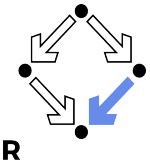
RISC



RESEARCH INSTITUTE FOR SYMBOLIC COMPUTATION

JKU

JOHANNES KEPLER UNIVERSITY LINZ

The RISCTP Theorem Proving Interface - Tutorial and Reference Manual (Version 1.0.*)

Wolfgang Schreiner

June 2022

RISC Report Series No. 22-07

ISSN: 2791-4267 (online)

Available at https://doi.org/10.35011/risc.22-07

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

The RISCTP Theorem Proving Interface

Tutorial and Reference Manual (Version 1.0.*)

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

June 8, 2022

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 such as Z3). 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 that allows its use as a theorem prover on its own. RISCTP already implements a backend that translates a proving problem into SMT-LIB and solves it by the "black box" application of Z3; in the future, RISCTP shall also provide built-in proving capabilities with greater transparency.

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 . . . 14
- 3.3 Proving with SMT . . . 16
- 3.4 Proving in First-Order Logic . . . 18

4 Conclusions **19**

A The RISCTP Software **21**
- A.1 Installing the Software . . . 21
- A.2 Running the Software . . . 23

B The RISCTP Language **25**
- B.1 Lexical Structure . . . 25
- B.2 Grammar . . . 26

C Examples **30**
- C.1 An Array Problem . . . 30
- C.2 A List Problem . . . 30

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 [7, 5, 9, 6, 8, 10], 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 envision the use of an external SMT (satisfiability modulo theories) solver such as Z3 [3, 4] as well the development of an internal first-order logic prover that interacts with an SMT solver.

Since most SMT solvers support the language of the SMT-LIB standard [2, 1], 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 two backends:

- An SMT backend that, via translation to SMT-LIB, solves proofs problems by applying Z3.
- A first-order logic backend which translates proof problems into classical first-order logic.

The first-order backend is, however, currently just a stub that still requires integration with or implementation of a corresponding prover.

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 1.0 (June 8, 2022)
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 $\neg F$ ("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. 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 choose expression the system generates a theorem that claims that (considering the context in which the expression occurs) for all possible values of the free variables of the choosee expression there exists a value that satisfies the formula.

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 `fol` (first-order logic) 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

The RISCTP implementation of the proof method `smt` (satisfiability modulo theories) is based on the SMT-LIB standard [2]. Specifically it requires an SMT solver that understands the language of SMT-LIB version 2.6 [1] (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 [3, 4]. Currently RISCTP uses Z3 version 4.8.17.

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

```
> RISCTP -method smt arrays.txt
RISC Theorem Proving Interface 1.0 (June 8, 2022)
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 (26 ms).
Proving theorem 'typecheck(Index)§2'...
SUCCESS: theorem was proved (2 ms).
Proving theorem 'typecheck(T)§7'...
SUCCESS: theorem was proved (6 ms).
Proving theorem T...
SUCCESS: theorem was proved (5 ms).
===
SUCCESS termination.
```

Thus Z3 could prove al 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 (> 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§86| ( Int  Int ) Int)
(assert (forall ((from Int)) (forall ((to Int))
  (let ( (i (|choose§86| 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§86| 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.
```

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 values are explicitly specified.

3.4 Proving in First-Order Logic

The proof method fol (first-order logic) requires a prover that understands the language of classical first-order logic with equality and integer arithmetic. Currently, the method is only a stub; it only generates a proof problem in this language without attempting to prove it.

When invoking the software with options `-method fol` and `-pat` (print axioms and theorems), we get the following output that shows the core of the generated proof problem:

```
> RISCTP -method fol -pat arrays.txt
RISC Theorem Proving Interface 1.0 (June 8, 2022)
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...
=== axioms and theorems:
axiom 'def§24' ⇔ ∀value:Int. ('Nat::type'(value) ⇔ (value ≥ 0));
theorem 'typecheck(Nat)§0' ⇔ ∃value:Int. 'Nat::type'(value);
axiom posN ⇔ N > 0;
axiom 'def§40' ⇔ ∀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'⟩) ⇒
    ('=§0'('.::1§1','.::1§2') ∧ '=§1'('.::2§1','.::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'(m1,m2));
axiom 'def§86' ⇔ ∀value:Map[Int,Tuple[Int,Value]].
  ('Map[Index,Elem]::type'(value) ⇔ (∀x:Int. ((¬'Index::type'(x)) ⇒
    '=§2'(value[x],'Tuple[Int,Value]§undef'))));
axiom 'ext§68' ⇔ ∀m1:Map[Index,Elem]. ('Map[Index,Elem]::type'(m1) ⇒
  (∀m2:Map[Index,Elem]. ('Map[Index,Elem]::type'(m2) ⇒
    ((∀k:Index. ('Index::type'(k) ⇒ '=§2'('[]§1'(m1,k),'[]§1'(m2,k)))) ⇒ '=§3'(m1,m2)))));
axiom 'def§101' ⇔ ∀value:Map[Int,Tuple[Int,Value]].
  ('Array::type'(value) ⇔ 'Map[Index,Elem]::type'(value));
axiom 'def§103' ⇔ ∀e:Elem. '=§0'(key(e),e.1);
axiom 'def§105' ⇔ ∀a:Array,from:Index,to:Index. (sorted(a,from,to) ⇔
    (∀i:Index. ('Index::type'(i) ⇒ (∀j:Index. ('Index::type'(j) ⇒
      ((((from ≤ i) ∧ (i < j)) ∧ (j ≤ to)) ⇒ (key(a[i]) ≤ key(a[j])))))))));
theorem 'typecheck(T)§7' ⇔ ∀a:Array. ('Array::type'(a) ⇒
  (∀from:Index. ('Index::type'(from) ⇒ (∀to:Index. ('Index::type'(to) ⇒
    (∀x:Int. (((from ≤ to) ∧ sorted(a,from,to)) ⇒
      (∃i:Index. ('Index::type'(i) ∧ ((from ≤ i) ∧ (i ≤ to))))))))));
axiom 'choose§87' ⇔ ∀from:Index,to:Index. ('Index::type'('choose§86'(from,to)) ∧
  ((from ≤ 'choose§86'(from,to)) ∧ ('choose§86'(from,to) ≤ to)));
theorem T ⇔ ∀a:Array. ('Array::type'(a) ⇒
  (∀from:Index. ('Index::type'(from) ⇒ (∀to:Index. ('Index::type'(to) ⇒
    (∀x:Int. (((from ≤ to) ∧ sorted(a,from,to)) ⇒
      ((key(a['choose§86'(from,to)]) < x) ⇒
        (¬(∃j:Index. ('Index::type'(j) ∧ (((from ≤ j) ∧
          (j < 'choose§86'(from,to))) ∧ '=§0'(key(a[j]),x))))))))))));
===
=== proof method 'fol' not yet implemented
FAILURE termination.
```

In the future, we plan to extend this step to an actual prover that combines internal methods for reasoning in first-order logic (possibly including equality) with an SMT solver for reasoning about equality and integer arithmetic.

4 Conclusions

Currently, only the proof method `smt` is well supported; RISCTP can be thus seen as just a convenient mechanism to access the functionality of an SMT solver such as Z3. This mechanism is availablle via the command-line interface but also via the internal Java API of the software: our next steps will be to integrate RISCTP into the RISCAL model checker, to translate RISCAL proof problems to RISCTP, and to crack them via SMT solving.

In the future, however, we also plan to elaborate the proof method `fol` by implementing a prover for first-order logic; here our main goal will be to make proof attempts as transparent as possible such that the human user understands why a proof has *failed*. This is in our opinion the main drawback of fully automated theorem proving systems such as Z3 that are essentially "black boxes": if a proof succeeds, everything is fine; however, if a proof fails, the underlying reason remains unclear. Our plan is here to implement a "human-understandable" proof strategy that clearly illustrates the overall goal and structure of a proof; this prover, however, will be combined with an SMT solver, in particular for lower level reasoning about equality and integer arithmetic. If successful, RISCTP may thus become interesting as a proof system on its own.

References

[1] 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.

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

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

[4] 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, Berlin, Germany, 2008. Springer. doi:10.1007/978-3-540-78800-3_24.

[5] 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 [7].

[6] 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, 2018. doi:10.1007/978-3-319-96812-4_21.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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
satisfability modulo theories solvers such as Z3). 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 that allows its interactive use as a theorem
prover on its own. RISCTP already implements a backend that translates a
proving problem into SMT-LIB and solves it by the "black box" application of
Z3; in the future, RISCTP shall also provide built-in proving capabilities
with greater transparency.

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)
  lib/
    *.jar    ... the Java-compiled libraries
  doc/
    main.pdf ... the manual
  spec/
    *.txt    ... sample specifications
  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 11 recommended)
http://www.oracle.com/technetwork/java/javase/downloads/index.html
-------------------------------------------------------------------
Go to the "Downloads" section to download the JDK.

Oracle JDK 11 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 4.9.2
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 has to be installed (above versions 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.

------------------------------------------------------------------------------
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 1.0 (June 8, 2022)
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, fol; default: none)
-mode M: use proof mode M (default 2)
   M=0: prove only type-checking theorems
   M=1: prove no type-checking theorems
   M=2: prove both kinds of theorems
-solver S: use solver S (default: <Z3>, currently only choice)
-path P: set path to solver executable (default: <z3>)
-timeout T: use timeout T ms (default: 15000)
-q: do not print copyright message
-pall: switch on all of the following print options
-p: print problem after parsing
-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
FAILURE termination.
```

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 SMT solver is applied to solve the problem. If `fol` is chosen, the problem is translated into a problem of first-order logic (however, currently no corresponding prover is provided by RISCTP yet).

-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, only solver `Z3` may be selected.

-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`).

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

-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.

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 1 for the ASCII strings and the corresponding Unicode symbols.

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.

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 1: ASCII Strings and their Equivalent Unicode Characters

  - 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`.

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 1.2 2022/06/02 14:29:52 schreine Exp $
//
```

[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.

```
// 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

// 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

// 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  #PId
| QID #QId
;

// 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 1.0 (June 8, 2022)
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/arrays.txt...
=== SMT solving
SMT solver: Z3 version 4.8.17 - 64 bit
Proving theorem 'typecheck(Nat)§0'...
SUCCESS: theorem was proved (38 ms).
Proving theorem 'typecheck(Index)§2'...
SUCCESS: theorem was proved (3 ms).
Proving theorem 'typecheck(T)§7'...
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 1.0 (June 8, 2022)
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 (36 ms).
Proving theorem T...
SUCCESS: theorem was proved (7 ms).
===
SUCCESS termination.
```