Elements

Comments

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

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 4.1) 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 4.1.

*	+	-
->	/	/=
<	<=	<=>
=	=>	>
>=	^	and
assert	bitvector	bool
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	check	define
define-type	distinct	dump-context
echo	eval	exists
exit	false	forall
help	if	include
int	ite	lambda
let	mk-bv	mk-tuple
not	or	pop
push	real	reset
reset-stats	scalar	select
set-param	set-timeout	show-model
show-param	show-params	show-stats
true	tuple	tuple-update
update	xor	

Table 4.1: Keywords and predefined symbols

4.3.2 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.

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.

4.3.3 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 2.1.

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 2.1, 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.

4.3.4 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 2.1, 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 SMTLIB 2, Yices does not allow one to write `(= x y z t u)`. On the other hand, Yices includes an n-ary `distinct` operator (for $n \geq 2$) that generalizes disequality. The Boolean term

```
(distinct <t1> .... <t_n> )
```

is true if <t1>, ..., <t_n> are all different from each other. The terms <t1> to <t_n> must all have compatible types. For example, (distinct <t1> <t2>) is the same thing as (/= <t1> <t2>).

Boolean Operators

true	false
and	or
not	xor
<=>	=>

Table 4.2: Boolean Constants and Operators

The usual Boolean constants and functions are available. They are listed in Table 4.2. The associative/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>).

Arithmetic

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

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 4.3: Arithmetic Operations

The usual arithmetic operations and comparison operators are summarized in Table 4.3. 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. Sect. 3.2).

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

```
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 4.4 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-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 4.4: Bitvector Operations (Arithmetic and Bitwise Logic)

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 4.5: 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 4.6: Bitvector Operations (Structural Operators)

Bitvector Shift and Rotate

Table 4.5 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 4.5) take two bitvector arguments `u` and `v`, which must be bitvectors of the same size n. The shift operation is applied to `u`; 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 4.6 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.
```

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.

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 4.7: Bitvector Operations (Divisions)

Bitvector Division

Table 4.7 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 2.3.2, `(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

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

The last operator in Table 4.7 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 2.3.2.

Bitvector Inequalities

Table 4.8 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.

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 4.8: Bitvector Operations (Comparison)

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 (i1 ... 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 `i1`,...,`i_n` to `v`. The semantics and type-checking rules of this operator are explained in Section 2.2.

4.3.5 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 4.3.2, 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.

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 can also control heuristics employed by the solver. Yices use 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`.

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 4.5. 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 4.5. 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 4.5, 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`.

Input

```
(define a::(-> int bool))
(define b::(-> int bool))
(define c::(-> int bool))
(define x::int)
(define y::int)
(assert (and (a x) (b y)))
(assert (/= x y))
(assert (distinct a b c))
(check)
(show-model)
```

Model

```
(= y 0)
(= x -579)
(function c
 (type (-> int bool))
 (default true))
(function a
 (type (-> int bool))
 (= (a 1) false)
 (default true))
(function b
 (type (-> int bool))
 (= (b 0) true)
 (= (b 1) true)
 (default false))
```

Figure 4.5: Model Display Format

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 4.5, 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.

Parameters

A number of parameters control 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)
```

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)
```

One 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.

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 rest 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)
```

```
(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 4.6: Example Use of the `echo` Command

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 4.6 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>)
```

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`, one 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, one can 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 5

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 2.0 and the older SMT-LIB 1.2 notations.

5.1 SMT-LIB 2.0

To process SMT-LIB 2.0 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 2.0 language is defined in [BST12]. More information about the various logics defined in SMT-LIB 2.0 is available at the SMT-LIB website: `www.smtlib.org`. David Cock's tutorial covers all aspects of the language in detail [Coc13].

5.1.1 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 2.0 (cf. Section 3.3).

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 [BST12]).

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`.

`--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).

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

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

5.1.2 SMT-LIB 2.0 Compliance

Yices follows the SMT-LIB 2.0 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.

Yices does not yet support the operators `div`, `mod`, and `abs` defined in the SMT-LIB theory `Ints`. This will be added in future releases.

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 2.3.2.

Unsupported Commands

Some commands defined in SMT-LIB 2.0 are optional. This version of Yices supports the basic commands for declaration and definition of sorts and terms, assertions, and satisfiability checking. It also implements the commands `push` and `pop`, and the optional commands `get-value` and `get-assignment`. Yices does not support the other optional commands: `get-assertions`, `get-proof`, and `get-unsat-core`.

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`.

In-line Definitions

In SMT-LIB 2.0, 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 solver 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 even 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.

Non-standard Extensions

Yices provides a few commands that are not defined in SMT-LIB 2.0 but should be useful. Similar commands are available in other solvers such as MathSAT 5[1] and Z3[2].

The primary command to examine models in SMT-LIB 2.0 is `get-value`, which prints the value of a list of terms in the current model. It is often more convenient to just display the whole model. Yices provides command `(get-model)` for this purpose. It displays the model in a format similar to the one illustrated in Figure 4.5, except that constants are printed with the SMT-LIB 2.0 syntax.

When Yices is run in incremental mode, a `(reset)` command is available to remove all assertions, declarations, and definitions from the context. This command cannot be used before `set-logic` (cf. [BST12]). It resets the solver to its initial state, but it keeps the logic and all options unchanged.

Yices provides the non-standard command `(echo <string>)` that just prints the given `<string>` on the output channel.

[1] http://mathsat.fbk.eu/smt2examples.html
[2] http://rise4fun.com/z3/tutorial/guide

Global Declarations

Yices supports the option `:global-decls` introduced by MathSAT to control the removal of declarations in incremental mode. By default, `:global-decls` is `false` and Yices follows the SMT-LIB 2.0 conventions. In this mode, the command `(pop ..)` removes all the terms and sorts declared since the matching `(push ...)`. Here is a small example of this default behavior

```
(set-logic QF_UF)
(declare-sort U 0)
(declare-fun a () U)
(push 1)
(declare-fun b () U)
(assert (not (= a b)))
(check-sat)
(pop 1)
(declare-fun f (U) U)
(assert (= a (f b))) ;; error: b is not in scope here
(check-sat)
```

In this small script, the command `(pop 1)` retracts the first assertion and removes the declaration of constant `b`. The second assertion is then incorrect as `b` is now undefined.

Setting `:global-decls` to `true` makes all declarations global and unaffected by `push` and `pop`. A small change to the previous scripts shows the difference:

```
(set-option :global-decls true)
(set-logic QF_UF)
(declare-sort U 0)
(declare-fun a () U)
(push 1)
(declare-fun b () U)
(assert (not (= a b)))
(check-sat)
(pop 1)
(declare-fun f (U) U)
(assert (= a (f b))) ;; correct: b is still in scope here
(check-sat)
```

The second assertion is now correct as the command `(pop 1)` just retracts the first assertion but it does not remove the declaration of `b`. Like other options in SMT-LIB 2.0, the `:global-decls` must be set before `set-logic`.

5.2 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.

5.2.1 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.

5.2.2 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 6

Yices API

As sketched in Figure 3.1, 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.

6.1 A Minimal Example

The distribution includes four include 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.

```
#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 6.1: Minimal Example

- `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` defines 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 6.1. 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 a Yices distribution 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_PAH` on Mac OS X).

6.2 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 `example/example1.c` included in the distribution.

```
// 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 6.2: Term Construction using the API

Global Initialization

Before doing anything with Yices, make sure to initialize all internal data structures by calling function `yices_init`. To avoid memory leaks, it is a good idea to call `yices_exit` when one is done. This frees all the memory that Yices has allocated internally.

Term Construction

Figure 6.2 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 6.3. 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.

```
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 6.3: Pretty Printing a Term

Building a Context and Checking Satisfiability

To check whether formula `f` (or `f_var`) constructed previously is satisfiable, we construct a fresh context, assert formula `f` in this context, then call function `yices_check_context`. This is illustrated in Figure 6.4.

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 6.5. The code also shows how to print the model and how to evaluate the value of terms in a model.

6.2.1 Full API

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

```
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 6.4: 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 6.5: Building and Querying a Model

Chapter 7

License Terms

Before downloading and using Yices, you will be asked to agree to the Yices license terms reproduced below. SRI is open to distributing Yices under other agreements. Contact us at `fm-licensing@csl.sri.com` to discuss alternative license terms.

END-USER LICENSE AGREEMENT

IMPORTANT - READ CAREFULLY. Be sure to carefully read and understand all of the rights and restrictions described in this End-User License Agreement ("EULA"). You will be asked to review and either accept or not accept the terms of the EULA. You will not be permitted to access or use the Software unless or until you accept the terms of the EULA. Alternative license terms may be available to you by contacting fm-licensing@csl.sri.com.

This EULA is a legal agreement between you (either an individual or a single entity) and SRI International ("SRI") for the software referred to by SRI as "Yices", which includes the computer software accessible via this web browser interface, and may include associated media, printed materials and any "online" or electronic documentation ("Software"). By utilizing the Software, you agree to be bound by the terms of this EULA. If you do not agree to the terms of this EULA, you may not access or use the Software.

GRANT OF LIMITED LICENSE. SRI hereby grants to you a personal, non-exclusive, non-transferable, royalty-free license to access and use the Software for your own internal purposes. The Software is licensed to you, and such license does not constitute a sale of the Software. SRI reserves the right to release the Software under different license terms or to stop distributing or providing access to the Software at any time.

RESTRICTIONS. You may not: (i) distribute, sublicense, rent or lease the Software; (ii) modify, adapt, translate, reverse engineer, decompile, disassemble or create derivative works based on the Software; or (iii) create more than one (1) copy of the Software or any related documentation.

OWNERSHIP. SRI is the sole owner of the Software. You agree that SRI retains title to and ownership of the Software and that you will keep confidential and use your best efforts to prevent

and protect the Software from unauthorized access, use or disclosure. All trademarks, service marks, and trade names are proprietary to SRI. All rights not expressly granted herein are hereby reserved.

TERMINATION. The EULA is effective upon the date you first use the Software and shall continue until terminated as specified below. You may terminate the EULA at any time prior to the natural expiration date by destroying the Software and any and all related documentation and copies and installations thereof, whether made under the terms of these terms or otherwise. SRI may terminate the EULA if you fail to comply with any condition of the EULA or at SRI's discretion for good cause. Upon termination, you must destroy the Software in your possession, if any, and any and all copies thereof. In the event of termination for any reason, the provisions set forth under the paragraphs entitled DISCLAIMER OF ALL WARRANTIES, EXCLUSION OF ALL DAMAGES, and LIMITATION AND RELEASE OF LIABILITY shall survive.

U.S. GOVERNMENT RESTRICTED RIGHTS. The Software is deemed to be "commercial software" and "commercial computer software documentation", respectively, pursuant to DFARS §227.702 and FAR 12.212, as applicable. Any use, modification, reproduction, release, performance, display, or disclosure of the Software by the U.S. Government or any of its agencies or by a U.S. Government prime contractor or subcontractor (at any tier) shall have only "Restricted Rights", shall be governed solely by the terms of this EULA, and shall be prohibited except to the extent expressly permitted by the terms of this EULA.

DISCLAIMER OF ALL WARRANTIES. SRI PROVIDES THE SOFTWARE "AS IS" AND WITH ALL FAULTS, AND HEREBY DISCLAIMS ALL OTHER WARRANTIES AND CONDITIONS, EITHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY (IF ANY) IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, OF FITNESS FOR A PARTICULAR PURPOSE, OF LACK OF VIRUSES AND OF LACK OF NEGLIGENCE OR LACK OF WORKMANLIKE EFFORT. ALSO, THERE IS NO WARRANTY OR CONDITION OF TITLE, OF QUIET ENJOYMENT OR OF NON-INFRINGEMENT. THE ENTIRE RISK ARISING OUT OF THE USE OR PERFORMANCE OF THE SOFTWARE IS WITH YOU.

EXCLUSION OF ALL DAMAGES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL SRI BE LIABLE FOR ANY CONSEQUENTIAL, INCIDENTAL, DIRECT, INDIRECT, SPECIAL, PUNITIVE OR OTHER DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR ANY INJURY TO PERSON OR PROPERTY, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, FOR LOSS OF PRIVACY FOR FAILURE TO MEET ANY DUTY INCLUDING OF GOOD FAITH OR OF REASONABLE CARE, FOR NEGLIGENCE AND FOR ANY PECUNIARY OR OTHER LOSS WHATSOEVER) ARISING OUT OF OR IN ANY WAY RELATED TO THE USE OF OR INABILITY TO USE THE SOFTWARE, EVEN IF SRI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THIS EXCLUSION OF DAMAGES SHALL BE EFFECTIVE EVEN IF ANY REMEDY FAILS OF ITS ESSENTIAL PURPOSE.

LIMITATION AND RELEASE OF LIABILITY. SRI has included in this EULA terms that disclaim all warranties and liability for the Software. To the full extent allowed by law, YOU

HEREBY RELEASE SRI FROM ANY AND ALL LIABILITY ARISING FROM OR RELATED TO ALL CLAIMS CONCERNING THE SOFTWARE OR ITS USE. If you do not wish to accept access to the Software under the terms of this EULA, do not access or use the Software. No refund will be made because the SOFTWARE was provided to you at no charge. Independent of, severable from, and to be enforced independently of any other provision of this EULA, UNDER NO CIRCUMSTANCE SHALL SRI'S aggregate LIABILITY TO YOU (INCLUDING LIABILITY TO ANY THIRD PERSON OR PERSONS WHOSE CLAIM OR CLAIMS ARE BASED ON OR DERIVED FROM A RIGHT OR RIGHTS CLAIMED BY YOU), WITH RESPECT TO ANY AND ALL CLAIMS AT ANY AND ALL TIMES ARISING FROM OR RELATED TO THE SUBJECT MATTER OF THIS EULA, IN CONTRACT, TORT, OR OTHERWISE, EXCEED THE TOTAL AMOUNT ACTUALLY PAID BY YOU to SRI pursuant to THIS EULA, IF ANY.

JURISDICTIONAL ISSUES. This Software is controlled by SRI from its offices within the State of California. SRI makes no representation that the Software is appropriate or available for use in other locations. Those who choose to access this Software from other locations do so at their own initiative and are responsible for compliance with local laws, if and to the extent local laws are applicable. You hereby acknowledge that the rights and obligations of the EULA are subject to the laws and regulations of the United States relating to the export of products and technical information. Without limitation, you shall comply with all such laws and regulations, including the restriction that the Software may not be accessed from, used or otherwise exported or reexported (i) into (or to a national or resident of) any country to which the U.S. has embargoed goods; or (ii) to anyone on the U.S. Treasury Department's list of Specialty Designated Nationals or the U.S. Commerce Department's Table of Deny Orders. By accessing or using the Software, you represent and warrant that you are not located in, under the control of, or a national or resident of any such country on any such list.

Notice and Procedure for Making Claims of Copyright Infringement. Pursuant to Title 17, United States Code, Section 512(c)(2), notifications of claimed copyright infringement should be sent to SRI International, Office of the General Counsel, 333 Ravenswood Ave., Menlo Park, CA 94025.

SUPPORT, UPDATES AND NEW RELEASES. The EULA does not grant you any rights to any software support, enhancements or updates. Any updates or new releases of the Software which SRI chooses at its own discretion to distribute or provide access to shall be subject to the terms hereof.

GENERAL INFORMATION. The EULA constitutes the entire agreement between you and SRI and governs your access to and use of the Software. The EULA shall not be modified except in writing by both parties.

The EULA shall be governed by and construed in accordance with the laws of the State of California, without regard to the conflicts of law principles thereof. The parties shall resolve any disputes arising out of this EULA, including disputes about the scope of this arbitration provision, by final and binding arbitration seated and held in San Francisco, California before a single arbitrator. JAMS shall administer the arbitration under its comprehensive arbitration rules and procedures. The arbitrator shall aware the prevailing party its reasonable attorney's fees and expenses, and its arbitration fees and associated costs. Any court of competent jurisdiction may enter judgment on the award.

If any provision of the EULA shall be deemed unlawful, void, or for any reason unenforceable, then that provision shall be deemed severable from these terms and shall not affect the validity and enforceability of any remaining provisions.

In consideration of your use of the Software, you represent that you are of legal age to form a binding contract and are not a person barred from receiving services under the laws of the United States or other applicable jurisdiction.

The failure of SRI to exercise or enforce any right or provision of the EULA shall not constitute a waiver of such right or provision.

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.

[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.

[Coc13] David R. Cock. 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.

[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.

[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.