The RISCTP Theorem Proving Interface

Tutorial and Reference Manual (Version 1.7.*)

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at

November 23, 2023

Abstract

This report documents the RISCTP theorem proving interface. RISCTP consists of a language for specifying proof problems and of an associated software for solving these problems. The RISCTP language is a typed variant of first-order logic whose level of abstraction is between that of higher level formal specification languages (such as the language of the RISCAL model checker) and lower level theorem proving languages (such as the language SMT-LIB supported by various satisfiability modulo theories solvers and theorem provers such as Z3, cvc5, or Vampire). Thus the RISCTP language can serve as an intermediate layer that simplifies the connection of specification and verification systems to theorem provers; in fact, it was developed to equip the RISCAL model checker with theorem proving capabilities. The RISCTP software is implemented in Java with an API that enables the implementation of such connections; however, RISCTP also provides a text-based frontend and a web-based GUI that allow its use as a theorem prover on its own. RISCTP provides a backend that translates a proving problem into SMT-LIB and solves it by the "black box" application of an external SMT solver. Furthermore, RISCTP implements an internal prover for first-order logic with equality and the possibility to inspect successful proofs as well as failed proof attempts. This prover also supports reasoning in the theories of integers, arrays, tuples, and algebraic datatypes, via a corresponding axiomatization, and/or the application of an external SMT solver.

This document will be continuously revised; its most recent version can be found at the following URL: https://www.risc.jku.at/research/formal/software/RISCTP

Contents

1 Introduction **3**

2 Proof Problems **4**
- 2.1 Parsing and Precedence Rules . . . 4
- 2.2 Declarations . . . 5
- 2.3 Names and Overloading . . . 7
- 2.4 Types . . . 8
- 2.5 Expressions . . . 12

3 Proofs **13**
- 3.1 Type-Checking Theorems . . . 13
- 3.2 Further Processing . . . 15
- 3.3 Proving with SMT Solving . . . 16
- 3.4 Proving with MESON . . . 20
- 3.5 The Web GUI . . . 25

4 Conclusions **30**

A The RISCTP Software **33**
- A.1 Installing the Software . . . 33
- A.2 Running the Software . . . 36

B The RISCTP Language **39**
- B.1 Lexical Structure . . . 39
- B.2 Grammar . . . 41

C Examples **44**
- C.1 An Array Problem . . . 44
- C.2 A List Problem . . . 45
- C.3 A First-Order Logic Problem . . . 45
- C.4 A MESON Proof . . . 46

1 Introduction

RISCTP is a theorem proving interface that consists of a language for specifying proof problems and a software for solving these problems. The goal of this interface is to simplify the extension of formal specification and verification systems with theorem proving capabilities, either by connections to external provers or by implementations of internal ones. The concrete motivation for the development of RISCTP has been to supplement the model checking capabilities of the RISCAL software [17, 15, 19, 16, 18, 20], which is able to verify theorems and algorithms in specific finite instances of an infinite class of models, by theorem proving capabilities that allow the verification with respect to the whole model class. For this, we apply the external SMT (satisfiability modulo theories) solvers respectively theorem provers Z3 [6, 7], cvc5 [5, 2], Vampire [22, 9], and ExSpec [13]. However, we also have developed an internal prover for first-order logic with equality that may interact with an external SMT solver.

Since most SMT solvers support the language of the SMT-LIB standard [4, 3], it is natural to consider the SMT-LIB language itself as a suitable interface language. Indeed the RISCTP language is modeled after SMT-LIB in that it supports the following concepts:

- A typed variant of first-order logic (extended with various convenience features).
- Algebraic data types (whose values can be processed by "match" expressions).
- Functional arrays with extensionality (as provided by the SMT-LIB theory "ArraysEx").
- Integer arithmetic (as provided by the SMT-LIB theory "Ints").

However, RISCTP also provides concepts that are not available in SMT-LIB but are extremely helpful in translations from higher level specification languages:

- Overloading of function names (subsequently resolved by renaming to unique names).
- Subtypes constrained by formulas (subsequently translated to explicit predicate applications).
- Tuples with a generic tuple type constructor (internally translated to algebraic types).
- Choose expressions (subsequently translated to axiomatized functions).

Last but not least, the RISCTP language has a concrete syntax (modeled after the syntax of the RISCAL language) that resembles traditional mathematical/logical syntax and allows to specify proof problems in a much more convenient way than in the Lisp-like SMT-LIB language with its fully parenthesized prefix notation. Indeed the RISCTP software provides a parser that translates proof problems from concrete syntax into abstract syntax trees for further processing.

The core of this software is a Java library with an API for the construction of proof problems and an implementation of (currently) two backends:

- An SMT backend that, via translation to SMT-LIB, performs proofs by an external solver.
- A first-order logic backend which translates proof problems into classical first-order logic and applies the proof method "model elimination, subgoal-oriented" (MESON) [11].

The first-order prover also supports equational reasoning and reasoning about the theories of arrays, algebraic datatypes/tuples, and integers, internally via a corresponding axiomatization (which is complete for arrays and algebraic datatypes but incomplete for integers) as well as externally via interaction with an SMT solver.

The RISCTP software is freely available as open source under the GNU General Public License, Version 3 at the following URL:

https://www.risc.jku.at/research/formal/software/RISCTP

Here one can also find the most recent version of this document (which will be continuously revised).

The rest of this paper is structured as follows: Section 2 describes the language of RISCTP by a concrete example that is also used in Section 3 to illustrate the use of the RISCTP software. Section 4 outlines our plans for the further use and development of RISCTP. Appendix A describes the installation of the software and its various execution options. Appendix B gives the concrete grammar for the RISCTP language. Appendix C lists the RISCTP examples used in this document.

2 Proof Problems

In RISCTP, a "proof problem" (short "problem") is a sequence of declarations written as Unicode text. An example of such a problem is given below:

```
// problem file "arrays.txt"
const N:Nat; axiom posN ⇔ N > 0;
type Index = Nat with value < N;
type Value; type Elem = Tuple[Int,Value]; type Array = Map[Index,Elem];
fun key(e:Elem):Int = e.1;
pred sorted(a:Array,from:Index,to:Index) ⇔
  ∀i:Index,j:Index. from ≤ i ∧ i < j ∧ j ≤ to ⇒ key(a[i]) ≤ key(a[j]);
theorem T ⇔
  ∀a:Array,from:Index,to:Index,x:Int.
    from ≤ to ∧ sorted(a,from,to) ⇒
    // let i = (from+to)/2 in
    let i = choose i:Index with from ≤ i ∧ i ≤ to in
    key(a[i]) < x ⇒ ¬∃j:Index. from ≤ j ∧ j < i ∧ key(a[j]) = x;
```

In the following, we will use above example to explain the main features of RISCTP.

2.1 Parsing and Precedence Rules

In a proof problem, expressions (terms and formulas) are parsed according to the usual precedence rules, e.g., arithmetic expressions like $a + b \cdot c$, are interpreted as in $a + (b \cdot c)$. However, there is no universally established convention for precedence rules for all the kinds of expressions supported by RISCTP. These precedence rules are established by the ANTLR4 grammar of the RISCTP language presented in Subsection B.2: within a rule of the grammar, options that appear

earlier have higher precedence. If we are in doubt how RISCTP parses expressions, we may invoke RISCTP with option -p (see Subsection A.2):

```
> RISCTP -p arrays.txt
RISC Theorem Proving Interface 1.7.1 (November 23, 2023)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
-----------------------------------------------------------------
Reading file /usr2/schreine/papers/RISCTP2022/problems/arrays.txt...
=== after parsing:
const N:Nat;
axiom posN ⇔ N > 0;
type Index = Nat with value < N;
type Value;
type Elem = Tuple[Int,Value];
type Array = Map[Index,Elem];
fun key(e:Elem):Int = e.1;
pred sorted(a:Array,from:Index,to:Index) ⇔
  ∀i:Index,j:Index. ((((from ≤ i) ∧ (i < j)) ∧ (j ≤ to)) ⇒
    (key(a[i]) ≤ key(a[j])));
theorem T ⇔
  ∀a:Array,from:Index,to:Index,x:Int.
    (((from ≤ to) ∧ sorted(a,from,to)) ⇒
      (let i = choose i:Index.
        ((from ≤ i) ∧ (i ≤ to)) in ((key(a[i]) < x) ⇒
          (¬(∃j:Index. (((from ≤ j) ∧ (j < i)) ∧ (key(a[j]) = x)))))));
===
=== no proof method selected
FAILURE termination.
```

As we can see, the software prints the problem file after parsing, with parentheses printed around subexpressions; this makes the syntactic interpretation of nested expressions unambiguous.

2.2 Declarations

Declarations introduce the following kinds of entities.

Types The type declaration

```
type Value;
```

introduces a (not further specified) type `Value`. The type definition

```
type Elem = Tuple[Int,Value];
```

introduces a type "Elem" which is defined by the type term `Tuple[Int,Value]`; this type is constructed from the builtin type `Int` (the integer numbers) and the type `Value` by application of the builtin type constructor `Tuple`; it contains all binary tuples whose first component has type `Int` and whose second component has type `Value`. A new type may be also derived from another type by adding a subtype constraint, such as in the declaration

```
type Index = Nat with value < N;
```

which introduces the type `Index` as a type of all values from type `Nat` (the builtin-type of all natural numbers, i.e., non-negative integers) that are smaller than a previously declared constant `N`. In fact, also `Nat` is considered as a a subtype of `Int` with the constraint that a value of this type must be greater equal zero. More details on the RISCTP type system will be given in Subsection 2.4.

Constants The constant declaration

```
const N:Nat;
```

introduces a new constant `N` of type `Nat` without specifying its value. The subsequent constant definition (not in above example)

```
const M:Nat = N+1;
```

defines a new constant `M` of type `Nat` whose value equals `N+1`. In fact, constants are just special cases of functions without arguments (see below).

Functions The function declaration (not in above example)

```
fun f(x:Int,y:Int):Int;
```

introduces a binary function `f` on type `Int` without specifying its result value for given arguments. The function definition

```
fun key(e:Elem):Int = e.1
```

defines a unary function `key` from argument type `Elem` to result type `Int` and specifies that its result, for given argument tuple `e`, is the first tuple component `e.1`.

Predicates The predicate declaration (not in above example)

```
pred p(x:Int,y:Int);
```

introduces a binary predicate "p" on "Int" without specifying its truth value for given arguments. The predicate definition

```
pred sorted(a:Array,from:Index,to:Index) ⇔
  ∀i:Index,j:Index. from ≤ i ∧ i < j ∧ j ≤ to ⇒ key(a[i]) ≤ key(a[j]);
```

introduces a predicate `sorted` and specifies its truth value for given arguments `a`, `from`, and `to` by a formula that states that the elements of array `a` are sorted within the index range with endpoints `from` and `to` in ascending order of the element keys.

In RISCTP, formulas are expressions of the builtin type `Bool` of Boolean values; unlike in classical first-order logic, there is no a priori syntactic difference between formulas and terms (in fact, since predicates are just functions with result type `Bool`, above declaration could have also written as a function definition).

Axioms The definition

```
axiom posN ⇔ N > 0;
```

introduces a named axiom `posN` which constrains by the formula `N > 0` the interpretation of the previously introduced constant `N` of type `Nat` such that it must denote a positive natural number.

Theorems The definition

```
theorem T ⇔
  ∀a:Array,from:Index,to:Index,x:Int.
    from ≤ to ∧ sorted(a,from,to) ⇒
    // let i = (from+to)/2 in
    let i = choose i:Index with from ≤ i ∧ i ≤ to in
    key(a[i]) < x ⇒ ¬∃j:Index. from ≤ j ∧ j < i ∧ key(a[j]) = x;
```

introduces a named theorem `T` which states that for every array `a` that is sorted within the non-empty index range wth endpoints `from` and `to`, at an arbitrarily chosen index `i` in that range, if the key of the element at that index is less than an arbitrary integer `x`, then `x` can also not occur as an element key at any index in the sorted range smaller than `i`.

Before we describe the types and expressions of RISCTP in more detail, we will briefly discuss how entities in RISCTP can be named.

2.3 Names and Overloading

In RISCTP, the names of all declared entities (including locally declared entities such as function parameters and quantified variables) can be plain identifiers (such as `from`) which start with letters and can also contain digits and the underscore character (_); however they can be also quoted identifiers of the form `'...'` where ... can be almost any non-empty sequence of characters; see Subsection B.1. For instance, we may introduce a function `'+1'` by a definition

```
fun '+1'(x:Int):Int = x+1;
```

and later invoke it on argument `a+b` as follows:

```
'+1'(a+b)
```

In fact, the expression `a+b` is just a shortcut for the application

```
'+'(a,b)
```

of a function `'+'`.

Globally declared entities fall into one of three categories: types, functions (including constants and predicates), and formulas (axioms and theorems). Entities in different categories may share the same name, but entities in the same category must have different names (with some exception on function names given below). Thus two types must not have the same name and a theorem and an axiom must not have the same name (both are formulas), but a type and a function may be named alike.

The category of "functions" also includes constants and predicates, because constants are considered as functions without arguments and predicates are considered as functions with result type `Bool`. Two functions may have the same name *provided that* they differ in the number or types of their formal parameters; in this case, the "overloading" of the function name with multiple functions is allowed, because we can distinguish them in all their applications from the number and types of their actual arguments. However, here we consider two types T_1 and T_2 only as different, if one is not derived from the other by a sequence of type definitions $T_1 = \ldots = T_n$ (even if these definitions contain subtype constraints); e.g., the type `Int` of integer numbers and its subtype `Nat` of non-negative integers are considered as the *same* type when it comes to overloading function names.

2.4 Types

All types provide the binary predicates = and ~= (also denoted by the Unicode character "≠") denoting the "equality" and "inequality" of values of these types. In detail, we have the following types and type constructions:

Declared Types A type `T` introduced by a declaration

```
type T;
```

denotes a new type that is different from any other type. Apart from the comparison predicates available on all types, there do not exist any predefined operations on this type.

Defined types A type `T` introduced by a definition

```
type T = ...;
```

does not denote a new type but just another name for the given type `...`. Thus all operations available on type `...` can be also applied to values of type `T`.

Subtypes A type `T` introduced by a definition

```
type T = ... with ...value...;
```

introduces a subtype of type `...`. This subtype consists of every value of the original type that satisfies the formula `...value...` (whose only free variable must be `value`). Thus all operations of type `...` can be also applied to values of type `T`.

The defined subtype must not be empty, i.e., there must exist a value that satisfies the stated formula. When type-checking the declaration, thus a corresponding type checking condition is generated that has to be proved subsequently; see Section 3 for more details.

Booleans `Bool` is the type of the Boolean (truth) values `true` (also denoted by the Unicode character "⊤") and `false` (also denoted by the Unicode character "⊥"). Subsection 2.5 describes various operations on this type that allow to build logical "formulas".

Integers `Int` is the type of integer numbers whose non-negative values are denoted by decimal literals such as `0`, `1`, or `2`. The type provides the usual functions +, - (unary and binary), * (also denoted by the Unicode character "·"), / (truncated quotient) and % (remainder) and

predicates <, <= (also denoted by the Unicode character "≤"), >, and >= (also denoted by the Unicode character "≥").

The value of an expression `e1/e2` or `e1%e2` is undefined if the value of `e2` is `0`. When type checking these expressions, corresponding type-checking conditions are generated as proof obligations; see Section 3 for more details.

`Nat` is a subtype of `Int` that contains the integer numbers with non-negative values only; it is predefined by a declaration

```
type Nat = Int with value ≥ 0;
```

Maps `Map[K,V]` is the type of maps (unary functions) from arguments of type `K` (the "keys") to results of type `V` (the "values"). The expression `m[k]` denotes the value to which key `k` is mapped by map `m`. The expression `map[K,V](v)` denotes a map of type `Map[K,V]` that maps every key to value `v`. The expression `m with [k] = v` denotes the map that is identical to map `m` except that key `k` is mapped to value `v`.

A map of type `Map[Nat,E]` can be considered as an array of elements of type `E`; since every natural number is mapped to an element, the array has infinite length. However, a type `Map[I,E]` with a finite subtype `I` of `Nat` contains only arrays of finite length.

A map of type `Map[E,Bool]` can be considered as a set of elements of type `E`: an element is mapped to `true` if and only if it is in the set.

Tuples `Tuple[`T_1`,...,`T_n`]` is the type of tuples with n components of types T_1, ..., T_n. The expressions `t.1`, ..., `t.`n denote the n components of tuple `t`. The expression `<<`c_1`,...,`c_n`>>` (or $\langle c_1, \ldots, c_n \rangle$ with Unicode characters "⟨" and "⟩" as delimiters) denotes the tuple with n components $c_1, \ldots, c_n$. The expression `t with .i = c` denotes the tuple that is identical to tuple `t` except that component `i` has value `c`.

Every tuple type is internally translated to a corresponding algebraic datatype (see below), e.g., the type `Tuple[Int,Bool]` is translated to the type

```
datatype 'Tuple[Int,Bool]' = '⟨⟩'('.1':Int,'.2':Bool);
```

with constructor `'⟨⟩'` and selectors `'.1'` and `'.2'`.

Algebraic Datatypes A declaration

```
datatype T = c1(s11:T11,...) | ... | cn(sn1:Tn1,...);
```

introduces a new type `T` whose values are constructed by application of one of the n *constructors* `c1`, ..., `cn`. These constructors must all have different names; they denote functions as if they would have been introduced as follows:

```
fun c1(s11:T11,...):T;
...
fun cn(sn1:Tn1,...):T;
```

Thus there must not exist any previously declared function that has the same name and the same number and types of arguments as one of these constructors. If a constructor `c` has no arguments, the parentheses in its declaration are dropped: thus the constructor denotes a constant as if declared as follows:

```
const c:T;
```

The declared type T is an *algebraic datatype*: it consists of exactly those values that can be constructed from applications of its constructors and two of its values are only identical if they have been constructed by application of the same constructor to the same argument.

The type T may appear as the type of a constructor parameter; thus we may from a constructor constant generate arbitrary many values of the type. For instance, the declaration

```
datatype IntList = empty | cons(head:Int,tail:IntList);
```

describes the type of all values that are constructed by an expression of form

```
cons(i1, cons(i2, ..., cons(in, empty)))
```

with integer expressions i1, i2, ..., in; this expression can be identified with an integer sequence $[i_1, i_2 \ldots, i_n]$ where $i_1, i_2, \ldots, i_n$ are the values of the integer expressions. Thus IntList can be considered as the type of all finite lists of integers where empty denotes the empty integer list and cons denotes the function that constructs a new integer list by adding an integer to the front of a previously constructed list.

Each constructor c is equipped with a corresponding *tester* 'is::c', a predicate that, when applied to a value of type T, returns true if and only if that value was constructed by application of the corresponding constructor. For instance, if a value l of above type IntList was constructed as cons(2, cons(1, empty)), then the expression 'is::cons'(l) denotes true while 'is::empty'(l) denotes false.

Furthermore, the constructor parameters s_{ij} become *selectors*, i.e., functions as if they would have been introduced by the following declarations:

```
fun s11(x:T):T11;
...
fun sn1(x:T):Tn1;
...
```

Thus the selectors of all constructors of the type must have different names. Every selector returns, when applied to a value constructed with the constructor in which it was declared, the corresponding constructor argument. For instance, for above value l of type IntList, the expression head(l) denotes 2 and tail(l) denotes cons(1, empty). However, the expressions head(empty) and tail(empty) denote unknown values, because head and tail are not selectors of constructor empty.

If e is an expression of type T, then a "match expression"

```
match e with
| c1(x11:T11,...) -> e1
| c2(x21:T21,...) -> e2
...
| _ -> e0
```

returns the value of expression e1 if e was constructed by application of constructor c1, the value of e2 if e was constructed by application of c2, ..., and the value of e0, otherwise. Here the expression e1 may refer to the variables x11, ... declared in

the constructor pattern `c1(...)`, `e2` may refer to the variables `x21`, ... declared in the constructor pattern `c2(...)`, and so on; these variables are bound to the values of the corresponding selector applications. The order of the constructor patterns need not match the order of the constructor declarations and not all constructors need to appear as patterns. However, if the expression contains a pattern for every constructor, the default branch with the expression e_0 may be omitted.

For instance, the function

```
fun sum(l:IntList):Int;
axiom sumax ⇔ ∀l:IntList.
  sum(l) =
    match l with
    | empty -> 0
    | cons(h:Int,t:IntList) -> h+sum(t);
```

uses in axiom `sumax` a match expression to state that, for every value `l` of type `IntList`, the function application `sum(l)` returns the sum of all values of `l`; this axiom effectively gives a "recursive" definition of function `sum`.

The value of a `match` expression can be also determined by the application of selectors and testers. For instance, above axiom `sumax` could be also defined as follows:

```
fun sum(l:IntList):Int;
axiom sumax ⇔ ∀l:IntList.
  sum(l) =
    if 'is::empty(l)'
      then 0
      else let h = head(l), t = tail(l) in h+sum(t);
```

The expressions `if ... then ... else ...` and `let ... in ...` used in this example will be further explained in Subsection 2.5.

A single `datatype` declaration may also define multiple datatypes in a "mutually recursive" way in the form

```
datatype T1 = ... and ... and Tn = ...;
```

Here we introduce n types `T1`, ... `Tn` where every constructor of every type may have arguments of every other type.

Finally, a datatype may be also constrained to a subtype of a algebraic type by a declaration of form

```
datatype T = ... where ...value... ;
```

where T contains only values admitted by the formula `...value...`. This formula may refer to a function with the following declaration:

```
fun height(x:T):Nat;
```

When applied to a value of type T, this function returns a natural number which may express a measure for the complexity of the construction of the value. However, this function

is only declared; to give it a specific meaning, the problem specification must provide corresponding axioms.

Type Checking When type checking the correct application of functions and predicates, we only consider the "root type" of an expression, ignoring type definitions and subtype declarations. For instance, for the declarations

```
type Number = Int;
type NonZero = Number with value ≠ 0;
```

the root type of a value of type `NonZero` is `Int`; this value can be thus passed to any integer function. Likewise, an argument of type Int can be passed to a function that has a parameter of type NonZero; however, in this case the type checker generates a type checking condition as a proof obligation, see Subsection 3.1 for more details.

2.5 Expressions

In Subsection 2.4, we have described the operations (functions and predicates) that are available on the various types. These operations can be referenced in function applications and atomic formulas which are the fundamental expressions of the language. In this subsection, we will describe how from these expressions larger expressions can be constructed.

Unlike classical first-order logic (and in line with most modern specification and theorem proving languages) RISCTP language does not have separate syntactic categories of "terms" and "formulas" but only a single category of "expressions" where "formulas" are just expressions of type `Bool`. In the following, we describe those constructions that allow to build more complex formulas from simpler ones.

Propositional Formulas From formulas F, F_1, F_2, we can construct the following formulas:

Negation ~F or ¬F ("not F")

Conjunction F_1 /\ F_2 or $F_1 \wedge F_2$ ("F_1 and F_2")

Disjunction F_1 \/ F_2 or $F_1 \vee F_2$ ("F_1 or F_2")

Implication F_1 => F_2 or $F_1 \Rightarrow F_2$ ("if F_1 then F_2")

Equivalence F_1 <=> F_2 or $F_1 \Leftrightarrow F_2$ ("if F_1 then F_2 and vice versa")

The semantics of these logical connectives is that of classical propositional logic.

Quantified Formulas From a name x, type T, and formula F, we can construct these formulas:

Universal Quantification `forall` x:T.F or $\forall x{:}T.F$ ("for all x of type T, F)"

Existential Quantification `exists` x:T.F or $\exists x{:}T.F$ ("for some x of type T, F)"

The semantics of these quantifiers is that of first-order logic extended to a typed framework.

Conditional Expressions For every formula F and expressions E_1 and E_2 of the same type T, the expression

```
if F then E1 else E2
```

is an expression of type T. It denotes the value of E_1, if F is true, and that of E_2 otherwise.

Binder Expressions For all variables $x_1, \ldots, x_n$, expressions $E_1, \ldots, E_n$, and an expression E of type T, the expression

```
let x1 = E1, ..., xn = En in E
```

is an expression of type T. It denotes the value of E when evaluated in a context in which the variables $x_1, \ldots, x_n$ are bound to the values of $E_1, \ldots, E_n$, respectively. These bindings take place *simultaneously* (not in sequence). For instance, in a context where x has value 0, the expression

```
let x=x+1, y=x in ...
```

the expression `...` is evaluated in a context where x has value 1 and y has value 0 (not 1).

Choose Expressions For every name x, type T, and formula F, the expression

```
choose x:T with F
```

denotes any value x of type T that makes formula F true (which may have x as a free variable). If such a value does not exist, the value of the expression is an unknown value of type T. Also the expression

```
choose[sat] x:T with F
```

denotes any value x of type T that makes formula F true. In addition, however, the expression asserts that F is universally satisfiable, i.e., that a value x satisfying F exists for *all* values of the free variables of the expression.

Choose expressions with tag `choose[sat]` give rise to theorems that are simpler to prove; however, if the expression is applied to a formula F that is not universally satisfiable, the resulting theory becomes inconsistent such that every formula becomes provable.

When type checking a choose expression, the type checker generates a corresponding type checking condition as a proof obligation; see Subsection 3.1 for more details.

3 Proofs

When RISCTP is invoked on a proof problem, it first parses the specification to derive an internal representation and then processes the specification in multiple phases, starting with type-checking the specification.

3.1 Type-Checking Theorems

When type-checking a specification, the software may generate auxiliary "type-checking theorems" whose validity has to be subsequently proved in order to ensure that the specification is well-formed:

Subtypes Every type must have at least one value, which is ensured by the pre-defined types and type constructors. However, when the user defines a subtype, this subtype might be empty

because the specified subtype constraint is unsatisfiable. Therefore the system generates a theorem that claims that there exists some value that satisfies the constraint.

Function Definitions When a function (constant, term) is defined, it has to be ensured that the defining expression denotes a value of the result type of the function. If the result type involves a subtype, the system generates a theorem that claims that for all possible argument values, the result value of the function satisfies the subtype constraint.

Function Applications If the type of some function parameter involve a subtype, the argument value provided in a function application for that parameter must satisfy the subtype constraint. Therefore, for every such function application, the system generates a theorem that claims that (considering the context in which this application occurs) this is indeed the case.

Choose Expressions The value of a choose expression is undefined, if no value satisfies the choice formula. Therefore, for every expression with tag `choose` the system generates a theorem that claims that there exists a value that satisfies the formula, for all possible values of the free variables of the expression *that may occur in the context of the expression*. However, for every expression with tag `choose[sat]` the system generates a theorem that claims that there exists a value that satisfies the formula, for all possible values of the free variables of the expression *independently of the context of the expression*.

If RISCTP is invoked with the option `-ptc` it prints the proof problem after type-checking, including all generated type theorems:

```
> RISCTP -ptc arrays.txt
...
=== after type-checking:
...
pred 'Nat::type'(value:Int) ⇔ value ≥ 0;
theorem 'typecheck(Nat)§0' ⇔ ∃value:Int. 'Nat::type'(value);
...
pred 'Index::type'(value:Int) ⇔ 'Nat::type'(value) ∧ (value < N);
theorem 'typecheck(Index)§2' ⇔ ∃value:Int. 'Index::type'(value);
...
theorem 'typecheck(T)§7' ⇔
  ∀a:Array,from:Index,to:Index,x:Int.
    (((from ≤ to) ∧ sorted(a,from,to)) ⇒
      (∃i:Index. ((from ≤ i) ∧ (i ≤ to))));
...
===
=== no proof method selected
FAILURE termination.
```

In above example, we therefore have two type-checking theorems arising from the subtypes `Nat` and `Index` and one theorem arising from the choose expression in theorem `T`.

3.2 Further Processing

After type-checking, the specification is further processed in multiple stages:

Overloading Overloaded functions and predicates are given unique names.

Subtypes Subtypes are replaced by their defining types, but predicate parameters and quantified variables are explicitly constrained by applications of the subtype predicates.

Choose Expressions Every occurrence of a choose expression is replaced by the application of a newly generated "choice function" whose result is not defined but constrained by an axiom generated from the choice formula.

Specification Pruning The specification is pruned such that only those items remain that are relevant for proving the stated theorems. Here two major assumptions are made:

- The problem specification has a model, thus the axioms are not contradictory. This assumption allows to remove axioms that only constrain entities that are not (directly or indirectly) referenced in the theorems to be proved.
- If a function (constant, predicate) is explicitly defined, no axiom that occurs after the definition constrains the interpretation of the function further than the definition alone does. This assumption allows to remove axioms that refer only to defined functions and entities that are irrelevant for proving the stated theorems.

Above steps are those needed for invoking the proof method `smt` (satisfiability modulo theories) that will be described in Subsection 3.3; this method is based on the SMT-LIB language using the theories "ArraysEX" (functional arrays with equality) and "Ints" (integer numbers).

For the proof method `meson` (model elimination, subgoal-oriented) described in Subsection 3.4, however, more is required. First, as a preparation for this method, the type-checker has already generated all the declarations, definitions, and axioms that explicitly characterize the theories of maps, tuples, and algebraic datatypes. While thus the proof method needs not have any more builtin knowledge of these theories, we still assume builtin knowledge of the theories of equality and integer arithmetic; for the entities of these theories, only declarations are generated without further definition or axiomatization. All these additional declarations are also printed when RISCTP is invoked with the option `-ptc` (see Subsection A.2):

```
> RISCTP -ptc arrays.txt
...
=== after type-checking:
...
type Int;
const 'Int§undef':Int;
pred '='(x:Int,y:Int);
pred '≠'(x:Int,y:Int) ⇔ ¬(x = y);
pred '<'(x1:Int,x2:Int);
pred '≤'(x1:Int,x2:Int);
```

```
...
const 0:Int;
...
type 'NonZero§' = Int with value ≠ 0;
...
fun '+'(x1:Int,x2:Int):Int;
fun '-'(x1:Int,x2:Int):Int;
fun '-'(x:Int):Int;
fun '·'(x1:Int,x2:Int):Int;
fun '/'(x1:Int,x2:'NonZero§'):Int;
fun '%'(x1:Int,x2:'NonZero§'):Int;
...
===
=== no proof method selected
FAILURE termination.
```

Second, we transform the language of RISCTP into the language of classical first-order logic by performing the following steps:

Variables In an expression, any occurrence of a name without parameters may denote a variable or a constant. Initially, the parser parses all such names as constants. Now, however, names that are introduced as parameters or bound by quantifiers or binder expressions are transformed from constants to variables.

Datatypes and Match Expressions Datatype declarations are replaced by declared types for which constructors, selectors, and testers are explicitly introduced and axiomatized; furthermore, match expressions are replaced by expressions that apply testers and selectors.

Binder Expressions Binder expressions are removed by explicitly substituting occurrences of the bound variables by their defining terms (appropriately renaming other bound variables whenever necessary).

Function Definitions Function definitions are replaced by function declarations and axioms that describe the values of the functions by universally quantified equalities. After this step, the only occurrences of expressions are in theorems and axioms.

Terms and Formulas Expressions are decomposed into terms and formulas such that the only occurrences of terms are as arguments of functions and predicates; this also entails the translation of conditional expressions to formulas.

The result is therefore a proof problem in the language of classical first-order logic with equality and integer arithmetic.

3.3 Proving with SMT Solving

The RISCTP implementation of the proof method `smt` (satisfiability modulo theories) is mainly based on the SMT-LIB standard [4]. Specifically it requires an SMT solver that understands the

language of SMT-LIB version 2.6 [3] (which includes features such as quantifiers and algebraic datatypes) and that supports the theories "ArraysEX" (functional arrays with equality) and "Ints" (integer numbers). One such solver is the Z3 Theorem Prover developed at Microsoft Research by Nikolaj Bjørner and Leonardo de Moura [6, 7], which is used by RISCTP as the default. Other supported solvers respectively theorem provers are cvc5 [5, 2] and Vampire [22, 9].

Invoking RISCTP with option `-method smt` gives the following output:

```
> RISCTP -method smt arrays.txt
RISC Theorem Proving Interface 1.7.1 (November 23, 2023)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
------------------------------------------------------------------
Reading file /usr2/schreine/papers/RISCTP2022/problems/arrays.txt...
=== SMT solving
SMT solver: Z3 version 4.8.17 - 64 bit
Proving theorem 'typecheck(Nat)§0'...
SUCCESS: theorem was proved (24 ms).
Proving theorem 'typecheck(Index)§2'...
SUCCESS: theorem was proved (5 ms).
Proving theorem 'typecheck(T)§3'...
SUCCESS: theorem was proved (7 ms).
Proving theorem T...
SUCCESS: theorem was proved (12 ms).
===
SUCCESS termination.
```

Thus Z3 could prove all type-checking theorems and the user-provided theorem. If additionally, the option `-psmt` is provided, also the SMT-LIB translation itself is displayed:

```
> RISCTP -method smt -psmt arrays.txt
...
=== SMT-LIB translation
(set-logic ALL)
(define-sort Nat () Int)
(define-fun |Nat::type| ( (value Int) ) Bool (>= value 0))
(push 1)
(assert (not (exists ((value Int)) (|Nat::type| value))))
(check-sat)
(pop 1)
(declare-const N Int)
(assert (|Nat::type| N))
(assert (> N 0))
(define-sort Index () Int)
(define-fun |Index::type| ( (value Int) ) Bool (and (|Nat::type| value) (< value N)))
(push 1)
(assert (not (exists ((value Int)) (|Index::type| value))))
(check-sat)
(pop 1)
(declare-sort Value 0)
```

```
(declare-datatype |Tuple[Int,Value]| ( (|⟨⟩| (|.::1| Int) (|.::2| Value)) ))
(declare-const |Tuple[Int,Value]§undef| |Tuple[Int,Value]|)
(define-sort Elem () |Tuple[Int,Value]|)
(define-fun |Map[Index,Elem]::type| ( (value (Array Int |Tuple[Int,Value]|)) ) Bool
  (forall ((x Int)) (=> (not (|Index::type| x))
    (= (select value x) |Tuple[Int,Value]§undef|))))
(define-sort |§Array§| () (Array Int |Tuple[Int,Value]|))
(define-fun |Array::type| ( (value (Array Int |Tuple[Int,Value]|)) ) Bool (|Map[Index,Elem]::type| value))
(define-fun key ( (e |Tuple[Int,Value]|) ) Int (|.::1| e))
(define-fun sorted ( (a (Array Int |Tuple[Int,Value]|))  (from Int)  (to Int) ) Bool
  (forall ((i Int)) (=> (|Index::type| i)
    (forall ((j Int)) (=> (|Index::type| j) (=> (and (and (<= from i) (< i j)) (<= j to))
      (<= (key (select a i)) (key (select a j)))))))))
(push 1)
(assert (not (forall ((a (Array Int |Tuple[Int,Value]|))) (=> (|Array::type| a)
  (forall ((from Int)) (=> (|Index::type| from)
    (forall ((to Int)) (=> (|Index::type| to)
      (forall ((x Int)) (=> (and (<= from to) (sorted a from to))
        (exists ((i Int)) (and (|Index::type| i) (and (<= from i) (<= i to))))))))))))))
(check-sat)
(pop 1)
(declare-fun |choose§78| ( Int  Int ) Int)
(assert (forall ((from Int)) (forall ((to Int))
  (=> (exists ((i Int)) (and (|Index::type| i) (and (<= from i) (<= i to))))
    (let ( (i (|choose§78| from to)) ) (and (|Index::type| i) (and (<= from i) (<= i to))))))))
(push 1)
(assert (not (forall ((a (Array Int |Tuple[Int,Value]|))) (=> (|Array::type| a)
  (forall ((from Int)) (=> (|Index::type| from)
    (forall ((to Int)) (=> (|Index::type| to)
      (forall ((x Int)) (=> (and (<= from to) (sorted a from to))
        (let ( (i (|choose§78| from to)) )
          (=> (< (key (select a i)) x)
            (not (exists ((j Int)) (and (|Index::type| j)
              (and (and (<= from j) (< j i)) (= (key (select a j)) x))))))))))))))))
(check-sat)
(pop 1)
(exit)
===
...
SUCCESS termination.
```

If it cannot prove a theorem, Z3 generally prints output `unknown`. However, sometimes Z3 can actually *disprove* an alleged "theorem" by producing a model in which the corresponding formula does not hold. In this case Z3 prints output `unsat` together with a description of this model. For instance, an attempt to add to above problem the declaration

```
theorem T0 ⇔ N < 3;
```

yields the following output:

```
Proving theorem T0...
FAILURE: theorem was not proved (20 ms).
theorem T0 ⇔ N < 3;
sat
((N 3))
```

While Z3 is generally very powerful and can solve many interesting proof problems, it also has its limitations: currently, for example, it is not able to prove the existence of maps even if their values are explicitly specified.

Apart from SMT-LIB, RISCTP also provides experimental support for the language of Stefan Ratschan's ExSpec solver [13]. ExSpec is actually a program for "executing algorithm specifications" written in first-order logic with variables ranging over integers and integer arrays (functions from integers to integers). In particular, for a problem with input variable x, output variable y, precondition $P(x)$, and postcondition $Q(x,y)$, it finds values for the variables that satisfy the formula $P(x) \wedge Q(x,y)$; thus, if $P(x)$ fixes input x to some specific value, ExSpec "computes" the corresponding output y. In RISCTP, this can be exploited to show that a conjecture like $P(x) \Rightarrow \neg Q(x,y)$ is not a theorem by demonstrating the satisfiability of its negation $P(x) \wedge Q(x,y)$.

For instance, consider the RISCTP specification for a "theorem" that claims that for a specific array a of length *len* there *not* any sorted version b for any permutation p of the array elements:

```
// file "sort_exspec.txt";
// derived from example "sort_correct1" of exspec distribution:
// specification of the sorting of an integer array of length len to
// an array b of same length using the permutation p
// the satisfiying counterexample provides a,len,b,p

const a:Map[Int,Int];
const len:Int;
const b:Map[Int,Int];
const p:Map[Int,Int];

theorem T ⇔
  len = 4 ∧ a[0] = 3 ∧ a[1] = 1 ∧ a[2] = 0 ∧ a[3] = 3 ⇒
  ¬(
    (∀i:Int. i ≥ 0 ∧ i < len ⇒ p[i] ≥ 0 ∧ p[i] < len ∧ b[i] = a[p[i]]) ∧
    (∀i:Int. i ≥ 0 ∧ i < len ⇒ ∃j:Int. j ≥ 0 ∧ j < len ∧ p[j] = i) ∧
    (∀i:Int. i ≥ 0 ∧ i < len-1 ⇒ b[i] <= b[i+1])
  )
;
```

The following execution of RISCTP with the option `-solver exspec` demonstrates that this is indeed not a theorem by delivering a suitable array b and permutation p:

```
> RISCTP -mode 1 -method smt -solver exspec -path .../exspec sort_exspec.txt
=== SMT solving
SMT solver: exspec 0.3 (Stefan Ratschan)
Proving theorem T...
Translation to ExSpec:
a:ℤ→ℤ, len:ℤ, b:ℤ→ℤ, p:ℤ→ℤ .
  ¬ [[[len=4 ∧ a(0)=3 ∧ a(1)=1 ∧ a(2)=0 ∧ a(3)=3]] ⇒
  ¬ [[∀ i:ℤ . [[[i≥0 ∧ i<len]] ⇒ [p(i)≥0 ∧ p(i)<len ∧ b(i)=a(p(i))]] ∧
      ∀ i:ℤ . [[[i≥0 ∧ i<len]] ⇒ ∃ j:ℤ . [[j≥0 ∧ j<len ∧ p(j)=i]]] ∧
      ∀ i:ℤ . [[[i≥0 ∧ i<len-1]] ⇒ b(i)≤b(i+1)]]]]
```

```
FAILURE: theorem was not proved (3849 ms).
Output of ExSpec:
satisfiable
len: 4
a(0): 3
a(1): 1
a(2): 0
a(3): 3
b(0): 0
b(1): 1
b(2): 3
b(3): 3
p(0): 2
p(1): 1
p(2): 3
p(3): 0

===
FAILURE termination (3853 ms).
```

Compared with SMT-LIB, the language of ExSpec is rather limited in only supporting the datatypes of integers and integer functions. Since also the type of Boolean values is not supported, RISCTP should be invoked with the option `-mode 1` ("without type-checking theorems"), otherwise the translation to ExSpec would fail with an error message.

3.4 Proving with MESON

The proof method `meson` (MESON: model elimination, subgoal-oriented) implements a first-order logic prover, extended by capabilities for equality reasoning and (via suitable axiomatization) reasoning about the theories of maps/arrays, algebraic data types/tuples, and integers. In this section, we explain the use of MESON, see Subsection 3.5 for some explanation of equality and theory reasoning.

MESON applies backward reasoning ("backward chaining") in order to repeatedly reduce a goal to be proved to simpler subgoals until the subgoals are so simple that they can be easily proved from the available knowledge. In the following we give a very rough sketch of the method of model elimination which was invented by Loveland [11] and subsequently reformulated by him as MESON [12]. MESON was later implemented by Stickel on the basis of Prolog technology [21]; our own RISCAL implementation was inspired by the description of Harrison [8] (Section 3.15). For more details, see also the handbook chapter of Letz and Stenz [10].

The goal of MESON is to show that the negation of a theorem is unsatisfiable, which implies the validity of the theorem. For this, the method applies a "Prolog-like" proof search strategy where the negated theorem is translated into a set of clauses that are considered as "rules" of form $(\forall \vec{y}.\ A_1 \wedge \ldots \wedge A_m \Rightarrow B_1 \vee \ldots \vee B_o)$ with quantified variables $\vec{y}$ and atoms (unnegated literals) $A_1, \ldots, A_m, B_1, \ldots, B_o$; however, one clause is negated to become the "goal" $(\exists \vec{x}.\ G_1 \wedge \ldots \wedge G_n)$

with (potentially negated) literals $G_1, \ldots, G_n$. The prover attempts to apply the rules to determine values for the quantified variables $\vec{x}$ that make all the literals in the goal true; this shows that the rules imply the negation of the goal, thus the clauses arising from the negated theorem are contradictory and the negated theorem is unsatisfiable.

For this purpose, the prover picks for every literal G_i some rule $(\forall \vec{y}.\ A_1 \wedge \ldots \wedge A_m \Rightarrow B_1 \vee \ldots \vee B_o)$ where G_i matches some atoms B_j on the right side of the rule or the negation of some atom A_k on the left side; this matching determines a suitable substitution for the variables in G_i. If we assume that the matching literal is B_o, the prover proceeds with the proof of the subgoal $(\exists \vec{y}.\ A_1 \wedge \ldots \wedge A_m \wedge \neg B_1 \wedge \ldots \wedge B_{o-1})$ from which the truth of G_i can be concluded; this process is repeated until the derived subgoal becomes empty, i.e., no more literal remains to be proved. In the proof of the subgoal (and its subgoals) the negation of G_i is kept as an assumption which may be (in addition to the rules) also used to match goal literals.

The whole proof may be visualized as a tree where the root represents the goal, every other node represents a subgoal that is derived from its parent by the application of a rule, and every leaf represent an empty subgoal. Since for every literal multiple matching clauses may exist, the construction of this tree is in general nondeterministic; the prover has to search for the "right" application of clauses that determine suitable values for all the variables in the tree.

To ensure that the proof of a valid theorem is eventually found, the prover organizes its search in the form of "iterative deepening" where a bound on the search tree is imposed, either on its "depth" (the number of rules that have been applied to derive the current subgoal from the root goal) or on its "size" (the total number of rule applications to derive all subgoals in the current proof tree); in the GUI this bound may be set to a fixed value or iteratively incremented from 1 to some maximum value (see also the corresponding command line options `-medepth` and `-mesize`). Furthermore, if the proof with one goal clause does not succeed, the prover has to attempt as a goal also every other clause arising from the theorem to be proved (but not from the axioms stated in the proof problem if we assume that the axioms are not contradictory).

After this brief introduction to MESON, let us return to the RISCTP implementation of this method. By invoking the software with options `-method meson` and `-pat` (print axioms and theorems), we get the following output that shows the core of the generated proof problem:

```
> RISCTP -method meson -pat arrays.txt
RISC Theorem Proving Interface 1.7.1 (November 23, 2023)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
------------------------------------------------------------------
Reading file /usr2/schreine/repositories/RISCTP/trunk/problems/arrays.txt...
=== axioms and theorems:
axiom def§25 ⇔ ∀value:Int. ('Nat::type'(value) ⇔ (value ≥ 0));
theorem typecheck(Nat)§0 ⇔ ∃value:Int. 'Nat::type'(value);
axiom N§type ⇔ 'Nat::type'(N);
axiom posN ⇔ N > 0;
axiom def§42 ⇔ ∀value:Int. ('Index::type'(value) ⇔ ('Nat::type'(value) ∧ (value < N)));
theorem typecheck(Index)§2 ⇔ ∃value:Int. 'Index::type'(value);
axiom inject§45 ⇔ ∀.::1§1:Int,.::1§2:Int,.::2§1:Value,.::2§2:Value. ('=§2'(⟨.::1§1,.::2§1⟩,⟨.::1§2,.::2§2⟩) ⇒ ('=
axiom select§47 ⇔ ∀'.::1':Int,'.::2':Value. '=§0'(⟨'.::1','.::2'⟩.1,'.::1');
axiom select§49 ⇔ ∀'.::1':Int,'.::2':Value. '=§1'(⟨'.::1','.::2'⟩.2,'.::2');
axiom datatype§51 ⇔ ∀§x:Tuple[Int,Value]. (∃'.::1':Int,'.::2':Value. '=§2'(§x,⟨'.::1','.::2'⟩));
axiom ext§60 ⇔ ∀m1:Map[Int,Tuple[Int,Value]],m2:Map[Int,Tuple[Int,Value]]. ((∀k:Int. '=§2'(m1[k],m2[k])) ⇒ '=§3'
```

```
axiom def§89 ⇔ ∀value:Map[Int,Tuple[Int,Value]]. ('Map[Index,Elem]::type'(value) ⇔ (∀x:Int. ((¬'Index::type'(x))
axiom def§92 ⇔ ∀value:Map[Int,Tuple[Int,Value]]. ('Array::type'(value) ⇔ 'Map[Index,Elem]::type'(value));
axiom def§94 ⇔ ∀e:Elem. '=§0'(key(e),e.1);
axiom def§96 ⇔ ∀a:Array,from:Index,to:Index. (sorted(a,from,to) ⇔ (∀i:Index. ('Index::type'(i) ⇒ (∀j:Index. ('I
theorem typecheck(T)§3 ⇔ ∀a:Array. ('Array::type'(a) ⇒ (∀from:Index. ('Index::type'(from) ⇒ (∀to:Index. ('Index
axiom choose§79 ⇔ ∀from:Index,to:Index. ((∃i:Index. ('Index::type'(i) ∧ ((from ≤ i) ∧ (i ≤ to)))) ⇒ ('Index::ty
theorem T ⇔ ∀a:Array. ('Array::type'(a) ⇒ (∀from:Index. ('Index::type'(from) ⇒ (∀to:Index. ('Index::type'(to) =
===
=== proof method 'meson': model elimination (subgoal-oriented)
Goal: ∀value:Int. 'Nat::type'(value) ⇒ ⊥
Iteration 1 (proof depth 1)...
SUCCESS: the proof problem has been solved (11 clause applications).
===
=== proof method 'meson': model elimination (subgoal-oriented)
Goal: ∀value:Int. 'Index::type'(value) ⇒ ⊥
Iteration 1 (proof depth 1)...
...
The proof is aborted because the timeout limit has been reached.
FAILURE: the proof problem has NOT been solved (7871672 clause applications).
===
=== proof method 'meson': model elimination (subgoal-oriented)
Goal: ∀i3:Index. 'Index::type'(i3) ∧ ≤(from0,i3) ∧ ≤(i3,to0) ⇒ ⊥
Iteration 1 (proof depth 1)...

The proof is aborted because the timeout limit has been reached.
FAILURE: the proof problem has NOT been solved (15775494 clause applications).
===
=== proof method 'meson': model elimination (subgoal-oriented)
Goal: ⊤ ⇒ '=§0'(key([](a0,j1)),x1)
Iteration 1 (proof depth 1)...
...
Goal: ⊤ ⇒ <(j1,choose§78(from0,to0))
...
The proof is aborted because the timeout limit has been reached.
FAILURE: the proof problem has NOT been solved (17089908 clause applications).
===
FAILURE termination (180898 ms).
```

In this example, only one subproblem arising from the type checking theorem could be solved; the other subproblems depend on arithmetic properties of the integer numbers that require some more machinery which is described in Subsection 3.5.

Nevertheless, the prover is indeed able to prove non-trivial first-order theorems such as the one described by the following proof problem:

```
// problem file "fol.txt"
type T;
pred p(x:T);
pred q(x:T);
pred r(x:T);
fun f(x:T):T;
theorem Theorem ⇔
  ∀a:T,b:T.
    (p(a) ∨ q(b)) ∧
    (∀x:T. p(x) ⇒ r(x)) ∧
    (∀x:T. q(x) ⇒ r(f(x))) ⇒
```

```
(∃x:T. p(x) ∨ q(x)) ∧ (∃x:T. r(x));
```

The proof succeeds after a few rounds of iterative deepening:

```
RISC Theorem Proving Interface 1.7.1 (November 23, 2023)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
------------------------------------------------------------------
Reading file /usr2/schreine/repositories/RISCTP/trunk/problems/fol.txt...
=== proof method 'meson': model elimination (subgoal-oriented)
Goal: ∀x6:T,x8:T. q(x6) ∧ r(x8) ⇒ ⊥
Iteration 1 (proof depth 1)...
Iteration 2 (proof depth 2)...
Iteration 3 (proof depth 3)...
Iteration 4 (proof depth 4)...
SUCCESS: the proof problem has been solved (64 clause applications).
===
SUCCESS termination (29 ms).
```

However, such proofs usually become easier to perform (and to understand) if the software is started with the option `-decompose`; then the problem is first (before the translation of the negated theorem into clauses) decomposed into subproblems by applying the rules of the sequent calculus. If we add the option `-pdec 1`, the decomposition process is printed (with low verbosity); if we also add the option `-pcla`, the clause forms of the resulting subproblems are displayed:

```
> RISCTP -method fol -pat -decompose -pdec 1 -pcla arrays.txt
RISC Theorem Proving Interface 1.7.1 (November 23, 2023)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
------------------------------------------------------------------
Reading file /usr2/schreine/repositories/RISCTP/trunk/problems/fol.txt...
=== decomposition of proof problem into subproblems
=== Theorem (rule [∀-R | ∃-L] applied to the goal creates 1 subproblem)
  === Theorem.1 (rule [∀-R | ∃-L] applied to the goal creates 1 subproblem)
    === Theorem.1.1 (rule [⇒-R | ∨-R | ∧-L] applied to the goal creates 1 subproblem)
      === Theorem.1.1.1 (rule [∧-R | ∨-L | ⇒-L] applied to the goal creates 2 subproblems)
        === Theorem.1.1.1.1 (rule [⇒-R | ∨-R | ∧-L] applied to formula [1] creates 1 subproblem)
          === Theorem.1.1.1.1.1 (rule [⇒-R | ∨-R | ∧-L] applied to formula [1] creates 1 subproblem)
            === Theorem.1.1.1.1.1.1 (rule [∧-R | ∨-L | ⇒-L] applied to formula [1] creates 2 subproblems)
              === Theorem.1.1.1.1.1.1.1 (open)
              === Theorem.1.1.1.1.1.1.2 (open)
        === Theorem.1.1.1.2 (rule [⇒-R | ∨-R | ∧-L] applied to formula [1] creates 1 subproblem)
          === Theorem.1.1.1.2.1 (rule [⇒-R | ∨-R | ∧-L] applied to formula [1] creates 1 subproblem)
            === Theorem.1.1.1.2.1.1 (rule [∧-R | ∨-L | ⇒-L] applied to formula [1] creates 2 subproblems)
              === Theorem.1.1.1.2.1.1.1 (open)
              === Theorem.1.1.1.2.1.1.2 (open)
decomposition created 4 subproblems.
=== clause form of problem
```

```
type T;
pred p(x:T);
pred q(x:T);
pred r(x:T);
fun f(x:T):T;
const a0:T;
const b0:T;
theorem Theorem.0.0.2.1 ⇔ ∃x:T. (p(x) ∨ q(x));

----------------------------------------
1:[Theorem.0.0.1.1.1.1] ⊤ ⇒ p(a0)
2:[Theorem.0.0.1.1.2] ∀x:T. p(x) ⇒ r(x)
3:[Theorem.0.0.1.2] ∀x:T. q(x) ⇒ r(f(x))
4:[Theorem.0.0.2.1.0] ∀x:T. p(x) ⇒ ⊥
5:[Theorem.0.0.2.1.1] ∀x:T. q(x) ⇒ ⊥
=== proof method 'meson': model elimination (subgoal-oriented)
Goal: ∀x:T. q(x) ⇒ ⊥
Iteration 1 (proof depth 1)...
Iteration 2 (proof depth 2)...
Iteration 3 (proof depth 3)...
Iteration 4 (proof depth 4)...
Iteration 5 (proof depth 5)...
Iteration 6 (proof depth 6)...
Iteration 7 (proof depth 7)...
Iteration 8 (proof depth 8)...
Iteration 9 (proof depth 9)...
Iteration 10 (proof depth 10)...
Goal: ∀x:T. p(x) ⇒ ⊥
Iteration 1 (proof depth 1)...
SUCCESS: the proof problem has been solved (1 clause applications).
===
=== clause form of problem
type T;
pred p(x:T);
pred q(x:T);
pred r(x:T);
fun f(x:T):T;
const a0:T;
const b0:T;
theorem Theorem.0.0.2.1 ⇔ ∃x:T. (p(x) ∨ q(x));

----------------------------------------
1:[Theorem.0.0.1.1.1.2] ⊤ ⇒ q(b0)
2:[Theorem.0.0.1.1.2] ∀x:T. p(x) ⇒ r(x)
3:[Theorem.0.0.1.2] ∀x:T. q(x) ⇒ r(f(x))
4:[Theorem.0.0.2.1.0] ∀x:T. p(x) ⇒ ⊥
5:[Theorem.0.0.2.1.1] ∀x:T. q(x) ⇒ ⊥
=== proof method 'meson': model elimination (subgoal-oriented)
Goal: ∀x:T. q(x) ⇒ ⊥
Iteration 1 (proof depth 1)...
SUCCESS: the proof problem has been solved (1 clause applications).
===
=== clause form of problem
type T;
pred p(x:T);
pred q(x:T);
pred r(x:T);
fun f(x:T):T;
const a0:T;
const b0:T;
theorem Theorem.0.0.2.2 ⇔ ∃x:T. r(x);
```

```
---------------------------------------
1:[Theorem.0.0.1.1.1.1] ⊤ ⇒ p(a0)
2:[Theorem.0.0.1.1.2] ∀x:T. p(x) ⇒ r(x)
3:[Theorem.0.0.1.2] ∀x:T. q(x) ⇒ r(f(x))
4:[Theorem.0.0.2.2] ∀x:T. r(x) ⇒ ⊥
=== proof method 'meson': model elimination (subgoal-oriented)
Goal: ∀x:T. r(x) ⇒ ⊥
Iteration 1 (proof depth 1)...
SUCCESS: the proof problem has been solved (3 clause applications).
===
=== clause form of problem
type T;
pred p(x:T);
pred q(x:T);
pred r(x:T);
fun f(x:T):T;
const a0:T;
const b0:T;
theorem Theorem.0.0.2.2 ⇔ ∃x:T. r(x);

---------------------------------------
1:[Theorem.0.0.1.1.1.2] ⊤ ⇒ q(b0)
2:[Theorem.0.0.1.1.2] ∀x:T. p(x) ⇒ r(x)
3:[Theorem.0.0.1.2] ∀x:T. q(x) ⇒ r(f(x))
4:[Theorem.0.0.2.2] ∀x:T. r(x) ⇒ ⊥
=== proof method 'meson': model elimination (subgoal-oriented)
Goal: ∀x:T. r(x) ⇒ ⊥
Iteration 1 (proof depth 1)...
SUCCESS: the proof problem has been solved (2 clause applications).
===
SUCCESS termination (20 ms).
```

If invoked with the option `-pproof`, also the generated proof tree is printed (but note that the generation of the proof tree slows down the proof search); Appendix C.4 displays the proof arising from above proof problem. Furthermore, if invoked with the option `-psearch`, also the proof search process itself is displayed (this slows down the proof search even more and also requires much more memory). However, the amount of output quickly becomes overwhelming; for this kind of investigations, better the graphical user interface should be applied, as described in the next subsection.

3.5 The Web GUI

While all features of RISCTP can be accessed via the command line, the software also provides a graphical user interface that is recommended for interactive use. If executed with command line option `-web 9999 1`, RISCTP shows the following output:

```
> RISCTP -web 9999 1
RISC Theorem Proving Interface 1.7.1 (November 23, 2023)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
-----------------------------------------------------------------
```

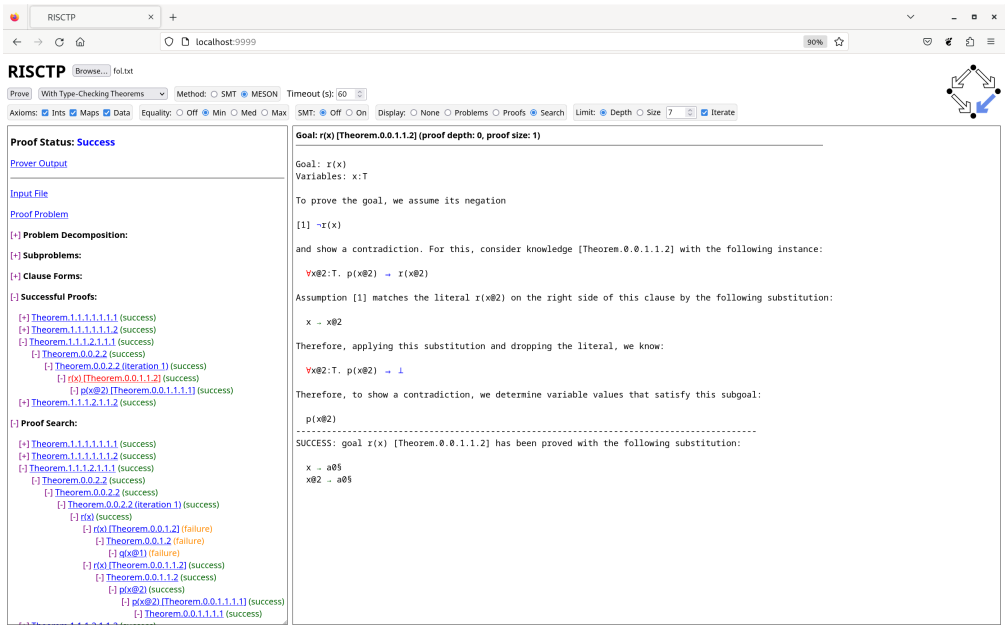

Figure 1: The RISCTP Web GUI

```
RISCTP GUI can be browsed at http://localhost:9999/
Press <Enter> to terminate the server.
```

By the option `-web` P B, we start a web server that presents at URL `http://localhost:`P`/` a graphical user interface (GUI) which can be accessed by a web browser; argument $B = 1$ equips the GUI with buttons to control the execution of the software. Figure 1 depicts the status of the GUI after file `fol.txt` has been loaded by pressing the button "Browse" and the prover has been started by pressing the button "Prove" with options "Method: MESON" and "Display: Search" selected (the other options do not really matter here, they will be explained below with another example). The loaded file contains the following proof problem (which was already presented in the previous subsection):

```
// problem file "fol.txt"
type T;
pred p(x:T);
pred q(x:T);
pred r(x:T);
fun f(x:T):T;
theorem Theorem ⇔
  ∀a:T,b:T.
    (p(a) ∨ q(b)) ∧
    (∀x:T. p(x) ⇒ r(x)) ∧
    (∀x:T. q(x) ⇒ r(f(x))) ⇒
```

```
(∃x:T. p(x) ∨ q(x)) ∧ (∃x:T. r(x));
```

The GUI depicts the proof tree arising from the proof of a subproblem that arose from the decomposition of the original problem according to the rules of the sequent calculus; the tree is visualized as a nested list where each entry denotes a subproblem which can be expanded and collapsed by clicking on the tag `[-]` or `[+]`, respectively. By clicking on the blue link in an entry, the link turns red and the corresponding proof situation is shown in the right pane; the relative width of the panes can be resized by a marker on the lower right corner of the left pane.

The elements of the user interface are continuously active and may be even used while the RISCTP software is still busy with solving a proof problem (the "Run" button turns into an "Abort" button that may be used to interrupt the attempt; in any case, the attempt stops after the selected "Timeout" value has been reached). The link "Prover Output" shows the output produced by the software so far (to see the progress of the output, the link has to be clicked again). The link "Input File" shows the content of the selected file. The header "Proof Problem" prints the proof problem after parsing the file (with all parentheses inserted to clarify the formula structure). The header "Problem Decomposition" displays the individual steps of decomposing the problem into subproblems. The header "Subproblems" illustrates the subproblems after decomposition. The header "Successful Proofs" shows the steps steps that lead to successful proofs. The header "Proof Search" enumerates all the steps performed in the proof search. The overall proof status is depicted at the top; the status "Success" indicates that all subproblems could be proved.

Please note, that selecting the option "Display: Proof" slows down the proof search due to the overhead of creating the proof tree. Selecting the option "Display: Search" slows down the search even more and also requires a lot more memory; so these options should be used with care.

After this presentation of a proof in pure-first order logic, we now return to our standard problem which also involves arrays and other data types:

```
// problem file "arrays.txt"
const N:Nat; axiom posN ⇔ N > 0;
type Index = Nat with value < N;
type Value; type Elem = Tuple[Int,Value]; type Array = Map[Index,Elem];
fun key(e:Elem):Int = e.1;
pred sorted(a:Array,from:Index,to:Index) ⇔
  ∀i:Index,j:Index. from ≤ i ∧ i < j ∧ j ≤ to ⇒ key(a[i]) ≤ key(a[j]);
theorem T ⇔
  ∀a:Array,from:Index,to:Index,x:Int.
    from ≤ to ∧ sorted(a,from,to) ⇒
    // let i = (from+to)/2 in
    let i = choose i:Index with from ≤ i ∧ i ≤ to in
    key(a[i]) < x ⇒ ¬∃j:Index. from ≤ j ∧ j < i ∧ key(a[j]) = x;
```

Its solution by the MESON prover is demonstrated in Figure 2. Here the option "Axioms: Ints" has been selected, which adds to the problem a number of axioms that characterize the builtin integer operations, in particular the predicates ">", "<", "≤" on which the sample specification depends. With the help of these axioms, the MESON prover indeed finds proofs for all three type checking axioms and the core theorem of the theory. The proof of the theorem depends in particular on the transitivity of "≤" (axiom `trans2<=`), as well as on some other properties.

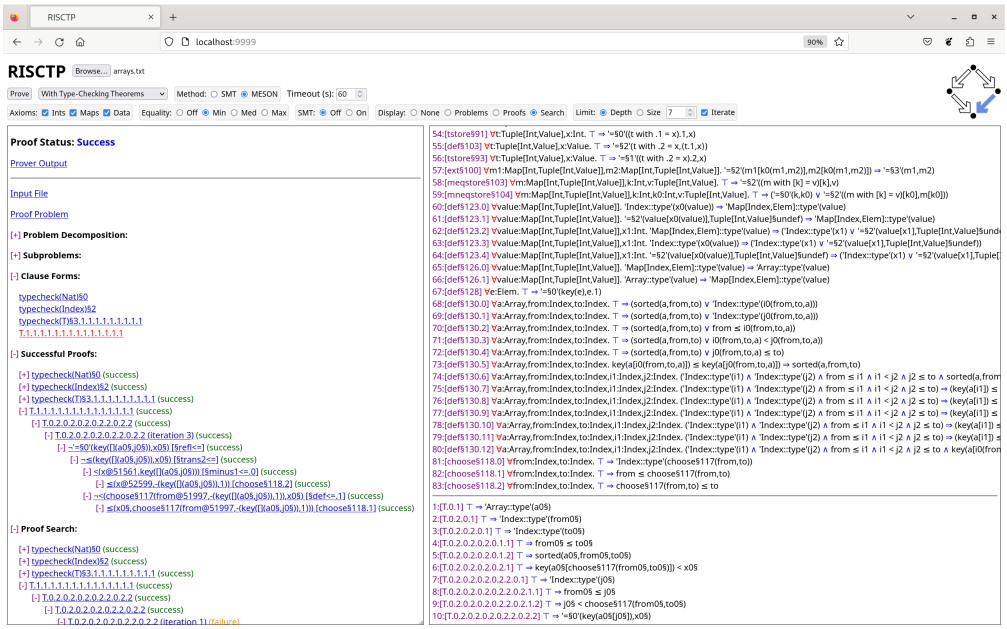

Figure 2: The Arrays Proof with MESON

Additionally we have also selected the options “Axioms: Maps” and “Axioms: Data” which add axiomatizations of array update operations and algebraic data type constructions (including tuple constructions), respectively. Also the option “Equality:Min” has been selected, which switches on some form of equality reasoning. However, these are actually not required in our sample problem (and the proof search can be made a bit faster by switching them off).

However, the options “Axioms: Int”, “Axioms: Data”, and “Equality: Min” are needed to solve the following proof problem:

```
// problem file "lists.txt"
datatype IntList = empty | cons(head:Int,tail:IntList);
fun sum(l:IntList):Int;
axiom sumax ⇔ ∀l:IntList.
  sum(l) =
    match l with
    | empty -> 0
    | cons(h:Int,t:IntList) -> h+sum(t);
theorem T ⇔
  let l = cons(1, cons(2, empty)) in sum(l) = 3;
```

Here we introduce the algebraic data type of integer lists and a function sum whose axiomatization essentially describes a recursive definition of the function which computes the sum of all elements of a given list. The theorem states that the sum of the list with elements 1 and 2 is 3; the proof of this theorem is illustrated in Figure 3.

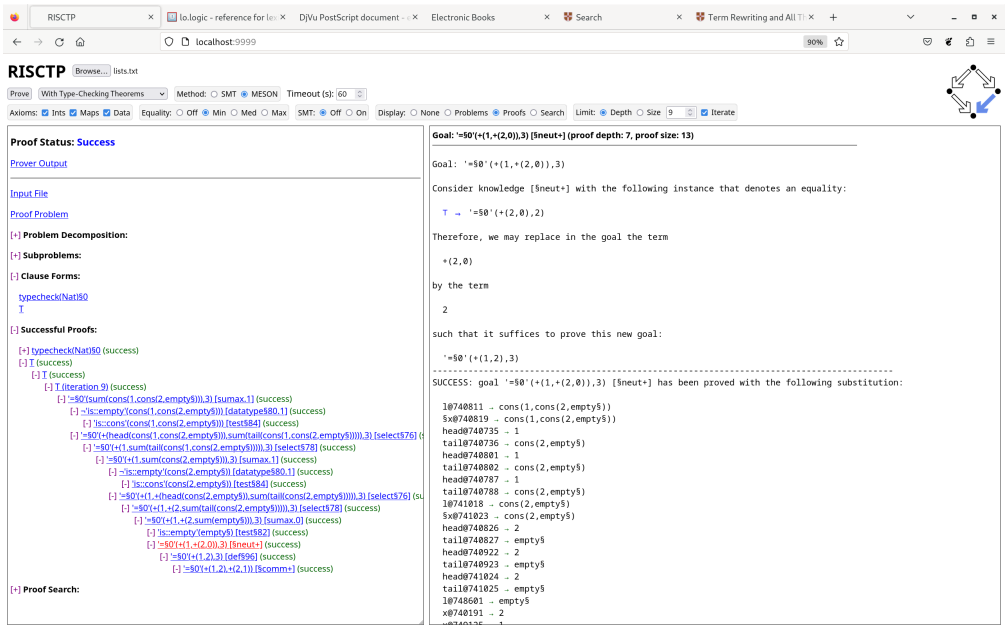

Figure 3: The Lists Proof with MESON

The option "Equality: Min" enables equational reasoning (in a minimal form, which is the most efficient one), essentially based on the proof principle of "paramodulation" [14], a form of equality reasoning based on "rewriting": here we consider rules of form $(\forall x.A_1 \wedge \ldots \wedge A_m \Longrightarrow B_1 \vee \ldots \vee B_n \vee t = u)$ that assert an equality $t = u$ of two terms t and u (if $m > 0$ or $n > 0$, this equality holds only under the given side conditions). Such a rule can be utilized for the proof of a goal literal $G(t')$ that includes a term t' which can be unified with one side of the equality, say t, by some variable substitution s: rather than proving $G(t')$ we can then prove $G(u')$ where u' is derived from the other side u of the equality by application of s; furthermore, we have to prove the auxiliary goals $A'_1, \ldots, A'_m, \neg B'_1, \ldots, \neg B'_n$ that are derived from the side conditions of the rule (if there are any), also by application of s.

With the option "Equality: Min", several restrictions apply to the use of equalities as rewrite rules: terms are only rewritten at those positions that appear in the original literal (no rewriting inside the terms with which the variables in the literal have been instantiated), rules are only applied in a direction that does not increase the term according to a variant of the "lexicographic path ordering" [1, 8], and rules are not applied to rewrite just variables. With option "Equality: Med", rules are also applied at all term positions, with option "Equality: Max" all restrictions are dropped: rewriting may increase the term and also rewriting into variables is allowed. Many practical proofs already succeed with "Equality: Min"; option "Equality: Max" may increase the proof search space extremely and should be only used when really required.

Finally, if also the option "SMT: On" is selected, in every proof situation elaborated by MESON, also an external SMT solver is applied in an attempt to close the current proof situation; in above example, however, this option is not selected: it is here not of any help but would just

(considerably) slow down the proof search. Please also note that with this option it is usually still necessary to apply the various "Axioms" options: only then the strategy of "backward chaining" applied by MESON is likely to generate proof situations that may be closed by SMT solving.

4 Conclusions

In earlier versions of RISCTP, mainly the proof method `smt` has been suitable for solving proof problems arising in the context of program verification; thus RISCTP can be seen as a convenient mechanism to access the functionality of SMT solvers such as Z3 or cvc5. This mechanism is available via the command-line interface and the web interface but also via the internal Java API of the software; in fact, using this API, RISCTP has been integrated into version 4.0 of the RISCAL model checker such that RISCAL proof problems can be translated to the RISCTP language and cracked via SMT solving.

However, in the current version of RISCTP also the proof method `meson` may become useful. With this method, which implements a MESON prover for first-order logic, the main goal is to make proof attempts as transparent as possible such that the human user can understand a proof and, in particular, also why a proof attempt has *failed*. This is in our opinion the main drawback of fully automated theorem proving systems that are essentially "black boxes": if a proof succeeds, everything is fine; however, if a proof fails, the underlying reasons remain unclear or are at least difficult to understand. On the contrary, the MESON prover implements a rather intuitive proof strategy that clearly illustrates the goal and structure of a proof; it also supports reasoning about equality (by applying a variant of paramodulation) and about the theories of arrays, algebraic data types, and integers (by adding corresponding axioms to the proof problems and/or by seeking the aid of an external SMT solver).

The focus of our current work is to further investigate the usefulness of RISCTP as a proof system on its own. For this purpose we will integrate the new version of RISCTP into the RISCAL software whose purpose is the specification and verification of algorithms. We hope to demonstrate that the proof problems arising from RISCAL are not only manageable by SMT solving but also by the reasoning capabilities of RISCTP itself.

References

[1] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK, 2012. doi:10.1017/CBO9781139172752.

[2] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In Dana Fisman and Grigore Rosu, editors, *TACAS 2022, Tools and Algorithms for the Construction and Analysis of Systems, 28th International Conference, Munich, Germany, April 2–7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, Cham, Switzerland, 2022. https://doi.org/10.1007/978-3-030-99524-9_24.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, The SMT-LIB Initiative, 2021. http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). http://SMT-LIB.org, 2022.

[5] Clark Barrett and Cesare Tinelli. cvc5 — An Efficient Open-Source Automatic Theorem Prover for Satisfiability Modulo Theories (SMT) Problems. Stanford University and University of Iowa, 2022. https://cvc5.github.io.

[6] Nikolaj Bjørner. Z3 Theorem Prover. Microsoft Research, 2022. https://github.com/Z3Prover.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS 2008, Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary, March 29 – April 6*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, Berlin, Germany, 2008. doi:10.1007/978-3-540-78800-3_24.

[8] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, Cambridge, UK, 2009. doi:10.1017/CBO9780511576430.

[9] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification, 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013, Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, Berlin, Germany, 2013. https://doi.org/10.1007/978-3-642-39799-8_1.

[10] Reinhold Letz and Gernot Stenz. Model Elimination and Connection Tableau Procedures. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 28, pages 2015–2114. North-Holland, Amsterdam, The Netherlands, 2001. doi:10.1016/B978-044450813-3/50030-8.

[11] Donald W. Loveland. Mechanical Theorem-Proving by Model Elimination. *Journal of the ACM*, 15(2):236—251, April 1968. doi:10.1145/321450.321456.

[12] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis*, volume 6 of *Fundamental Studies in Computer Science*. North-Holland, Amsterdam, The Netherlands, 1978. doi:10.1016/c2009-0-12705-8.

[13] Stefan Ratschan. ExSpec – Executable Specifications. Institute of Computer Science, The Czech Academy of Sciences, Prague, Czech Republic, September 2023. https://www.cs.cas.cz/~ratschan/exspec.

[14] G. Robinson and L. Wos. Paramodulation and Theorem-Proving in First-Order Theories with Equality. *Machine Intelligence*, 4:135–150, 1969. doi:10.1007/978-3-642-81955-1_19 (reprint).

[15] Wolfgang Schreiner. The RISC Algorithm Language (RISCAL) — Tutorial and Reference Manual (Version 1.0). Technical report, RISC, Johannes Kepler University, Linz, Austria, March 2017. Available at [17].

[16] Wolfgang Schreiner. Validating Mathematical Theories and Algorithms with RISCAL. In F. Rabe, W. Farmer, G. Passmore, and A. Youssef, editors, *CICM 2018, 11th Conference on Intelligent Computer Mathematics, Hagenberg, Austria, August 13–17*, volume 11006 of *Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence*, pages 248–254. Springer, Berlin, Germany, 2018. doi:10.1007/978-3-319-96812-4_21.

[17] Wolfgang Schreiner. The RISC Algorithm Language (RISCAL). Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2019. https://www.risc.jku.at/research/formal/software/RISCAL.

[18] Wolfgang Schreiner. Theorem and Algorithm Checking for Courses on Logic and Formal Methods. In Pedro Quaresma and Walther Neuper, editors, *Post-Proceedings ThEdu'18, Theorem proving components for Educational software, Oxford, United Kingdom, July 18, 2018*, volume 290 of *EPTCS*, pages 56–75, 2019. doi:10.4204/EPTCS.290.5.

[19] Wolfgang Schreiner, Alexander Brunhuemer, and Christoph Fürst. Teaching the Formalization of Mathematical Theories and Algorithms via the Automatic Checking of Finite Models. In Pedro Quaresma and Walther Neuper, editors, *Post-Proceedings ThEdu'17, Theorem proving components for Educatio nal software, Gothenburg, Sweden, August 6, 2017*, volume 267 of *EPTCS*, pages 120–139, 2018. doi:10.4204/EPTCS.267.8.

[20] Wolfgang Schreiner and Franz-Xaver Reichl. First-Order Logic in Finite Domains: Where Semantic Evaluation Competes with SMT Solving. In Temur Kutsia, editor, *SCSS 2021, 9th International Symposium on Symbolic Computation in Software Science, Hagenberg, Austria, September 8-10, 2021*, volume 342 of *EPTCS*, pages 99–113, 2021. doi:10.4204/EPTCS.342.9.

[21] Mark E. Stickel. A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler. *Journal of Automated Reasoning*, 4:353–380, 1988. doi:10.1007/BF00297245.

[22] Andrei Voronkov and Laura Kovács. Vampire. Univeristy of Manchester and TU Wien, 2022. https://vprover.github.io.

A The RISCTP Software

In the following sections, we describe the software that implements the RISCTP language.

A.1 Installing the Software

The README file of the installation is included below.

```
------------------------------------------------------------------------------
README
Information on RISCTP.

Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
Copyright (C) 2022-, Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria, https://www.risc.jku.at

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

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

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

The RISCTP Theorem Proving Interface
====================================
https://www.risc.jku.at/research/formal/software/RISCTP

The RISCTP theorem proving interface consists of a language for specifying proof
problems and of an associated software for solving these problems. The RISCTP
language is a typed variant of first-order logic whose level of abstraction is
between that of higher level formal specification languages (such as the
language of the RISCAL model checker) and lower level theorem proving languages
(such as the language SMT-LIB supported by various satisfiability modulo
theories solvers and theorem provers such as Z3, cvc5, or Vampire). Thus the
RISCTP language can serve as an intermediate layer that simplifies the
connection of specification and verification systems to theorem provers; in
fact, it was developed to equip the RISCAL model checker with theorem proving
capabilities. The RISCTP software is implemented in Java with an API that
enables the implementation of such connections; however, RISCTP also provides a
text-based frontend and a web-based GUI that allow its use as a theorem prover
on its own. RISCTP provides a backend that translates a proving problem into
SMT-LIB and solves it by the "black box" application of an external SMT solver.
Furthermore, RISCTP implements an internal prover for first-order logic with
equality and the possibility to inspect successful proofs as well as failed
proof attempts. This prover also supports reasoning in the theories of integers,
arrays, tuples, and algebraic datatypes, via a corresponding axiomatization,
```

```
and/or the application of an external SMT solver.

The Distribution
================
The distribution has the following contents:

  README     ... this file
  COPYING    ... the GNU General Public Licence Version 3
  CHANGES    ... the version history of the software
  etc/
    RISCTP   ... the execution script
    z3       ... Z3 (GNU Linux/x86-64 executable)
    cvc5     ... cvc5 (GNU Linux/x86-64 executable)
    vampire  ... Vampire (GNU Linux/x86-64 executable)
    exspec   ... ExSpec (GNU Linux/x86-54 exexutable)
  lib/
    *.jar    ... the Java-compiled libraries
  doc/
    main.pdf ... the manual
  problems/
    *.txt    ... sample problem files
  src/
    */*.java ... the Java source code

Installation
============
First make sure that you have installed the Java Development Kit (see below).

Then copy file etc/RISCTP to a directory in your PATH and adapt in this file
the variable JAVA to point to the Java executable "java". Adapt LIB to point
to the directory "lib" of the RISCTP distribution.

You should then be able to execute

  RISCTP -h

Third Party Software That You Have to Install
=============================================
RISCTP assumes that the following third party software is installed on your
computer (if it is not already provided by your GNU/Linux distribution, you have
to downlad and install it manually).

Java Development Kit (Oracle JDK 17 recommended)
http://www.oracle.com/technetwork/java/javase/downloads/index.html
-------------------------------------------------------------------
Go to the "Downloads" section to download the JDK.

Oracle JDK 17 is recommended. Other JDK versions will most probably work but
have not been tested.

Third Party Software That Comes with RISCTP
===========================================
RISCTP also uses the the following open source software developed by third
parties. This software is already included in the distribution, but if you want
```

```
or need, you can download the source code from the denoted locations and compile
it on your own. Many thanks to the respective developers for making this great
software freely available!

ANTLR 4.13.1
http://www.antlr.org
--------------------
This is a framework for constructing parsers and lexical analyzers used for
processing the RISCTP language.

On a Debian 11 "bullseye" GNU/Linux distribution, just install the package
"antlr4" by executing (as superuser) the command

  apt-get install antlr4

Z3 4.8.17: https://github.com/Z3Prover
-------------------------------------
To use the SMT mode "z3", Z3 has to be installed (above version has been
tested, other versions may or may not work).

The RISCTP distribution is bundled with an GNU Linux/x86-64 executable of
this solver.

cvc5 1.0.0: https://cvc5.github.io/
-----------------------------------
To use the SMT mode "cvc5", cvc5 has to be installed (above version has
been tested, other versions may or may not work).

The RISCTP distribution is bundled with an GNU Linux/x86-64 executable of
this solver.

Vampire 4.6.1: https://vprover.github.io/
-----------------------------------------
To use the SMT mode "vampire", Vampire has to be installed (above version
has been tested, other versions may or may not work).

The RISCTP distribution is bundled with an GNU Linux/x86-64 executable of
this solver.

ExSpec 0.3: https://www.cs.cas.cz/~ratschan/exspec/
--------------------------------------------------
To use the SMT mode "exspec", ExSpec has to be installed (above version
has been tested, other versions may or may not work). Please note that the
RISCTP support of this solver is still limited and considered experimental.

The RISCTP distribution is bundled with an GNU Linux/x86-64 executable of
this solver.

------------------------------------------------------------------------------
End of README.
------------------------------------------------------------------------------
```

A.2 Running the Software

The RISCAL software can be used in interactive mode by executing the shell script

```
RISCTP
```

which prints out the copyright message

```
RISC Theorem Proving Interface 1.7.1 (November 23, 2023)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
---------------------------------------------------------------
Reading standard input...
```

If we then press <CTRL>-D to close the standard input stream, this terminates the program. In general, RISCTP terminates with a negative exit code if an error has occurred and with a non-negative code, otherwise.

However, if we execute (as indicated in above message)

```
RISCTP -h
```

we get the following output:

```
RISCTP [ <options> ] [ <path> ]
<path>: path of problem file (if none, read from stdin)
<options>: the following command line options
-h: print this message and exit
-options: show further options
FAILURE termination.
```

The option `-h` just prints the help message and exits. However, if `RISCTP -options` is executed, then at last the full list of options is printed:

```
RISCTP [ <options> ] [ <path> ]
<path>: path of problem file (if none, read from stdin)
<options>: the following command line options
-h: print this message and exit
-options: show further options
<options> includes the following options:
-theorem T: prove theorem T (default: all theorems)
-method M: user proof method M (none, smt, meson, res; default: none)
-decompose: decompose problem before application of method fol
-mode M: use proof mode M (default 2)
   M=0: prove only type-checking theorems
   M=1: prove without type-checking theorems
   M=2: prove with type-checking theorems
```

```
-solver S: use solver S (z3, cvc5, vampire, exspec; default: z3)
-path P: set path to solver executable (default: "z3")
-timeout T: use timeout T ms (default: 15000)
-axints: use integer axioms
-axmaps: use map axioms
-axdata: use data type axioms
-mesmt: aid MESON by application of external SMT solver
-meeq E: set level of MESON goal literal rewriting
   E=0: none (default)
   E=1: minimal (only rewrite at outer term positions)
   E=2: medium (rewrite also at inner term positions)
   E=3: maximum (rewrite also at variables, do not order equalities)
-medepth B E I: loop parameters for MESON proof depth D: for (D=B; D<=E; D+=I)
-mesize B E I: loop parameters for MESON proof size S: for (S=B; S<=E; S+=I)
-q: quiet execution, minimize output
-web P B: use web GUI on port P with button control B
   B=0: without buttons (proof is automatically started)
   B=1: with buttons (proof has to be manually started)
   B=2: with buttons except 'browse' (problem cannot be changed)
-pall: switch on all of the following print options
-p: print problem after parsing
-pap: print problem after all processing
-ptc: print problem after type-checking
-pov: print problem after overloading resolution
-pch: print problem after choice removal
-pst: print problem after subtype removal
-pva: print problem after variable processing
-pfd: print problem after function definition processing
-pdt: print problem after datatype processing
-ple: print problem after let processing
-pfo: print problem after formula processing
-ppr: print problem after pruning the problem
-pat: print axioms and theorems
-psmt: print SMT-LIB translation of problem
-pdec V: print decomposition with verbosity V (0-3, default: 0)
-psub: print subproblems after decomposition
-pcla: print clause form of problems
-pproof: print proof (method 'meson' only)
-psearch: print proof search (method 'meson' only)
FAILURE termination (1 ms).
```

The interpretation of these options is as follows:

-theorem *T* indicates that only theorem *T* is to be proved (multiple instances of this option may be given). If no such option is given, all theorems in the problem file are proved.

-method *M* determines the method to be be used for proving the theorem. If method `none` is chosen, no method is selected and no proof is performed (this is the default). If `smt` is chosen, the problem is translated into the SMT-LIB language and an internal SMT solver is applied to solve the problem. If `meson` is chosen, the problem is translated into a problem of first-order logic and the internal MESON prover is applied (currently without support for reasoning about equality and special theories).

-decompose states that the input problem is to be decomposed into subproblems before proving (only when method `"fol"` is selected).

-mode *M* selects the proof mode. For $M = 0$, only the type-checking theorems generated from the declarations in the problem file are proved. For $M = 1$, only the declared theorems (but not the type-checking theorems) are proved. For $M = 2$, both the generated type-checking theorems and the declared theorems are proved (this is the default).

-solver *S* If proof method `smt` has been selected (see option `-method`), this option determines the SMT solver to be applied. Currently, the solvers `z3`, `cvc5`, `vampire`, and `exspec` may be selected. The use of `vampire` or `exspec`, however, is only advisable with a single theorem and mode $M = 1$, because these solvers can only solve a single query per problem. Furthermore, `exspec` should be only selected on problems that involve no other types than `Int` and `Map[Int,Int]` (and potentially functions on these types).

-path *P* If proof method `smt` has been selected (see option `-method`), this option determines the path to the executable of the SMT solver; the current default and only choice is `z3`.

-timeout *T* This option determines the number *T* of milliseconds, after which the attempt to prove a theorem may be aborted (the current default is `15000`).

-axints, -axmaps, -axdata If these options are selected, proof problems are extended by axioms for the builtin operations on integers, maps, and algebraic data types (including tuples), respectively. If some of this options are selected, one may also consider the option `-mesmt`.

-mesmt If the proof method `meson` is chosen, the system applies in every proof step the currently configured SMT solver (the solver option "exspec" is not applicable) to the current proof context, i.e., unquantified clauses, assumed literals, and the literal to be proved. While this option may help to close some proof situations, it may also considerably slow down the proof search and should be therefore applied with care. Also note that this options is most likely not useful without also applying (some of) the options "-axints", "-axmaps", "-axdata", since without the correspondingly added axioms the proof search strategy applied by MESON will most likely not expose proof situations that can be closed by SMT solving.

-meeq *E* This option determines the amount of rewriting of goal literals applied in MESON based on the paramodulation method: $E = 0$ switches off rewriting; $E = 1$ only rewrites at outer term positions, $E = 2$ also rewrites at inner term positions, $E =$ also rewrites variables and does not order equalities.

-medepth *B E I* If the proof method `meson` is chosen, the systems applies "iterative deepening" on the proof *depth*, with start value *B*, end value *E*, and increment *I* (if not explicitly specified, some default values are chosen).

-mesize *B E I* If the proof method `meson` is chosen, the system applies "iterative deepening" of the proof *size*, with start value *B*, end value *E*, and increment *I* (if not explicitly specified, iterative deepening of the proof *depth* is applied with some default values).

-q This option indicates that software shall run "quietly"; in particular, the copyright message shall not be printed.

-web *P B* This option indicates that the program shall be started in GUI mode, i.e., as a web server listening at port *P*; the GUI can then be accessed by a browser at URL `http://localhost:`*P*`/`. If $B = 0$, no buttons are shown in the GUI and the execution immediately starts with the command line arguments. If $B = 1$, buttons are shown that allow to select the problem file and various execution options. If $B = 2$, all buttons except the "browse" button are shown; thus the proof problem has to be specified via the "path" argument and cannot be changed.

-p* This set of options determines after which transformation stage(s) the current version of the proof problem shall be printed. Option `-p` lets the proof problem be printed after parsing (including all parentheses that make the interpretation of nested expressions unambiguous). Option `-pall` prints after every stage (maximal output). The other options print output after specific stages. If proof method `smt` is chosen (see option `-method`), option `-psmt` prints the SMT-LIB translation of the problem. If option `-decompose` is selected, option `-pdec` *V* prints the problem decomposition with verbosity *V* (0-3, default: 0); if option `-psub` is selected, the proof problems arising from the decomposition are printed. If option `-pcla` is selected, the clausal forms of the problems/subproblems are printed. If option `-pproof` is selected, every proof generated by method `-meson` is printed (but note that the creation of the proof tree slows down the proof search). If option `-psearch` is selected, a trace of the proof search of method `-meson` is printed (this slows down the proof search even more and also requires much more memory).

B The RISCTP Language

In the following sections, we describe the proof problem language.

B.1 Lexical Structure

On the lowest level, a RISCTP proof problem is a file encoded in UTF-8 format. RISCTP uses several Unicode characters that cannot be found on keyboards, but for each such character there exists an equivalent string in ASCII format that can be typed on a keyboard. While the RISCTP grammar supports both alternatives, the use of the Unicode characters yields much prettier specifications and is thus recommended. See Figure 4 for the ASCII strings and the corresponding Unicode symbols.

ASCII String	Unicode Character
`Int`	$\mathbb{Z}$
`Nat`	$\mathbb{N}$
`:=`	$:=$
`true`	$\top$
`false`	$\bot$
`~`	$\neg$
`/\`	$\wedge$
`\/`	$\vee$
`=>`	$\Rightarrow$
`<=>`	$\Leftrightarrow$
`forall`	$\forall$
`exists`	$\exists$
`~=`	$\neq$
`<=`	$\leq$
`>=`	$\geq$
`*`	$\cdot$
`<<`	$\langle$
`>>`	$\rangle$

Figure 4: ASCII Strings and their Equivalent Unicode Characters

A specification file may include two kinds of comments which are ignored when processing the file:

- Comments starting with `//` and ranging till the end of the file.
- Comments starting with `/*` and ending with `*/` (such comments must not be nested).

Likewise white space characters (blanks, tabulators, new lines, returns, form feeds) are ignored.

The syntactical grammar of RISCTP uses the following kinds of terminal symbols:

- An *identifier* ⟨*ident*⟩ may be either a "plain" identifier or a "quoted" identifier.
  - A plain identifier is a non-empty sequence of (lower and upper case) ASCII letters, decimal digits, and the underscore character `_` starting with a letter or underscore, e.g.`pos0` or `_0x`.
  - A quoted identifier is any non-empty sequence of characters enclosed by single quotes, e.g. `'8-?'`, which does not include any of the characters `'` (single quote), `\` (backslash), and `§` (paragraph) between the single quotes[1].
- A *decimal number literal* ⟨*decimal*⟩ is a non-empty sequence of decimal digits, e.g. `123`.

[1] A backslash is not allowed, because this character is also prohibited in the quoted identifiers of SMT-LIB. A paragraph is not allowed, because it used for identifiers internally generated by RISCTP.

B.2 Grammar

The RISCAL grammar (for both lexical analysis and syntax analysis) is formally defined below as an ANTLR4 grammar file. Please note that the order of options in a grammar rules determine precedences; options that come earlier have higher precedence than others.

```
// ------------------------------------------------------------------------
// RISCTP.g4
// RISC Theorem Proving Interface ANTLR4 Grammar
// $Id: RISCTP.g4,v 1.4 2023/11/23 16:42:54 schreine Exp $
//
// Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
// Copyright (C) 2022-, Research Institute for Symbolic Computation (RISC)
// Johannes Kepler University, Linz, Austria, https://www.risc.jku.at
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.
// ------------------------------------------------------------------------

grammar RISCTP;

options
{
  language=Java;
}

@header
{
  package risctp.parser;
}

// ------------------------------------------------------------------------
// problems and declarations
// ------------------------------------------------------------------------

// problems
problem: ( decl )* EOF ;

// declarations
decl:
  'type' id ( '=' type ( 'with' exp )? )? EOS              #TypeDecl
| 'datatype' dtitem ( 'and' dtitem)* EOS                   #DataType
| 'fun' id ( '(' ( tvar ( ',' tvar )* )? ')' )? ':' type
   ('=' exp )? EOS                                         #Function
```

```
| 'const' id ':' type ( '=' exp )? EOS                    #Constant
| 'pred' id ( '(' ( tvar ( ',' tvar )* )? ')' )?
   ( ( '⇔' | '<=>' ) exp )? EOS                            #Predicate
| 'axiom' id ( '⇔' | '<=>' ) exp EOS                      #Axiom
| 'theorem' id ( '⇔' | '<=>' ) exp EOS                    #Theorem
;

// ------------------------------------------------------------------------
// expressions and types
// ------------------------------------------------------------------------

// expressions
exp  :
  dec                                  #Decimal
| id ( '(' ( exp ( ',' exp )* )? ')' )? #Apply
| id ( '(' ( exp ( ',' exp )* )? ')' )? '{*}' #ApplyNotGoal

// maps, tuples, variants
| 'map' '[' pid ',' pid ']' '(' exp ')'     #MapConstruct
| ( '⟨' | '⟨' | '⟨' | '<<' ) exp ( ',' exp )* ( '⟩' | '⟩' | '⟩' | '>>' )
                                            #TupleConstruct
| exp '[' exp ']'                           #MapSelect
| exp '.' dec                               #TupleSelect
| exp 'with' '[' exp ']' '=' exp            #MapStore
| exp 'with' '.' dec '=' exp                #TupleStore

// arithmetic
| '-' exp                #Neg
| exp ( '*' | '·' ) exp  #Mult
| exp '/' exp            #Div
| exp '%' exp            #Mod
| exp '-' exp            #Minus
| exp '+' exp            #Plus

// infix relations
| exp '=' exp             #Equal
| exp ( '≠' | '~=' ) exp  #NotEqual
| exp '<' exp             #Less
| exp ( '≤' | '<=' ) exp  #LessEqual
| exp '>' exp             #Greater
| exp ( '≥' | '>=' ) exp  #GreaterEqual

// formulas
| ( '⊥' | 'false' )               #False
| ( '⊤' | 'true'  )               #True
| ( '¬' | '~' ) exp               #Not
| exp ( '∧' | '/\\' ) exp         #And
| exp ( '∨' | '\\/' ) exp         #Or
| exp ( '⇒' | '=>' ) exp          #Imp
| exp ( '⇔' | '<=>' ) exp         #Equiv
| ( '∀' | 'forall' ) tvar ( ',' tvar )* '.' exp  #Forall
| ( '∃' | 'exists' ) tvar ( ',' tvar )* '.' exp  #Exists

// generic terms
```

```
| 'if' exp 'then' exp 'else' exp            #IfThenElse
| 'match' exp 'with'
  ( '|' )? mbinder ( '|' mbinder )*         #Match
| 'let' lbinder ( ',' lbinder )* 'in' exp   #Let
| 'choose' tvar 'with' exp                  #Choose
| 'choose' '[' 'sat' ']' tvar 'with' exp    #ChooseSat

// parenthesized expressions
| '(' exp ')'  #Parentheses
;

// types
type : id ( '[' ( type ( ',' type )* )? ']' )? ;

// ----------------------------------------------------------------------------
// miscellaneous
// ----------------------------------------------------------------------------

// typed variables
tvar : id ':' type ;

// let binders
lbinder : id '=' exp ;

// match binders
mbinder : pattern '->' exp ;

// patterns
pattern :
  '_'                                       #DefaultPattern
| id ( '(' ( tvar ( ',' tvar )* )? ')' )?   #ConstrPattern
;

// datatype items
dtitem: id '=' dtconstr ( '|' dtconstr )* ( 'with' exp )? ;

// datatype constructors
dtconstr : id ( '(' ( tvar ( ',' tvar )* )? ')' )? ;

// identifiers (plain or quoted)
id:
  ID  #PlainId
| QID #QuotedId
;

// plain ids
pid: ID;

// decimal literals
dec: DEC ;

// ----------------------------------------------------------------------------
// lexical rules
// ----------------------------------------------------------------------------
```

```
// reserve \ for internal identifiers
ID  : [a-zA-Z_][a-zA-Z_0-9]* ;
QID : [']~['\\§]+['] ;
DEC : [0-9]+ ;
EOS : ';' ;

WHITESPACE  : [ \t\r\n\f]+ -> skip ;
LINECOMMENT : '//' .*? '\r'? ('\n' | EOF) -> skip ;
COMMENT     : '/*' .*? '*/' -> skip ;

// matches any other character
ERROR : . ;

// ------------------------------------------------------------------------
// end of file
// ------------------------------------------------------------------------
```

C Examples

In the following, we list the example proof problems used in this document.

C.1 An Array Problem

```
// problem file "arrays.txt"
const N:Nat; axiom posN ⇔ N > 0;
type Index = Nat with value < N;
type Value; type Elem = Tuple[Int,Value]; type Array = Map[Index,Elem];
fun key(e:Elem):Int = e.1;
pred sorted(a:Array,from:Index,to:Index) ⇔
  ∀i:Index,j:Index. from ≤ i ∧ i < j ∧ j ≤ to ⇒ key(a[i]) ≤ key(a[j]);
theorem T ⇔
  ∀a:Array,from:Index,to:Index,x:Int.
    from ≤ to ∧ sorted(a,from,to) ⇒
    // let i = (from+to)/2 in
    let i = choose i:Index with from ≤ i ∧ i ≤ to in
    key(a[i]) < x ⇒ ¬∃j:Index. from ≤ j ∧ j < i ∧ key(a[j]) = x;

> RISCTP -method smt arrays.txt
RISC Theorem Proving Interface 1.7.1 (November 23, 2023)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
------------------------------------------------------------------
Reading file /usr2/schreine/papers/RISCTP2022/problems/arrays.txt...
=== SMT solving
SMT solver: Z3 version 4.8.17 - 64 bit
Proving theorem 'typecheck(Nat)§0'...
SUCCESS: theorem was proved (24 ms).
Proving theorem 'typecheck(Index)§2'...
SUCCESS: theorem was proved (5 ms).
```

```
Proving theorem 'typecheck(T)§3'...
SUCCESS: theorem was proved (7 ms).
Proving theorem T...
SUCCESS: theorem was proved (12 ms).
===
SUCCESS termination.
```

C.2 A List Problem

```
// problem file "lists.txt"
datatype IntList = empty | cons(head:Int,tail:IntList);
fun sum(l:IntList):Int;
axiom sumax ⇔ ∀l:IntList.
  sum(l) =
    match l with
    | empty -> 0
    | cons(h:Int,t:IntList) -> h+sum(t);
theorem T ⇔
  let l = cons(1, cons(2, empty)) in sum(l) = 3;

> RISCTP -method smt lists.txt
RISC Theorem Proving Interface 1.7.1 (November 23, 2023)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
------------------------------------------------------------------
Reading file /usr2/schreine/papers/RISCTP2022/problems/lists.txt...
=== SMT solving
SMT solver: Z3 version 4.8.17 - 64 bit
Proving theorem 'typecheck(Nat)§0'...
SUCCESS: theorem was proved (26 ms).
Proving theorem T...
SUCCESS: theorem was proved (7 ms).
===
SUCCESS termination.
```

C.3 A First-Order Logic Problem

```
// problem file "fol.txt"
type T;
pred p(x:T);
pred q(x:T);
pred r(x:T);
fun f(x:T):T;
theorem Theorem ⇔
  ∀a:T,b:T.
    (p(a) ∨ q(b)) ∧
    (∀x:T. p(x) ⇒ r(x)) ∧
    (∀x:T. q(x) ⇒ r(f(x))) ⇒
      (∃x:T. p(x) ∨ q(x)) ∧ (∃x:T. r(x));

> RISCTP -method smt fol.txt
```

```
RISC Theorem Proving Interface 1.7.1 (November 23, 2023)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
---------------------------------------------------------------
Reading file /usr2/schreine/papers/RISCTP2022/fol.txt...
=== SMT solving
SMT solver: Z3 version 4.8.17 - 64 bit
Proving theorem Theorem...
SUCCESS: theorem was proved (107 ms).
===
SUCCESS termination.
```

C.4 A MESON Proof

```
// problem file "fol.txt"
type T;
pred p(x:T);
pred q(x:T);
pred r(x:T);
fun f(x:T):T;
theorem Theorem ⇔
  ∀a:T,b:T.
    (p(a) ∨ q(b)) ∧
    (∀x:T. p(x) ⇒ r(x)) ∧
    (∀x:T. q(x) ⇒ r(f(x))) ⇒
      (∃x:T. p(x) ∨ q(x)) ∧ (∃x:T. r(x));
```

```
> RISCTP -method meson -pat -decompose -pdec 1 -pproof fol.txt
RISC Theorem Proving Interface 1.5.0 (June 12, 2023)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
---------------------------------------------------------------
Reading file /usr2/schreine/repositories/RISCTP/trunk/problems/fol.txt...
=== axioms and theorems:
theorem Theorem ⇔ ∀a:T,b:T. ((((p(a) ∨ q(b)) ∧ (∀x:T. (p(x) ⇒ r(x)))) ∧ (∀x:T. (q(x) ⇒ r(f(x))))) ⇒ ((∃
===
=== decomposition of proof problem into subproblems
=== Theorem (rule [∀-R | ∃-L] applied to the goal creates 1 subproblem)
  === Theorem.1 (rule [∀-R | ∃-L] applied to the goal creates 1 subproblem)
    === Theorem.1.1 (rule [⇒-R | ∨-R | ∧-L] applied to the goal creates 1 subproblem)
      === Theorem.1.1.1 (rule [∧-R | ∨-L | ⇒-L] applied to the goal creates 2 subproblems)
        === Theorem.1.1.1.1 (rule [⇒-R | ∨-R | ∧-L] applied to formula [1] creates 1 subproblem)
          === Theorem.1.1.1.1.1 (rule [⇒-R | ∨-R | ∧-L] applied to formula [1] creates 1 subproblem)
            === Theorem.1.1.1.1.1.1 (rule [∧-R | ∨-L | ⇒-L] applied to formula [1] creates 2 subproblems)
              === Theorem.1.1.1.1.1.1.1 (open)
              === Theorem.1.1.1.1.1.1.2 (open)
        === Theorem.1.1.1.2 (rule [⇒-R | ∨-R | ∧-L] applied to formula [1] creates 1 subproblem)
          === Theorem.1.1.1.2.1 (rule [⇒-R | ∨-R | ∧-L] applied to formula [1] creates 1 subproblem)
            === Theorem.1.1.1.2.1.1 (rule [∧-R | ∨-L | ⇒-L] applied to formula [1] creates 2 subproblems)
              === Theorem.1.1.1.2.1.1.1 (open)
```

```
              === Theorem.1.1.1.2.1.1.2 (open)
decomposition created 4 subproblems.
=== proof method 'meson': model elimination (subgoal-oriented)
Goal: ∀x:T. q(x) ⇒ ⊥
Iteration 1 (proof depth 1)...
Iteration 2 (proof depth 2)...
Iteration 3 (proof depth 3)...
Iteration 4 (proof depth 4)...
Iteration 5 (proof depth 5)...
Iteration 6 (proof depth 6)...
Iteration 7 (proof depth 7)...
Iteration 8 (proof depth 8)...
Iteration 9 (proof depth 9)...
Iteration 10 (proof depth 10)...
Goal: ∀x:T. p(x) ⇒ ⊥
Iteration 1 (proof depth 1)...
SUCCESS: the proof problem has been solved (1 clause applications).
===
Proof:
[] Theorem.1.1.1.1.1.1.1
  [1] Theorem.0.0.2.1.0
    [1.1] Theorem.0.0.2.1.0 (iteration 1)
      [1.1.1] p(x) [Theorem.0.0.1.1.1.1]
------------------------------------------------------------------------
Proof [Theorem.1.1.1.1.1.1.1]

The following "negated goals" represent the negation of the theorem to be proved:

[Theorem.0.0.1.1.1.1] ⊤ ⇒ p(a0)
[Theorem.0.0.1.1.2] ∀x:T. p(x) ⇒ r(x)
[Theorem.0.0.1.2] ∀x:T. q(x) ⇒ r(f(x))
[Theorem.0.0.2.1.0] ∀x:T. p(x) ⇒ ⊥
[Theorem.0.0.2.1.1] ∀x:T. q(x) ⇒ ⊥

To prove the theorem, we derive from the negated goals a contradiction. For this,
we prove some (not negated) goal from the "knowledge" represented by the other formulas.
We start the proof with the last goal; if this does not succeed, we also try the other ones.
------------------------------------------------------------------------------------------
SUCCESS: the proof has been completed.
------------------------------------------------------------------------
Goal: Theorem.0.0.2.1.0

Formula: ∃x:T. p(x)

Our goal is to prove this formula.
------------------------------------------------------------------------------------------
SUCCESS: goal Theorem.0.0.2.1.0 has been proved with the following substitution:

  x → a0
------------------------------------------------------------------------
Goal: Theorem.0.0.2.1.0 (iteration 1) (proof depth: 0, proof size: 0)

Formula: p(x)
Variables: x:T
```

```
To prove the goal formula, we determine variable values that satisfy each subformula:

  p(x)
-------------------------------------------------------------------------------------------
SUCCESS: goal Theorem.0.0.2.1.0 (iteration 1) has been proved with the following substitution:

  x → a0
-----------------------------------------------------------------------
Goal: p(x) [Theorem.0.0.1.1.1.1] (proof depth: 0, proof size: 1)

Formula: p(x)
Variables: x:T

To prove the goal formula, we assume its negation

[1] ¬p(x)

and show a contradiction. For this, consider knowledge [Theorem.0.0.1.1.1.1] with the following instance:

  ⊤ ⇒ p(a0)

Assumption [1] matches the literal p(a0) on the right side of this clause by the following substitution:

  x → a0

Therefore, applying this substitution and dropping the literal, we know:

  ⊤ ⇒ ⊥

Therefore we have a contradiction.
-----------------------------------------------------------------------------------------------
SUCCESS: goal p(x) [Theorem.0.0.1.1.1.1] has been proved with the following substitution:

  x → a0
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-------------------------------------------------------------
=== proof method 'meson': model elimination (subgoal-oriented)
Goal: ∀x:T. q(x) ⇒ ⊥
Iteration 1 (proof depth 1)...
SUCCESS: the proof problem has been solved (1 clause applications).
===
Proof:
[] Theorem.1.1.1.1.1.1.2
  [1] Theorem.0.0.2.1.1
    [1.1] Theorem.0.0.2.1.1 (iteration 1)
      [1.1.1] q(x) [Theorem.0.0.1.1.1.2]
-----------------------------------------------------------------------
Proof [Theorem.1.1.1.1.1.1.2]

The following "negated goals" represent the negation of the theorem to be proved:
```

[Theorem.0.0.1.1.1.2] ⊤ ⇒ q(b0)
[Theorem.0.0.1.1.2] ∀x:T. p(x) ⇒ r(x)
[Theorem.0.0.1.2] ∀x:T. q(x) ⇒ r(f(x))
[Theorem.0.0.2.1.0] ∀x:T. p(x) ⇒ ⊥
[Theorem.0.0.2.1.1] ∀x:T. q(x) ⇒ ⊥

To prove the theorem, we derive from the negated goals a contradiction. For this,
we prove some (not negated) goal from the "knowledge" represented by the other formulas.
We start the proof with the last goal; if this does not succeed, we also try the other ones.

SUCCESS: the proof has been completed.

Goal: Theorem.0.0.2.1.1

Formula: ∃x:T. q(x)

Our goal is to prove this formula.

SUCCESS: goal Theorem.0.0.2.1.1 has been proved with the following substitution:

  x → b0

Goal: Theorem.0.0.2.1.1 (iteration 1) (proof depth: 0, proof size: 0)

Formula: q(x)
Variables: x:T

To prove the goal formula, we determine variable values that satisfy each subformula:

  q(x)

SUCCESS: goal Theorem.0.0.2.1.1 (iteration 1) has been proved with the following substitution:

  x → b0

Goal: q(x) [Theorem.0.0.1.1.1.2] (proof depth: 0, proof size: 1)

Formula: q(x)
Variables: x:T

To prove the goal formula, we assume its negation

[1] ¬q(x)

and show a contradiction. For this, consider knowledge [Theorem.0.0.1.1.1.2] with the following instance:

  ⊤ ⇒ q(b0)

Assumption [1] matches the literal q(b0) on the right side of this clause by the following substitution:

  x → b0

Therefore, applying this substitution and dropping the literal, we know:

```
  ⊤ ⇒ ⊥

Therefore we have a contradiction.
--------------------------------------------------------------------------------------------
SUCCESS: goal q(x) [Theorem.0.0.1.1.1.2] has been proved with the following substitution:

  x → b0
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------
=== proof method 'meson': model elimination (subgoal-oriented)
Goal: ∀x:T. r(x) ⇒ ⊥
Iteration 1 (proof depth 1)...
SUCCESS: the proof problem has been solved (3 clause applications).
===
Proof:
[] Theorem.1.1.1.2.1.1.1
  [1] Theorem.0.0.2.2
    [1.1] Theorem.0.0.2.2 (iteration 1)
      [1.1.1] r(x) [Theorem.0.0.1.1.2]
        [1.1.1.1] p(x@2) [Theorem.0.0.1.1.1.1]
------------------------------------------------------------------------
Proof [Theorem.1.1.1.2.1.1.1]

The following "negated goals" represent the negation of the theorem to be proved:

[Theorem.0.0.1.1.1.1] ⊤ ⇒ p(a0)
[Theorem.0.0.1.1.2] ∀x:T. p(x) ⇒ r(x)
[Theorem.0.0.1.2] ∀x:T. q(x) ⇒ r(f(x))
[Theorem.0.0.2.2] ∀x:T. r(x) ⇒ ⊥

To prove the theorem, we derive from the negated goals a contradiction. For this,
we prove some (not negated) goal from the "knowledge" represented by the other formulas.
We start the proof with the last goal; if this does not succeed, we also try the other ones.
--------------------------------------------------------------------------------------------
SUCCESS: the proof has been completed.
------------------------------------------------------------------
Goal: Theorem.0.0.2.2

Formula: ∃x:T. r(x)

Our goal is to prove this formula.
--------------------------------------------------------------------------------------------
SUCCESS: goal Theorem.0.0.2.2 has been proved with the following substitution:

  x → a0
  x@2 → a0
------------------------------------------------------------------
Goal: Theorem.0.0.2.2 (iteration 1) (proof depth: 0, proof size: 0)

Formula: r(x)
```

```
Variables: x:T

To prove the goal formula, we determine variable values that satisfy each subformula:

  r(x)
-------------------------------------------------------------------------------------------
SUCCESS: goal Theorem.0.0.2.2 (iteration 1) has been proved with the following substitution:

  x → a0
  x@2 → a0
-----------------------------------------------------------------------
Goal: r(x) [Theorem.0.0.1.1.2] (proof depth: 0, proof size: 1)

Formula: r(x)
Variables: x:T

To prove the goal formula, we assume its negation

[1] ¬r(x)

and show a contradiction. For this, consider knowledge [Theorem.0.0.1.1.2] with the following instance:

  ∀x@2:T. p(x@2) ⇒ r(x@2)

Assumption [1] matches the literal r(x@2) on the right side of this clause by the following substitution:

  x → x@2

Therefore, applying this substitution and dropping the literal, we know:

  ∀x@2:T. p(x@2) ⇒ ⊥

Therefore, to show a contradiction, we determine variable values that satisfy this subgoal:

  p(x@2)
-------------------------------------------------------------------------------------------
SUCCESS: goal r(x) [Theorem.0.0.1.1.2] has been proved with the following substitution:

  x → a0
  x@2 → a0
-----------------------------------------------------------------------
Goal: p(x@2) [Theorem.0.0.1.1.1.1] (proof depth: 1, proof size: 2)

Formula: p(x@2)

Assumptions:

[1] ¬r(x@2)

Variables: x@2:T

To prove the goal formula, we assume its negation

[2] ¬p(x@2)
```

```
and show a contradiction. For this, consider knowledge [Theorem.0.0.1.1.1.1] with the following instance:

  ⊤ ⇒ p(a0)

Assumption [2] matches the literal p(a0) on the right side of this clause by the following substitution:

  x@2 → a0

Therefore, applying this substitution and dropping the literal, we know:

  ⊤ ⇒ ⊥

Therefore we have a contradiction.
------------------------------------------------------------------------------------------
SUCCESS: goal p(x@2) [Theorem.0.0.1.1.1.1] has been proved with the following substitution:

  x → a0
  x@2 → a0
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------
=== proof method 'meson': model elimination (subgoal-oriented)
Goal: ∀x:T. r(x) ⇒ ⊥
Iteration 1 (proof depth 1)...
SUCCESS: the proof problem has been solved (2 clause applications).
===
Proof:
[] Theorem.1.1.1.2.1.1.2
  [1] Theorem.0.0.2.2
    [1.1] Theorem.0.0.2.2 (iteration 1)
      [1.1.1] r(x) [Theorem.0.0.1.2]
        [1.1.1.1] q(x@1) [Theorem.0.0.1.1.1.2]
------------------------------------------------------------------------
Proof [Theorem.1.1.1.2.1.1.2]

The following "negated goals" represent the negation of the theorem to be proved:

[Theorem.0.0.1.1.1.2] ⊤ ⇒ q(b0)
[Theorem.0.0.1.1.2] ∀x:T. p(x) ⇒ r(x)
[Theorem.0.0.1.2] ∀x:T. q(x) ⇒ r(f(x))
[Theorem.0.0.2.2] ∀x:T. r(x) ⇒ ⊥

To prove the theorem, we derive from the negated goals a contradiction. For this,
we prove some (not negated) goal from the "knowledge" represented by the other formulas.
We start the proof with the last goal; if this does not succeed, we also try the other ones.
------------------------------------------------------------------------------------------
SUCCESS: the proof has been completed.
------------------------------------------------------------------------
Goal: Theorem.0.0.2.2
```

```
Formula: ∃x:T. r(x)

Our goal is to prove this formula.
-------------------------------------------------------------------------------------------
SUCCESS: goal Theorem.0.0.2.2 has been proved with the following substitution:

  x → f(b0)
  x@1 → b0
-----------------------------------------------------------------------
Goal: Theorem.0.0.2.2 (iteration 1) (proof depth: 0, proof size: 0)

Formula: r(x)
Variables: x:T

To prove the goal formula, we determine variable values that satisfy each subformula:

  r(x)
-------------------------------------------------------------------------------------------
SUCCESS: goal Theorem.0.0.2.2 (iteration 1) has been proved with the following substitution:

  x → f(b0)
  x@1 → b0
-----------------------------------------------------------------------
Goal: r(x) [Theorem.0.0.1.2] (proof depth: 0, proof size: 1)

Formula: r(x)
Variables: x:T

To prove the goal formula, we assume its negation

[1] ¬r(x)

and show a contradiction. For this, consider knowledge [Theorem.0.0.1.2] with the following instance:

  ∀x@1:T. q(x@1) ⇒ r(f(x@1))

Assumption [1] matches the literal r(f(x@1)) on the right side of this clause by the following substitution:

  x → f(x@1)

Therefore, applying this substitution and dropping the literal, we know:

  ∀x@1:T. q(x@1) ⇒ ⊥

Therefore, to show a contradiction, we determine variable values that satisfy this subgoal:

  q(x@1)
-------------------------------------------------------------------------------------------
SUCCESS: goal r(x) [Theorem.0.0.1.2] has been proved with the following substitution:

  x → f(b0)
  x@1 → b0
-----------------------------------------------------------------------
Goal: q(x@1) [Theorem.0.0.1.1.1.2] (proof depth: 1, proof size: 2)
```

```
Formula: q(x@1)

Assumptions:

[1] ¬r(f(x@1))

Variables: x@1:T

To prove the goal formula, we assume its negation

[2] ¬q(x@1)

and show a contradiction. For this, consider knowledge [Theorem.0.0.1.1.1.2] with the following instance:

  ⊤ ⇒ q(b0)

Assumption [2] matches the literal q(b0) on the right side of this clause by the following substitution:

  x@1 → b0

Therefore, applying this substitution and dropping the literal, we know:

  ⊤ ⇒ ⊥

Therefore we have a contradiction.
--------------------------------------------------------------------------------------------
SUCCESS: goal q(x@1) [Theorem.0.0.1.1.1.2] has been proved with the following substitution:

  x → f(b0)
  x@1 → b0
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------
SUCCESS termination (36 ms).
```