Specification of Zinc and MiniZinc

Zinc version 0.11
MiniZinc version 1.1

Nicholas Nethercote Kim Marriott Reza Rafeh Mark Wallace
María García de la Banda

Contents

1 Introduction **3**

2 Overview of a Model **3**
2.1 Specifying a Problem . . . 3
2.2 Evaluation of a Model Instance . . . 3
2.2.1 Evaluation Phases . . . 3
2.2.2 Evaluation Outcomes . . . 4

3 Syntax Overview **4**
3.1 Character Set . . . 4
3.2 Comments . . . 4
3.3 Syntax Notation . . . 4
3.4 Identifiers . . . 5

4 High-level Model Structure **5**
4.1 Items . . . 5
4.2 Model Instance Files . . . 5
4.3 Namespaces . . . 6
4.4 Scopes . . . 6

5 Types and Type-insts **6**
5.1 Properties of Types . . . 7
5.2 Instantiations . . . 7
5.3 Type-insts . . . 7
5.4 Type-inst Expressions Overview . . . 7
5.5 Built-in Scalar Types and Type-insts . . . 8
5.5.1 Booleans . . . 8
5.5.2 Integers . . . 8
5.5.3 Floats . . . 9
5.5.4 Strings . . . 9
5.6 Built-in Compound Types and Type-insts . . . 9
5.6.1 Sets . . . 9
5.6.2 Arrays . . . 10
5.6.3 Tuples . . . 11
5.6.4 Records . . . 11
5.7 User-defined Types and Type-insts . . . 12
5.7.1 Type-inst Synonyms . . . 12
5.7.2 Enumerated Types . . . 12
5.7.3 The Annotation Type `ann` . . . 13
5.8 Other Types and Type-insts . . . 13
5.8.1 Type-inst Variables . . . 13
5.8.2 Higher-order Types . . . 13
5.9 Constrained Type-insts . . . 14
5.9.1 Set Expression Type-insts . . . 14
5.9.2 Float Range Type-insts . . . 14
5.9.3 Arbitrarily Constrained Type-insts . . . 14

6 Expressions **15**
6.1 Expressions Overview . . . 15
6.2 Operators . . . 16
6.3 Expression Atoms . . . 17
6.3.1 Identifier Expressions and Quoted Operator Expressions . . . 17
6.3.2 Anonymous Decision Variables . . . 18
6.3.3 Boolean Literals . . . 18
6.3.4 Integer and Float Literals . . . 18
6.3.5 String Literals . . . 18
6.3.6 Set Literals . . . 18
6.3.7 Set Comprehensions . . . 19
6.3.8 Simple Array Literals . . . 19
6.3.9 Simple 2d Array Literals . . . 20
6.3.10 Indexed Array Literals . . . 20
6.3.11 Simple Array Comprehensions . . . 20
6.3.12 Indexed Array Comprehensions . . . 20

6.3.13 Array Access Expressions . . . 21
6.3.14 Tuple Literals . . . 21
6.3.15 Tuple Access Expressions . . . 21
6.3.16 Record Literals . . . 21
6.3.17 Record Access Expressions . . . 21
6.3.18 Enum Literals . . . 22
6.3.19 Non-flat Enum Access Expressions . . . 22
6.3.20 Annotation Literals . . . 22
6.3.21 If-then-else Expressions . . . 22
6.3.22 Case Expressions . . . 22
6.3.23 Let Expressions . . . 23
6.3.24 Call Expressions . . . 23
6.3.25 Generator Call Expressions . . . 24

7 Items **24**
7.1 Type-inst Synonym Items . . . 24
7.2 Enum Items . . . 24
7.3 Include Items . . . 25
7.4 Variable Declaration Items . . . 25
7.5 Assignment Items . . . 25
7.6 Constraint Items . . . 26
7.7 Solve Items . . . 26
7.8 Output Items . . . 26
7.9 Annotation Items . . . 26
7.10 User-defined Operations . . . 27
7.10.1 Basic Properties . . . 27
7.10.2 Ad-hoc polymorphism . . . 27
7.10.3 Parametric Polymorphism . . . 28
7.10.4 Local Variables . . . 29

8 Annotations **29**

9 Partiality **30**
9.1 Partial Assignments . . . 30
9.2 Partial Predicate/Function and Annotation Arguments . . . 31
9.3 Partial Array Accesses . . . 31

A Built-in Operations **32**
A.1 Comparison Operations . . . 32
A.2 Arithmetic Operations . . . 32
A.3 Logical Operations . . . 33
A.4 Set Operations . . . 33
A.5 Array Operations . . . 34
A.6 Coercion Operations . . . 35
A.7 String Operations . . . 35
A.8 Bound and Domain Operations . . . 35
A.9 Other Operations . . . 36

B Libraries **38**
B.1 `globals.zinc` . . . 38

C Standard Annotations **39**
C.1 Annotations . . . 39
C.1.1 Null Annotation . . . 39
C.1.2 Solve Annotations . . . 39

D Zinc Grammar **40**
D.1 Items . . . 40
D.2 Type-Inst Expressions . . . 40
D.3 Expressions . . . 41
D.4 Miscellaneous Elements . . . 43

E MiniZinc **44**
E.1 Items 44
E.2 Type-insts and Expressions 44
E.3 Built-in Operations and Annotations 45
E.4 Other 45

1 Introduction

This document defines Zinc, a language for modelling constraint satisfaction and optimisation problems.

Zinc is a high-level, typed, mostly first-order, functional, modelling language. It provides:

- mathematical notation-like syntax (automatic coercions, overloading, iteration, sets, arrays);
- expressive constraints (finite domain, set, linear arithmetic, integer);
- support for different kinds of problems (satisfaction, explicit optimisation, preference (soft constraints));
- separation of data from model;
- high-level data structures and data encapsulation (sets, arrays, tuples, records, enumerated types, constrained type-insts);
- extensibility (user-defined functions and predicates);
- reliability (type checking, instantiation checking, assertions);
- solver-independent modelling;
- simple, declarative semantics.

Zinc extends OPL and moves closer to CLP languages such as ECLiPSe.

This document also defines MiniZinc, a smaller language that is a strict subset of Zinc.

This document has the following structure. Section 2 provides a high-level overview of Zinc models. Section 3 covers syntax basics. Section 4 covers high-level structure: items, multi-file models, namespaces, and scopes. Section 5 introduces types and type-insts. Section 6 covers expressions. Section 7 describes the top-level items in detail. Section 8 describes annotations. Section 9 describes how partiality is handled in various cases. Appendix A describes the language built-ins. Appendix B describes some language libraries. Appendix C describes the standard language annotations. Appendix D gives the Zinc grammar. Appendix E gives the definition of MiniZinc.

This document also provides some explanation of why certain design decisions were made. Such explanations are marked by the word "Rationale" and written in italics, and do not constitute part of the specification as such. ***Rationale.*** *These explanations are present because they are useful to both the designers and the users of Zinc.*

2 Overview of a Model

2.1 Specifying a Problem

Conceptually, a Zinc problem specification has two parts.

1. The *model*: the main part of the problem specification, which describes the structure of a particular class of problems.
2. The *data*: the input data for the model, which specifies one particular problem within this class of problems.

The pairing of a model with a particular data set is an *model instance* (sometimes abbreviated to *instance*).

The model and data may be separated, or the data may be "hard-wired" into the model. Section 4.2 specifies how the model and data can be structured within files in a model instance.

2.2 Evaluation of a Model Instance

2.2.1 Evaluation Phases

A Zinc model instance is evaluated in three distinct phases.

1. Model-time: data-independent static checking of the model.
2. Instance-time: static checking of the model instance.
3. Run-time: evaluation of the instance (i.e. constraint solving).

If the model and data are not separated (i.e. the data is "hard-wired" into the model, see Section 4.2) the model-time and instance-time phases will effectively be combined.

2.2.2 Evaluation Outcomes

There are four possible evaluation outcomes.

1. Static error: the model instance does not compile due to a problem with the model and/or data, detected at model-time or instance-time. This could be caused by a syntax error, a type-inst error, the use of an unsupported feature or operation, etc.
2. Run-time error: the evaluation fails to complete due to a problem with the model and/or data, detected at run-time. This could be caused by an assertion failure, division by zero, an array bounds error, etc. Alternatively, it could be due to an implementation shortcoming, such as a time-out due to excessive evaluation time, failure to determine if any solutions are possible, an overflow on an integer operation, etc.
3. Failure: no solutions are returned due to unsatisfiable constraints.
4. Success: one or more solutions are returned.

Static errors must be detected prior to run-time. The remaining outcomes—which can only occur for instances without static errors—are determined at run-time. However, an implementation is free to determine them earlier if it safely can. For example, an implementation may be able to determine that unsatisfiable constraints exist prior to run-time, and the resulting messages given to the user may be more helpful than if the unsatisfiability is detected at run-time.

An implementation must produce output in all outcomes. The form of the output in the error cases is implementation-dependent. The form of the output in the failure and success cases is described in Section 7.8.

An implementation may produce warnings during all evaluation phases.

3 Syntax Overview

3.1 Character Set

Zinc currently allows only ASCII characters. In the future we hope to support Unicode.

Zinc is case sensitive. There are no places where upper-case or lower-case letters must be used.

Zinc has no layout restrictions, i.e. any single piece of whitespace (containing spaces, tabs and newlines) is equivalent to any other.

3.2 Comments

A `%` indicates that the rest of the line is a comment. Zinc has no begin/end comment symbols (such as C's `/*` and `*/` comments).

3.3 Syntax Notation

The basics of the EBNF used for the Zinc grammar are as follows.

- Non-terminals are written between angle brackets, e.g. ⟨*item*⟩.
- Terminals are written in fixed-width font and underlined, e.g. `constraint`.
- Optional items are written in square brackets, e.g. [`var`].
- Sequences of zero or more items are written with parentheses and a star, e.g. (`,` ⟨*ident*⟩)*.
- Non-empty lists are written with an item, a separator/terminator terminal, and "...". For example, this:

  ⟨*expr*⟩ `,` ...

  is short for this:

  ⟨*expr*⟩ (`,` ⟨*expr*⟩)* [`,`]

  The final terminal is always optional in non-empty lists.
- Regular expressions, written in fixed-width font, are used in some productions, e.g. `[-+]?[0-9]+`.

Zinc's grammar is presented piece-by-piece throughout this document. It is also available as a whole in Appendix D.

3.4 Identifiers

Identifiers have the following syntax:

⟨*ident*⟩ ::= `[A-Za-z][A-Za-z0-9_]*` % excluding keywords

For example:

```
my_name_2
MyName2
```

A number of keywords are reserved and cannot be used as identifiers. The keywords are: `annotation`, `any`, `array`, `bool`, `case`, `constraint`, `diff`, `div`, `else`, `elseif`, `endif`, `enum`, `false`, `float`, `function`, `if`, `in`, `include`, `int`, `intersect`, `let`, `list`, `maximize`, `minimize`, `mod`, `not`, `of`, `satisfy`, `subset`, `superset`, `output`, `par`, `predicate`, `record`, `set`, `solve`, `string`, `symdiff`, `test`, `then`, `true`, `tuple`, `union`, `type`, `var`, `where`, `xor`.

A number of identifiers are used for built-ins; see Section A for details.

4 High-level Model Structure

4.1 Items

A Zinc model consists of multiple *items*:

⟨*model*⟩ ::= [⟨*item*⟩ `;` ...]

Items can occur in any order; identifiers need not be declared before they are used.

Items have the following top-level syntax:

⟨*item*⟩ ::= ⟨*type-inst-syn-item*⟩
| ⟨*enum-item*⟩
| ⟨*include-item*⟩
| ⟨*var-decl-item*⟩
| ⟨*assign-item*⟩
| ⟨*constraint-item*⟩
| ⟨*solve-item*⟩
| ⟨*output-item*⟩
| ⟨*predicate-item*⟩
| ⟨*test-item*⟩
| ⟨*function-item*⟩
| ⟨*annotation-item*⟩

Type-inst synonym items and enumerated type items define new types.

Include items provide a way of combining multiple files into a single instance. This allows a model to be split into multiple files (Section 7.3).

Variable declaration items introduce new global variables and possibly bind them to a value (Section 7.4).

Assignment items bind values to global variables (Section 7.5).

Constraint items describe model constraints (Section 7.6).

Solve items are the "starting point" of a model, and specify exactly what kind of solution is being looked for: plain satisfaction, or the minimization/maximization of an expression. Each model must have exactly one solve item (Section 7.7).

Output items are used for nicely presenting the result of a model execution (Section 7.8). *Note:* models to be converted to FlatZinc should use the built-in `is_output` annotation on variables rather than an output item. If an output item is provided, then the corresponding FlatZinc model should have output annotations placed on the variables appearing in the original output item.

Predicate items, test items (which are just a special type of predicate) and (Zinc-only) function items introduce new user-defined predicates and functions which can be called in expressions (Section 7.10). Predicates, functions, and built-in operators are described collectively as *operations*.

Annotation items augment the `ann` type, values of which can specify non-declarative and/or solver-specific information in a model.

4.2 Model Instance Files

Zinc models can be constructed from multiple files using include items (see Section 7.3). Zinc has no module system as such; all the included files are simply concatenated and processed as a whole, exactly as if they had all been part of a single file. ***Rationale.*** *We have not found much need for one so far. If bigger models become common and the single global namespace becomes a problem, this should be reconsidered.*

Each model may be paired with one or more data files. Data files are more restricted than model files. They may only contain variable assignments (see Section 7.5) and definitions of flat enums that were declared in a model file.

Data files may not include calls to user-defined operations.

Models do not contain the names of data files; doing so would fix the data file used by the model and defeat the purpose of allowing separate data files. Instead, an implementation must allow one or more data files to be combined with a model for evaluation via a mechanism such as the command-line.

An implementation should allow a model to be checked with and without its instance data. When checking a model without data, all global variables with fixed type-insts need not be assigned. When checking a model with data, all global variables with fixed type-insts must be assigned, unless they are not used (in which case they can be removed from the model without effect).

A data file can only be checked for static errors in conjunction with a model, since the model contains the declarations that include the types of the variables assigned in the data file.

A single data file may be shared between multiple models, so long as the definitions are compatible with all the models.

4.3 Namespaces

All names declared at the top-level belong to a single namespace. It includes the following names.

1. All global variable names.
2. All function and predicate names, both built-in and user-defined.
3. All user-defined type and type-inst names (type-inst synonyms and enumerated types).
4. All enum case names.
5. All annotation names.

Because multi-file Zinc models are composed via concatenation (Section 4.2), all files share this top-level namespace. Therefore a variable `v` declared in one model file could not be declared with a different type in a different file, for example.

Zinc supports overloading of built-in and user-defined operations.

Zinc has two kinds of local namespace: each record and (non-flat) enum has its own local namespace for field names. This means distinct records and (non-flat) enums can use the same field names. All names in these local namespaces co-exist without conflict with identical names in the top-level namespace—in any situation, which namespace applies can always be determined from context.

4.4 Scopes

Within the top-level namespace, there are several kinds of local scope that introduce local names:

- Comprehension expressions (Section 6.3.7).
- Let expressions (Section 6.3.23).
- Function and predicate argument lists and bodies (Section 7.10).
- Type-inst constraints (Section 5.9.3).

The listed sections specify these scopes in more detail. In each case, any names declared in the local scope overshadow identical global names.

5 Types and Type-insts

Zinc provides four scalar built-in types: Booleans, integers, floats, and strings; several compound built-in types: sets, multi-dimensional arrays with arbitrary index types, tuples, and records; and three kinds of user-defined types: type-inst synonyms, enumerated types, and `ann`, a user-extensible type that represents annotations. Zinc also allows type-inst variables in certain places, and has some very limited higher-order types.

Each type has one or more possible *instantiations*. The instantiation of a variable or value indicates if it is fixed to a known value or not. A pairing of a type and instantiation is called a *type-inst*.

Zinc also supports *constrained type-insts*, which are type-insts with an additional expression that constrains their possible values.

We begin by discussing some properties that apply to every type. We then introduce instantiations in more detail. We then cover each type individually, giving: an overview of the type and its possible instantiations, the syntax for its type-insts, whether it is a finite type (and if so, its domain), whether it is varifiable, the ordering and equality operations, whether its variables must be initialised at instance-time, and whether it can be involved in automatic coercions. We conclude by describing Zinc's constrained type-insts.

5.1 Properties of Types

The following list introduces some general properties of Zinc types.

- Currently all types are monotypes. Recursive types are not allowed.

  In the future we may allow types which are polymorphic in other types and also the associated constraints.

- We distinguish types which are *finite types*.

  In Zinc, finite types include Booleans, flat enums, types defined via set expression type-insts such as range types (see Section 5.9.1), as well as sets, arrays, tuples, records and non-flat enums composed of finite types. Types that are not finite types are unconstrained integers, unconstrained floats, unconstrained strings, and `ann`. Finite types are relevant to sets (Section 5.6.1) and array indices (Section 5.6.2).

  Every finite type has a *domain*, which is a set value that holds all the possible values represented by the type.

- Every first-order type (this excludes `ann`) has a built-in total order and a built-in equality; `>`, `<`, `==`/`=`, `!=`, `<=` and `>=` comparison operators can be applied to any pair of values of the same type. The ordering for user-defined types is the standard lexicographic ordering. ***Rationale.*** *This facilitates the specification of symmetry breaking and of polymorphic predicates and functions.* Note that, as in most languages, using equality on floats or types that contain floats is generally not reliable due to their inexact representation. An implementation may choose to warn about the use of equality with floats or types that contain floats.

- Higher-order types are used in very limited ways. They can only be used with the built-in functions `foldl` and `foldr`, which both take a function as their first argument.

5.2 Instantiations

When a Zinc model is evaluated, the value of each variable may initially be unknown. As it runs, each variable's *domain* (the set of values it may take) may be reduced, possibly to a single value.

An *instantiation* (sometimes abbreviated to *inst*) describes how fixed or unfixed a variable is at instance-time. At the most basic level, the instantiation system distinguishes between two kinds of variables:

1. *Parameters*, whose values are fixed at instance-time (usually written just as "fixed").
2. *Decision variables* (often abbreviated to *variables*), whose values may be completely unfixed at instance-time, but may become fixed at run-time (indeed, the fixing of decision variables is the whole aim of constraint solving).

There are also intermediate instantiations for some compound types—they can have a fixed size but may contain unfixed elements.

In Zinc decision variables can have the following types: Booleans, integers, floats, sets, and flat enums. Tuples, arrays, records, non-flat enums and `ann` can contain decision variables.

5.3 Type-insts

Because each variable has both a type and an inst, they are often combined into a single *type-inst*. Type-insts are primarily what we deal with when writing models, rather than types.

A variable's type-inst *never changes*. This means a decision variable whose value becomes fixed during model evaluation still has its original type-inst (e.g. `var int`), because that was its type-inst at instance-time.

Some type-insts can be automatically coerced to another type-inst. For example, if a `par int` value is used in a context where a `var int` is expected, it is automatically coerced to a `var int`. We write this `par int` $\xrightarrow{c}$ `var int`. Also, any type-inst can be considered coercible to itself. Zinc allows coercions between some types as well.

Some type-insts can be *varified*, i.e. made unfixed at the top-level. For example, `par int` is varified to `var int`. We write this `par int` $\xrightarrow{v}$ `var int`.

Type-insts that are varifiable include the type-insts of the types that can be decision variables (Booleans, integers, floats, sets, enumerated types), and also tuples and records, if their constituent elements can be varified. Varification is relevant to type-inst synonyms and array accesses.

5.4 Type-inst Expressions Overview

This section partly describes how to write type-insts in Zinc models. Further details are given for each type as they are described in the following sections.

A type-inst expression specifies a type-inst.

Type-inst expressions may include type-inst constraints.

Type-inst expressions appear in variable declarations (Section 7.4), user-defined operation items (Section 7.10), and type-inst synonyms (Section 7.1).

Type-inst expressions have this syntax:

⟨*ti-expr*⟩ ::= `(` ⟨*ti-expr*⟩ `:` ⟨*ident*⟩ `where` ⟨*expr*⟩ `)`
| ⟨*base-ti-expr*⟩

The first alternative is for arbitrarily constrained type-insts, which are described in Section 5.9.3.

The second alternative is for base type-inst expressions, which have the following syntax:

⟨*base-ti-expr*⟩ ::= ⟨*var-par*⟩ ⟨*base-ti-expr-tail*⟩
⟨*var-par*⟩ ::= `var` | `par` | ϵ
⟨*base-ti-expr-tail*⟩ ::= ⟨*ident*⟩
| `bool`
| `int`
| `float`
| `string`
| ⟨*set-ti-expr-tail*⟩
| ⟨*array-ti-expr-tail*⟩
| ⟨*tuple-ti-expr-tail*⟩
| ⟨*record-ti-expr-tail*⟩
| ⟨*ti-variable-expr-tail*⟩
| `ann`
| ⟨*op-ti-expr-tail*⟩
| `{` ⟨*expr*⟩ `,` ... `}`
| ⟨*num-expr*⟩ `..` ⟨*num-expr*⟩

(The final alternative, for range types, uses the integer-specific ⟨*int-expr*⟩ non-terminal, defined in Section 6.1, rather than the ⟨*expr*⟩ non-terminal. If this were not the case, the rule would never match because the '`..`' operator would always be matched by the first ⟨*expr*⟩.)

This fully covers the type-inst expressions for scalar types. The compound type-inst expression syntax is covered in more detail in Section 5.6. Type-inst variable syntax is described in more detail in Section 5.8.1.

The constrained type-inst expressions are covered in more detail in Section 5.9.

The `par` and `var` keywords (or lack of them) determine the instantiation. The `par` annotation can be omitted; the following two type-inst expressions are equivalent:

```
par int
int
```

Rationale. *The use of the explicit* `var` *keyword allows an implementation to check that all parameters are initialised in the model or the instance. It also clearly documents which variables are parameters, and allows more precise type-inst checking.*

A type-inst is fixed if it does not contain `var` or `any`, with the exception of `ann`.

Note that several type-inst expressions that are syntactically expressible represent illegal type-insts. For example, although the grammar allows `var` in front of all these base type-inst expression tails, it is a type-inst error to have `var` in the front of a string, array, tuple, record, or type-inst variable type-inst expression.

5.5 Built-in Scalar Types and Type-insts

5.5.1 Booleans

Overview. Booleans represent truthhood or falsity. ***Rationale.*** *Boolean values are not represented by integers, nor are they coercible to integers, because either option would be error-prone. Booleans can be explicit converted to integers with the* `bool2int` *function, which makes the user's intent clear.*

Allowed Insts. Booleans can be fixed or unfixed.

Syntax. Fixed Booleans are written `bool` or `par bool`. Unfixed Booleans are written as `var bool`.

Finite? Yes. The domain of a Boolean is {`false, true`}.

Varifiable? `par bool` $\xrightarrow{v}$ `var bool`, `var bool` $\xrightarrow{v}$ `var bool`.

Ordering. The value `false` is considered smaller than `true`.

Initialisation. A fixed Boolean variable must be initialised at instance-time; an unfixed Boolean variable need not be.

Coercions. `par bool` $\xrightarrow{c}$ `var bool`.

5.5.2 Integers

Overview. Integers represent integral numbers. Integer representations are implementation-defined. This means that the representable range of integers is implementation-defined. However, an implementation should abort at run-time if an integer operation overflows.

Allowed Insts. Integers can be fixed or unfixed.

Syntax. Fixed integers are written `int` or `par int`. Unfixed integers are written as `var int`.

Finite? Not unless constrained by a set expression (see Section 5.9.1).

Varifiable? `par int` $\xrightarrow{v}$ `var int`, `var int` $\xrightarrow{v}$ `var int`.

Ordering. The ordering on integers is the standard one.

Initialisation. A fixed integer variable must be initialised at instance-time; an unfixed integer variable need not be.

Coercions. `par int` $\xrightarrow{c}$ `var int`.

Also, integers can be automatically coerced to floats; see Section 5.5.3.

5.5.3 Floats

Overview. Floats represent real numbers. Float representations are implementation-defined. This means that the representable range and precision of floats is implementation-defined. However, an implementation should abort at run-time on exceptional float operations (e.g. those that produce `NaN`s, if using IEEE754 floats).

Allowed Insts. Floats can be fixed or unfixed.

Syntax. Fixed floats are written `float` or `par float`. Unfixed floats are written as `var float`.

Finite? Not unless constrained by a set expression (see Section 5.9.1).

Varifiable? `par float` $\xrightarrow{v}$ `var float`, `var float` $\xrightarrow{v}$ `var float`.

Ordering. The orderings on floats are the standard ones.

Initialisation. A fixed float variable must be initialised at instance-time; an unfixed float variable need not be.

Coercions. `par int` $\xrightarrow{c}$ `par float`, `par int` $\xrightarrow{c}$ `var float`, `par float` $\xrightarrow{c}$ `var float`.

5.5.4 Strings

Overview. Strings are primitive, i.e. they are not lists of characters.

String expressions are used in assertions, output items and annotations, and string literals are used in include items.

Allowed Insts. Strings must be fixed.

Syntax. Fixed strings are written `string` or `par string`.

Finite? Not unless constrained by a set expression (see Section 5.9.1).

Varifiable? No.

Ordering. Strings are ordered lexicographically using the underlying character codes.

Initialisation. A string variable (which can only be fixed) must be initialised at instance-time.

Coercions. None automatic. However, any non-string value can be manually converted to a string using the built-in `show` function.

5.6 Built-in Compound Types and Type-insts

5.6.1 Sets

Overview. A set is a collection with no duplicates.

Allowed Insts. The type-inst of a set's elements must be fixed. ***Rationale.*** *This is because current solvers are not powerful enough to handle sets containing decision variables.* Sets may contain any type, and may be fixed or unfixed. If a set is unfixed, its elements must be finite, unless it appears in an argument of a predicate, function or annotation.

Syntax. A set base type-inst expression tail has this syntax:

⟨*set-ti-expr-tail*⟩ ::= `set of` ⟨*ti-expr*⟩

Some example set type-inst expressions:

```
set of int
var set of bool
```

Finite? Yes, if the set elements are finite types. Otherwise, no.

The domain of a set type that is a finite type is the powerset of the domain of its element type. For example, the domain of `set of 1..2` is `powerset(1..2)`, which is `{{}, {1}, {2,2}, {2}}`.

Varifiable? `par set of TI` $\xrightarrow{v}$ `var set of TI`, `var set of TI` $\xrightarrow{v}$ `var set of TI`,

Ordering. The pre-defined ordering on sets is the lexicographic ordering on the corresponding *characteristic arrays*. For example,

```
{}  <  {2}  <  {1,3}
```

since

```
[0,0,0] < [0,1,0] < [1,0,1].
```

MiniZinc: The pre-defined ordering on sets is a lexicographic ordering of the *sorted set form*, where {1,2} is in sorted set form, for example, but {2,1} is not. This means, for instance, `{} < {1,3} < {2}`.

Rationale. *The order based on the characteristic array seems easier to propagate well (as it reduces to the array case) than the perhaps more natural one based on the sorted set form.*

Initialisation. A fixed set variable must be initialised at instance-time; an unfixed set variable need not be.

Coercions. `par set of TI` $\xrightarrow{c}$ `par set of UI` and `par set of TI` $\xrightarrow{c}$ `var set of UI` and `var set of TI` $\xrightarrow{c}$ `var set of UI`, if `TI` $\xrightarrow{c}$ `UI`.

Fixed sets can be automatically coerced to arrays; see section 5.6.2. This means that array accesses can be used on fixed sets; `S[1]` is the smallest element in a fixed set `S` while `S[card(S)]` is the largest.

5.6.2 Arrays

Overview. Zinc arrays are maps from fixed keys (a.k.a. indices) to values. Keys and values can be of any type. When used with integer keys, Zinc arrays can be used like arrays in languages like Java, but with other types of key they act like associative arrays. Using floats or types containing floats as keys can be dangerous because of their imprecise equality comparison, and an implementation may give a warning in this case. The values can have any type-inst. Arrays-of-arrays are allowed.

All arrays are one-dimensional. However, multi-dimensional arrays can be simulated using a tuple as the index, and there is some syntactic sugar to make this easier (see below and in Section 6.3.13).

Zinc arrays can be declared in two different ways.

1. *Explicitly-indexed* arrays have index types in the declaration that are finite types. For example:

   ```
   array[0..3] of int: a1;
   ```

   For such arrays, the index type specifies exactly the indices that will be in the array—the array's index set is the *domain* of the index type—and if the indices of the value assigned do not match then it is a run-time error.

   For example, the following assignments cause run-time errors:

   ```
   a1 = [0:0, 1:1, 2:2, 3:3, 4:4];            % too many elements
   array[1..5, 1..10] of var float: a5 = [];  % too few elements
   ```

2. *Implicitly-indexed* arrays have index types in the declaration that are not finite types. For example:

   ```
   array[int] of int: a6;
   ```

   No checking of indices occurs when these variables are assigned.

The initialisation of an array can be done in a separate assignment statement, which may be present in the model or a separate data file.

Arrays can be accessed. See Section 6.3.13 for details.

Allowed Insts. An array's size must be fixed. Its indices must also have fixed type-insts. Its elements may be fixed or unfixed.

Syntax. An array base type-inst expression tail has this syntax:

⟨*array-ti-expr-tail*⟩ ::= `array [` ⟨*ti-expr*⟩ `,` ... `]` `of` ⟨*ti-expr*⟩
| `list of` ⟨*ti-expr*⟩

Some example array type-inst expressions:

```
array[1..10] of int
list of var int
```

Note that `list of <T>` is just syntactic sugar for `array[int] of <T>`. ***Rationale.*** *Integer-indexed arrays of this form are very common, and so worthy of special support to make things easier for modellers. Implementing it using syntactic sugar avoids adding an extra type to the language, which keeps things simple for implementers.*

Because arrays must be fixed-size it is a type-inst error to precede an array type-inst expression with `var`.

Syntactic sugar exists for declaring tuple-indexed arrays. For example, the second of the following two declarations is syntactic sugar for the first.

```
array[tuple(1..5, 1..4)] of int: a5;
array[1..5, 1..4]        of int: a5;
```

Finite? Yes, if the index types and element type are all finite types. Otherwise, no.

The domain of an array type that is a finite array is the set of all distinct arrays of the appropriate length. For example, the domain of `array[5..6] of 1..2` is `{[5:1,6:1], [5:1,6:2], [5:2,6:1], [5:2,6:2]}`.

Varifiable? No.

Ordering. Arrays are ordered lexicographically, taking into consideration first keys and then values. For example, `[1:1, 2:2, 3:3]` is less than `[2:0, 3:0, 4:0]`.

Initialisation. An explicitly-indexed array variable must be initialised at instance-time only if its elements must be initialised at instance time. An implicitly-indexed array variable must be initialised at instance-time so that its length and index set is known.

Coercions. `set of TI` $\xrightarrow{c}$ `array[1..n] of UI` if `TI` $\xrightarrow{c}$ `UI`, where `n` is the number of elements in the set. The elements of the resulting array will be in sorted order. This means that elements of fixed sets can be accessed like array elements, using square brackets.

5.6.3 Tuples

Overview. Tuples are fixed-size, heterogeneous collections. They must contain at least two elements; unary tuples are not allowed.

Allowed Insts. Tuples may contain unfixed elements.

Syntax. A tuple base type-inst expression tail has this syntax:

⟨*tuple-ti-expr-tail*⟩ ::= `tuple` `(` ⟨*ti-expr*⟩ `,` ... `)`

An example tuple type-inst expression:

```
tuple(int, var float)
```

It is a type-inst error to precede a tuple type-inst expression with `var`.

Finite? Yes, if all its constituent elements are finite types. Otherwise, no.

The domain of a tuple type that is a finite type is the cartesian product of the domains of the element types. For example, the domain of `tuple(1..2, {3,5})` is `{(1,3), (1,5), (2,3), (2,5)}`.

Varifiable? Yes, if all its constituent elements are varifiable. A tuple is varified by varifying its constituent elements.

Ordering. Tuples are ordered lexicographically.

Initialisation. A tuple variable must be initialised at instance-time if any of its constituent elements must be initialised at instance-time.

Coercions. `tuple(TI1,...,TIn)` $\xrightarrow{c}$ `tuple(UI1,...UIn)`, if `TI1` $\xrightarrow{c}$ `UI1`,...,`TIn` $\xrightarrow{c}$ `UIn`. Also, tuples can be automatically coerced to records; see Section 5.6.4 for details.

5.6.4 Records

Overview. Records are fixed-size, heterogeneous collections. They are similar to tuples, but have named fields.

Field names in different records can be identical, because each record's field names belong to a different namespace (Section 4.3).

Record field order is significant; the following two record type-insts are distinct and do not match:

```
record(var int: x, var int: y)
record(var int: y, var int: x)
```

Rationale. *This is necessary to avoid ambiguity when tuples are coerced to records. If field ordering did not matter, we could have this array:*

```
[(x:3, y:4), (y:2, x:5), (7,8)]
```

in which the `(7,8)` *must be coerced to a record, but it is unclear if it should become* `(x:7, y:8)` *or* `(y:7, x:8)`*.*

Allowed Insts. Records may contain unfixed elements.

Syntax. A record base type-inst expression tail has this syntax:

⟨*record-ti-expr-tail*⟩ ::= `record` `(` ⟨*ti-expr-and-id*⟩ `,` ... `)`
⟨*ti-expr-and-id*⟩ ::= ⟨*ti-expr*⟩ `:` ⟨*ident*⟩

An example record type-inst expression:

```
record(int: x, int: y)
```

It is a type-inst error to precede a record type-inst expression with `var`.

Finite? Yes, if all its constituent elements are finite types. Otherwise, no.

The domain of a record type that is a finite type is the same as that of a tuple type, but with the fields included. For example, the domain of `record(1..2:x, {3,5}:y)` is `{(x:1,y:3), (x:1,y:5), (x:2,y:3), (x:2,y:5)}`.

Varifiable? Yes, if all its constituent elements are varifiable. A record is varified by varifying its constituent elements.

Ordering. Records are ordered lexicographically according to the values of the fields. The field names are irrelevant for comparisons because in order for two records to be compared they must have the same type-inst, in which case the field names and order must be the same.

Initialisation. A record variable must be initialised at instance-time if any of its constituent elements must be initialised at instance-time.

Coercions. `tuple(TI1,...,TIn)` $\xrightarrow{c}$ `record(UI1:x1,...UIn:xn)`, if `TI1` $\xrightarrow{c}$ `UI1`,...,`TIn` $\xrightarrow{c}$ `UIn`. This is useful for record initialisation. For example, we can initialise a record of type *Task* (defined in Section 5.9.3) using:

```
Task: T = (10,_,_);
```

which initialises `duration` field with 10 and the variable fields `start` and `finish` with '`_`'.

Also, `record(TI1:x1,...,TIn:xn)` $\xrightarrow{c}$ `record(UI1:x1,...UIn:xn)`, if `TI1` $\xrightarrow{c}$ `UI1`,...,`TIn` $\xrightarrow{c}$ `UIn`.

5.7 User-defined Types and Type-insts

This section introduces the properties of the user-defined types and type-insts. The syntax and details of the items that are used to declare these new types are in Section 7.

Because these user-defined types have names, their type-inst expressions are simple identifiers. Syntactically, any identifier may be used as a base type-inst expression. However, in a valid model any identifier within a base type-inst expression must be one of:

- the name of a user-defined type or type-inst (type-inst synonym or enumerated type);
- the name of a fixed set value (Section 5.9.1).

5.7.1 Type-inst Synonyms

Overview. A type-inst synonym is an alternative name for a pre-existing type-inst which can be used interchangeably with the pre-existing type-inst. For example, if `MyFixedInt` is a synonym for `par int` then `MyFixedInt` can be used anywhere `par int` can, and vice versa.

Allowed Insts. Preceding a type-inst synonym with `var` varifies it, unless the type-inst is not varifiable, in which case it is a type-inst error. Preceding a type-inst synonym with `par` has no effect—the `par` is ignored.

Syntax. A type-inst synonym named "X" is represented by the term `X`.

Finite? As for the pre-existing type-inst.

Varifiable? Yes, if the pre-existing type-inst is varifiable.

Ordering. As for the pre-existing type-inst.

Initialisation. As for the pre-existing type-inst.

Coercions. As for the pre-existing type-inst.

5.7.2 Enumerated Types

Overview. Enumerated types (or *enums* for short) provide a set of named alternatives. Unlike many languages, Zinc's enumerated types can have arguments, and each argument has a field name, so they are more like discriminated unions. However, they cannot be recursive. ***Rationale.*** *This restriction naturally follows from the decision to disallow recursive functions and predicates (see Section 7.10.1). Without such capabilities, recursive data structures are of little use as there is no general way to traverse them.*

We distinguish between *flat* enums, in which all the alternatives have no arguments, and *non-flat* enums, in which some or all alternatives have arguments. Each alternative is identified by its *case name.*

Allowed Insts. Flat enums can be fixed or unfixed. Non-flat enums cannot be preceded by `var`, but they may contain unfixed elements.

Syntax. Variables of an enumerated type named "X" are represented by the term `X` or `par X` if fixed, and (flat enums only) `var X` if unfixed.

Finite? If flat, yes. If non-flat, only if all its constituent elements (in all alternatives) are finite types; otherwise, no.

The domain of a flat enum is the set containing all of its case names. The domain of a non-flat enum is the set containing all the possible values of that enum. For example, these two enums:

```
enum C = { R, G, B };
enum X = { a(1..3:x), b(bool:y), c };
```

have these domains:

```
{ R, G, B }
{ a(x:1), a(x:2), a(x:3), b(y:false), b(y:true), c }
```

Varifiable? For flat enums: `par X` $\xrightarrow{v}$ `var X`, `var X` $\xrightarrow{v}$ `var X`.

For non-flat enums: no.

Ordering. When two enum values with different case names are compared, the value with the case name that is declared first is considered smaller than the value with the case name that is declared second. If the case names are the same, the ordering is lexicographic on the case arguments (if there are any).

Initialisation. A fixed flat enum variable must be initialised at instance-time; an unfixed flat enum variable need not be.

A non-flat enum variable must be initialised at instance-time if any of its constituent elements (in any of the alternatives) must be initialised at instance-time.

Coercions. For flat enums: `par X` $\xrightarrow{c}$ `var X`. For non-flat enums: none.

5.7.3 The Annotation Type `ann`

Overview. The annotation type, `ann`, can be used to represent arbitrary term structures. It is augmented by annotation items (7.9).

Allowed Insts. `ann` is always considered unfixed, because it may contain unfixed elements. It cannot be preceded by `var`.

Syntax. The annotation type is written `ann`.

Finite? No.

Varifiable? No.

Ordering. N/A. Annotation types do not have an ordering defined on them.

Initialisation. An `ann` variable must be initialised at instance-time.

Coercions. None.

5.8 Other Types and Type-insts

5.8.1 Type-inst Variables

Overview. Type-inst variables allow parametric polymorphism. They can appear in Zinc predicate and function arguments and return type-insts, in let expressions within predicate and function bodies, and in annotation declarations; if one is used outside a function or predicate or annotation declaration it is a type-inst error.

Allowed Insts. A type-inst variable expression consists of a type-inst variable and an optional prefix. Type-inst variables can be prefixed by `par`, in which case they match any fixed type-inst; the same is true if they lack a prefix. Type-inst variables can also be prefixed by `any`, in which case they match any first-order type-inst.

The meanings of the prefixes are discussed in further detail in Section 7.10.3.

Syntax. A type-inst variable expression tail has this syntax:

⟨*ti-variable-expr-tail*⟩ ::= [`any`] `$[A-Za-z][A-Za-z0-9_]*`

Some example type-inst variable expressions:

```
$T
par $U3
any $xyz
```

Finite? No. This is because they can be bound to any type-inst, and not all type-insts are finite.

Varifiable? No. This is because they can be bound to any type-inst, and not all type-insts can be varified.

Ordering. Values of equal type-inst variables can be compared. The comparison used will be the comparison of the underlying type-insts. This is possible because all type-insts have a built-in ordering.

Initialisation. A variable with a type-inst variable as its type-inst (which can only appear in let expressions) must be initialised if its prefix is `par`, otherwise it need not be.

Coercions. `par $T` $\xrightarrow{c}$ `any $T`.

5.8.2 Higher-order Types

Overview. Operations (e.g. predicates) have higher-order types. Operations can be used as values when passed as the first argument to `foldl` or `foldr`. They can also be used as arguments to annotations; annotation declarations are the only place where operation type-inst expressions are permitted.

Allowed Insts. The type-inst of a higher-order type is determined by the type-insts it can take as its arguments, and its return type-inst. A higher-order type value is never considered fixed, and so cannot be used as the element of a set, for example.

Syntax. An operation type-inst expression tail has this syntax:

⟨*op-ti-expr-tail*⟩ ::= `op` `(` ⟨*ti-expr*⟩ `:` `(` ⟨*ti-expr*⟩ `,` ... `)` `)`

Some example type-inst variable expressions:

```
op(int:(int));
op(var bool:(float, any $T));
```

Note that operation type-inst expressions cannot be written for nullary operations (i.e. those lacking arguments). ***Rationale.*** *This is because it is difficult to distinguish the name of a nullary operation from a call to it, and so passing nullary operations is difficult. Furthermore, they are little to no use, so they are unlikely to be missed.*

Finite? No.

Varifiable? +No.

Ordering. N/A. Higher-order types cannot be used in comparisons; it is a run-time error if they are.

Initialisation. N/A. Variables cannot have higher-order types.

Coercions. None.

5.9 Constrained Type-insts

One powerful feature of Zinc is *constrained type-insts*. A constrained type-inst is a restricted version of a *base* type-inst, i.e. a type-inst with fewer values in its domain.

5.9.1 Set Expression Type-insts

Three kinds of expressions can be used in type-insts.

1. Integer ranges: e.g. `1..3`.
2. Set literals: e.g. `var {1,3,5}`.
3. Identifiers: the name of a set parameter (which can be global, let-local, the argument of a predicate or function, or a generator value) can serve as a type-inst.

In each case the base type is that of the set's elements, and the values within the set serve as the domain. For example, whereas a variable with type-inst `var int` can take any integer value, a variable with type-inst `var 1..3` can only take the value 1, 2 or 3.

All set expression type-insts are finite types. Their domain is equal to the set itself.

5.9.2 Float Range Type-insts

Float ranges can be used as type-insts, e.g. `1.0 .. 3.0`. These are treated similarly to integer range type-insts, although `1.0 .. 3.0` is not a valid expression whereas `1 .. 3` is.

Float ranges are not finite types.

5.9.3 Arbitrarily Constrained Type-insts

A more general form of constrained type-inst allows an arbitrary Boolean *type-inst constraint* expression to be applied to a *base* type-inst.

Here are two examples of arbitrarily constrained type-inst expressions. The first is a fixed integer in the range 1–3, and the second is an unfixed non-negative float.

```
(par int:   i where i in 1..3): dom;
(var float: f where f >= 0)   : fplus;
```

The base type-inst appears before the ':'. The identifiers `i` and `f` are local identifiers used in the Boolean expression after the `where` keyword. The scope of the local identifiers `i` and `f` extends to the end of the Boolean expression after the `where`.

An arbitrarily constrained type-inst is finite only if its base type-inst is finite. Its domain is that of its base type-inst, minus those elements that do not satisfy its constraint.

Finiteness is thus the main difference between set expression type-insts (Section 5.9.1) and arbitrarily constrained type-insts. For example, in the following three lines, the first type-inst is equivalent to the second, with one exception.

```
par 1..3        (par int: i where i in 1..3)
var {1,2,3}     (var int: i where i in {1,2,3})
MySet           (    int: i where i in MySet)
```

The exception is that the type-insts on the left-hand side are finite, whereas those on the right are non-finite.

Every float range type-inst has an equivalent arbitrarily constrained type-inst. For example, the following two type-insts are equivalent:

```
1.0 .. 3.0      (float: f where 1.0 <= f /\ f <= 3.0)
```

An unconstrained type-inst can be viewed as an arbitrarily constrained type-inst with a *true* constraint. For example, the following two type-insts are equivalent:

```
par int
(par int: i where true)
```

The Boolean expression associated with a variable declared to have an arbitrarily constrained type-inst is either tested at instance-time if the variable is a parameter or else generates a constraint if it is a decision variable.

An arbitrarily constrained type-inst is varified by varifying its base type-inst.

Records can use type-inst constraints like other type-insts, for example:

```
type Task = (record(int:     duration,
                    var int: start,
                    var int: finish
                   ): t where t.finish == t.start + t.duration);
```

Non-flat enums can also involve type-inst constraints; see Section 7.2 for details.

6 Expressions

6.1 Expressions Overview

Expressions represent values. They occur in various kinds of items. They have the following syntax:

⟨*expr*⟩ ::= ⟨*expr-atom*⟩ ⟨*expr-binop-tail*⟩
⟨*expr-atom*⟩ ::= ⟨*expr-atom-head*⟩ ⟨*expr-atom-tail*⟩ ⟨*annotations*⟩
⟨*expr-binop-tail*⟩ ::= [⟨*bin-op*⟩ ⟨*expr*⟩]
⟨*expr-atom-head*⟩ ::= ⟨*builtin-un-op*⟩ ⟨*expr-atom*⟩
| (⟨*expr*⟩)
| ⟨*ident-or-quoted-op*⟩
| _
| ⟨*bool-literal*⟩
| ⟨*int-literal*⟩
| ⟨*float-literal*⟩
| ⟨*string-literal*⟩
| ⟨*set-literal*⟩
| ⟨*set-comp*⟩
| ⟨*simple-array-literal*⟩
| ⟨*simple-array-literal-2d*⟩
| ⟨*indexed-array-literal*⟩
| ⟨*simple-array-comp*⟩
| ⟨*indexed-array-comp*⟩
| ⟨*tuple-literal*⟩
| ⟨*record-literal*⟩
| ⟨*enum-literal*⟩
| ⟨*ann-literal*⟩
| ⟨*if-then-else-expr*⟩
| ⟨*case-expr*⟩
| ⟨*let-expr*⟩
| ⟨*call-expr*⟩
| ⟨*gen-call-expr*⟩
⟨*expr-atom-tail*⟩ ::= ϵ
| ⟨*array-access-tail*⟩ ⟨*expr-atom-tail*⟩
| ⟨*tuple-access-tail*⟩ ⟨*expr-atom-tail*⟩
| ⟨*record-access-tail*⟩ ⟨*expr-atom-tail*⟩

Expressions can be composed from sub-expressions combined with operators. All operators (binary and unary) are described in Section 6.2, including the precedences of the binary operators. All unary operators bind more tightly than all binary operators.

Expressions can have one or more annotations. Annotations bind more tightly than unary and binary operator applications, but less tightly than access operations and non-operator applications. In some cases this binding is non-intuitive. For example, in the first three of the following lines, the annotation `a` binds to the identifier expression `x` rather

than the operator application. However, the fourth line features a non-operator application (due to the single quotes around the `not`) and so the annotation binds to the whole application.

```
not x::a;
not (x)::a;
not(x)::a;
'not'(x)::a;
```

Section 8 has more on annotations.

Expressions can be contained within parentheses.

The array, tuple and (Zinc-only) record and non-flat enum access operations all bind more tightly than unary and binary operators and annotations. The access operations can be chained and they associate to the left. For example, these two access operations are equivalent:

```
x = a[1].field.1;
x = ((a[1]).field).1;
```

They are described in more detail in Sections 6.3.13, 6.3.15, 6.3.17 and 6.3.19.

The remaining kinds of expression atoms (from ⟨*ident*⟩ to ⟨*gen-call-expr*⟩) are described in Sections 6.3.1–6.3.25.

We also distinguish syntactically valid integer expressions. This allows range types to be parsed correctly.

⟨*num-expr*⟩ ::= ⟨*num-expr-atom*⟩ ⟨*num-expr-binop-tail*⟩
⟨*num-expr-atom*⟩ ::= ⟨*num-expr-atom-head*⟩ ⟨*expr-atom-tail*⟩ ⟨*annotations*⟩
⟨*num-expr-binop-tail*⟩ ::= [⟨*num-bin-op*⟩ ⟨*num-expr*⟩]
⟨*num-expr-atom-head*⟩ ::= ⟨*builtin-num-un-op*⟩ ⟨*num-expr-atom*⟩
| (⟨*num-expr*⟩)
| ⟨*ident-or-quoted-op*⟩
| ⟨*int-literal*⟩
| ⟨*float-literal*⟩
| ⟨*if-then-else-expr*⟩
| ⟨*case-expr*⟩
| ⟨*let-expr*⟩
| ⟨*call-expr*⟩
| ⟨*gen-call-expr*⟩

6.2 Operators

Operators are functions that are distinguished by their syntax in one or two ways. First, some of them contain non-alphanumeric characters that normal functions do not (e.g. '`+`'). Second, their application is written in a manner different to normal functions.

We distinguish between binary operators, which can be applied in an infix manner (e.g. `3 + 4`), and unary operators, which can be applied in a prefix manner without parentheses (e.g. `not x`). We also distinguish between built-in operators and user-defined operators. The syntax is the following:

⟨*builtin-op*⟩ ::= ⟨*builtin-bin-op*⟩
| ⟨*builtin-un-op*⟩
⟨*bin-op*⟩ ::= ⟨*builtin-bin-op*⟩
| `‘`⟨*ident*⟩`‘`
⟨*builtin-bin-op*⟩ ::= `<->` | `->` | `<-` | `\/` | `xor` | `/\`
| `<` | `>` | `<=` | `>=` | `==` | `=` | `!=`
| `in` | `subset` | `superset` | `union` | `diff` | `symdiff`
| `..` | `intersect`| `++` | ⟨*builtin-num-bin-op*⟩
⟨*builtin-un-op*⟩ ::= `not` | ⟨*builtin-num-un-op*⟩

Again, we syntactically distinguish integer operators.

⟨*num-bin-op*⟩ ::= ⟨*builtin-num-bin-op*⟩
| `‘`⟨*ident*⟩`‘`
⟨*builtin-num-bin-op*⟩ ::= `+` | `-` | `*` | `/` | `div` | `mod`
⟨*builtin-num-un-op*⟩ ::= `+` | `-`

The binary operators are listed in Table 1.

A user-defined binary operator is created by backquoting a normal identifier, for example:

```
A `min2` B
```

This is a static error if the identifier is not the name of a binary function or predicate.

The unary operators are: `+`, `-` and `not`. User-defined unary operators are not possible.

As Section 3.4 explains, any built-in operator can be used as a normal function identifier by quoting it, e.g: `'+'(3, 4)` is equivalent to `3 + 4`.

The meaning of each operator is given in Section A.

Symbol(s)	Assoc.	Prec.
`<->`	left	1200
`->`	left	1100
`<-`	left	1100
`\/`	left	1000
`xor`	left	1000
`/\`	left	900
`<`	none	800
`>`	none	800
`<=`	none	800
`>=`	none	800
`==`, `=`	none	800
`!=`	none	800
`in`	none	700
`subset`	none	700
`superset`	none	700
`union`	left	600
`diff`	left	600
`symdiff`	left	600
`..`	none	500
`+`	left	400
`-`	left	400
`*`	left	300
`div`	left	300
`mod`	left	300
`/`	left	300
`intersect`	left	300
`++`	right	200
‘⟨*ident*⟩‘	left	100

Table 1: Binary infix operators. A lower precedence number means tighter binding; for example, `1+2*3` is parsed as `1+(2*3)` because ‘`*`’ binds tighter than ‘`+`’. Associativity indicates how chains of operators with equal precedences are handled; for example, `1+2+3` is parsed as `(1+2)+3` because ‘`+`’ is left-associative, `a++b++c` is parsed as `a++(b++c)` because ‘`++`’ is right-associative, and `1<x<2` is a syntax error because ‘`<`’ is non-associative.

6.3 Expression Atoms

6.3.1 Identifier Expressions and Quoted Operator Expressions

Identifier expressions and quoted operator expressions have the following syntax:

⟨*ident-or-quoted-op*⟩ ::= ⟨*ident*⟩
| ’⟨*builtin-op*⟩’

Examples of identifiers were given in Section 3.4. The following are examples of quoted operators:

```
’+’
’union’
```

In quoted operators, whitespace is not permitted between either quote and the operator. Section 6.2 lists Zinc’s built-in operators.

Syntactically, any identifier or quoted operator can serve as an expression. However, in a valid model any identifier or quoted operator serving as an expression must be one of:

- the name of a variable;
- an enum case name (if it has no arguments);
- the name of a predicate, function or operator (but only as the first argument to `foldl` or `foldr`, or as an argument in an annotation literal; see Section 5.8.2);
- the name of (or a synonym of) a flat enum;
- a synonym of a fixed set type defined using a range expression or a literal set expression.

Thus some types can serve as set values: enums, synonyms of enums, and synonyms of fixed set types defined using a range expression or a literal set expression.

6.3.2 Anonymous Decision Variables

There is a special identifier, '`_`', that represents an unfixed, anonymous decision variable. It can take on any type that can be a decision variable. It is particularly useful for initialising decision variables within compound types. For example, in the following array the first and third elements are fixed to 1 and 3 respectively and the second and fourth elements are unfixed:

```
array[1..4] of var int: xs = [1, _, 3, _];
```

Any expression that does not contain '`_`' and does not involve decision variables is fixed.

6.3.3 Boolean Literals

Boolean literals have this syntax:

⟨*bool-literal*⟩ ::= `false` | `true`

6.3.4 Integer and Float Literals

There are three forms of integer literals—decimal, hexadecimal, and octal—with these respective forms:

```
⟨int-literal⟩ ::= [0-9]+
                | 0x[0-9A-Fa-f]+
                | 0o[0-7]+
```

For example: `0`, `005`, `123`, `0x1b7`, `0o777`; but not `-1`.

Float literals have the following form:

```
⟨float-literal⟩ ::= [0-9]+[.][0-9]+
                  | [0-9]+[.][0-9]+[Ee][-+]?[0-9]+
                  | [0-9]+[Ee][-+]?[0-9]+
```

For example: `1.05`, `1.3e-5`, `1.3+e5`; but not `1.`, `.5`, `1.e5`, `.1e5`, `-1.0`, `-1E05`. A '`-`' symbol preceding an integer or float literal is parsed as a unary minus (regardless of intervening whitespace), not as part of the literal. This is because it is not possible in general to distinguish a '`-`' for a negative integer or float literal from a binary minus when lexing.

6.3.5 String Literals

String literals are written as in C:

⟨*string-literal*⟩ ::= `"[^"\n]*"`

This includes C-style escape sequences, such as '`\"`' for double quotes, '`\\`' for backslash, and '`\n`' for newline.

For example: `"Hello, world!\n"`.

String literals must fit on a single line. Long string literals can be split across multiple lines using string concatenation. For example:

```
string: s = "This is a string literal "
         ++ "split across two lines.";
```

6.3.6 Set Literals

Set literals have this syntax:

⟨*set-literal*⟩ ::= { [⟨*expr*⟩ , ...] }

For example:

```
{ 1, 3, 5 }
{ }
{ 1, _ }
```

The type-insts of all elements in a literal set must be the same, or coercible to the same type-inst (as in the last example above, where the fixed integer `1` will be coerced to a `var int`).

6.3.7 Set Comprehensions

Set comprehensions have this syntax:

⟨*set-comp*⟩ ::= { ⟨*expr*⟩ | ⟨*comp-tail*⟩ }
⟨*comp-tail*⟩ ::= ⟨*generator*⟩ , ... [where ⟨*expr*⟩]
⟨*generator*⟩ ::= ⟨*ident*⟩ , ... in ⟨*expr*⟩

For example (with the literal equivalent on the right):

```
{ 2*i | i in 1..5 }     % { 2, 4, 6, 8, 10 }
{  1  | i in 1..5 }     % { 1 }   (no duplicates in sets)
```

The expression before the '|' is the *head expression*. The expression after the `in` is a *generator expression*. Generators can be restricted by a *where-expression*. For example:

```
{ i | i in 1..10 where (i mod 2 == 0) }     % { 2, 4, 6, 8, 10 }
```

When multiple generators are present, the right-most generator acts as the inner-most one. For example:

```
{ 3*i+j | i in 0..2, j in {0, 1} }    % { 0, 1, 3, 4, 6, 7 }
```

The scope of local generator variables is given by the following rules:

- They are visible within the head expression (before the '|').
- They are visible within the where-expression.
- They are visible within generator expressions in any subsequent generators.

The last of these rules means that the following set comprehension is allowed:

```
{ i+j | i in 1..3, j in 1..i }  % { 1+1, 2+1, 2+2, 3+1, 3+2, 3+3 }
```

A generator expression must be an array; a fixed set can also be used, as it will be implicitly coerced to an array. ***Rationale.*** *For set comprehensions, set generators would suffice, but for array comprehensions, array generators are required for full expressivity (e.g. to provide control over the order of the elements in the resulting array). Set comprehensions have array generators for consistency with array comprehensions, which makes implementations simpler.*

The where-expression (if present) must be Boolean. Currently it must also be fixed; this restriction may be removed in the future. Only one where-expression per comprehension is allowed.

Rationale. *Allowing one where-expression per generator is another possibility, and one that could seemingly result in more efficient evaluation in some cases. For example, consider the following comprehension:*

```
[f(i, j) | i in A1, j in A2 where p(i) /\ q(i,j)]
```

If multiple where-expressions were allowed, this could be expressed more efficiently in the following manner, which avoids the fruitless "inner loop iterations" for each "outer loop iteration" that does not satisfy `p(i)`*:*

```
[f(i, j) | i in A1 where p(i), j in A2 where q(i,j)]
```

However, this efficiency can also be achieved with nested comprehensions:

```
[f(i, j) | i in [k | k in A1 where p(k)], j in A2 where q(i,j)]
```

Therefore, a single where-expression is all that is supported.

6.3.8 Simple Array Literals

Simple array literals have this syntax:

⟨*simple-array-literal*⟩ ::= [[⟨*expr*⟩ , ...]]

For example:

```
[1, 2, 3, 4]
[]
[1, _]
```

In a simple array literal all elements must have the same type-inst, or be coercible to the same type-inst (as in the last example above, where the fixed integer `1` will be coerced to a `var int`).

The indices of a simple array literal are implicitly 1..n, where n is the length of the literal.

6.3.9 Simple 2d Array Literals

Simple 2d array literals have this syntax:

⟨*simple-array-literal-2d*⟩ ::= `[|` [(⟨*expr*⟩ `,` ...) `|` ...] `|]`

For example:

```
[| 1, 2, 3
 | 4, 5, 6
 | 7, 8, 9 |]       % array[1..3, 1..3]
[| x, y, z |]       % array[1..1, 1..3]
[| 1 | _ | _ |]     % array[3..1, 1..1]
```

In a simple 2d array literal, every sub-array must have the same length.

In a simple 2d array literal all elements must have the same type-inst, or be coercible to the same type-inst (as in the last example above, where the fixed integer `1` will be coerced to a `var int`).

The indices of a simple 2d array literal are implicitly $(1,1)..(m,n)$, where m and n are determined by the shape of the literal.

6.3.10 Indexed Array Literals

Indexed array literals have this syntax:

⟨*indexed-array-literal*⟩ ::= `[` [⟨*index-expr*⟩ `,` ...] `]`
⟨*index-expr*⟩ ::= ⟨*expr*⟩ `:` ⟨*expr*⟩

For example:

```
[1:1, 2:4, 3:3, 4:10, 5:5]
```

The expressions before the colon are keys, those after are values.

In an indexed array literal all keys must have the same type-inst or be coercible to the same type-inst, and all values must have the same type-inst or be coercible to the same type-inst.

The keys need not be specified in order.

6.3.11 Simple Array Comprehensions

Simple array comprehensions have this syntax:

⟨*simple-array-comp*⟩ ::= `[` ⟨*expr*⟩ `|` ⟨*comp-tail*⟩ `]`

For example (with the literal equivalents on the right):

```
[2*i | i in 1..5]       % [2, 4, 6, 8, 10]
```

Simple array comprehensions have the same type and inst requirements as set comprehensions (see Section 6.3.7).

The indices of an evaluated simple array comprehension are implicitly $1..n$, where n is the length of the evaluated comprehension.

6.3.12 Indexed Array Comprehensions

Indexed array comprehensions have this syntax:

⟨*indexed-array-comp*⟩ ::= `[` ⟨*index-expr*⟩ `|` ⟨*comp-tail*⟩ `]`

For example (with the literal equivalent on the right):

```
[i:2*i | i in 1..4]             % [1:2, 2:4, 3:6, 4:8]
```

Simple array comprehensions have the same type and inst requirements as set comprehensions (see Section 6.3.7).

The keys need not be computed in order.

6.3.13 Array Access Expressions

Array elements are accessed using square brackets after an expression:

⟨*array-access-tail*⟩ ::= `[` ⟨*expr*⟩ `,` ... `]`

For example:

```
int: x = a1[1];
```

If all the indices used in an array access are fixed, the type-inst of the result is the same as the element type-inst. However, if any indices are not fixed, the type-inst of the result is the varified element type-inst. For example, if we have:

```
array[1..2] of int: a13 = [1, 2];
var int: i;
```

then the type-inst of `a13[i]` is `var int`. If the element type-inst is not varifiable, such an access causes a static error.

Syntactic sugar exists for accessing tuple-indexed arrays. For example, the second of the following two accesses is syntactic sugar for the first.

```
int: y = a13[(1, 2)];
int: y = a13[1, 2];
```

Array accesses can be chained to access arrays-of-arrays, and sub-arrays can be extracted from arrays-of-arrays, as the following two examples show.

```
array[1..2] of array[1..3] of int: a14;
int: y = a14[1][2];
array[1..2] of int: a12 = a14[1];
```

6.3.14 Tuple Literals

A tuple expression has this syntax:

⟨*tuple-literal*⟩ ::= `(` ⟨*expr*⟩ `,` ... `)`

For example:

```
(1, 2.0)
```

Tuple expressions must have at least two elements. A tuple expression with a single element is not actually a tuple expression, but rather just a normal expression with parentheses around it.

6.3.15 Tuple Access Expressions

Tuple fields are accessed by using a '.' and the field number after a tuple expression:

⟨*tuple-access-tail*⟩ ::= `.` ⟨*int-literal*⟩

For example, this expression:

```
(3, 4.0).1
```

has the value 3. Access of a non-existent field number results in a static error.

6.3.16 Record Literals

A record expression has this syntax:

⟨*record-literal*⟩ ::= `(` ⟨*named-expr*⟩ `,` ... `)`
⟨*named-expr*⟩ ::= ⟨*ident*⟩ `:` ⟨*expr*⟩

For example:

```
Task: t = ( duration:10, start:_, finish:_ );
```

6.3.17 Record Access Expressions

Record fields are accessed by using a '.' and the field name after a record expression:

⟨*record-access-tail*⟩ ::= `.` ⟨*ident*⟩

For example:

```
int: d = t.duration;
```

Access of a non-existent field name results in a static error.

6.3.18 Enum Literals

An enum expression has one of the following forms:

⟨*enum-literal*⟩ ::= ⟨*ident*⟩ `(` ⟨*named-expr*⟩ `,` ... `)`
| ⟨*ident*⟩ `(` ⟨*expr*⟩ `,` ... `)`
| ⟨*ident*⟩

Flat enum expressions obviously overlap completely with identifier expressions (see Section 6.3.1).

Here is an example of initialising parameters of non-flat enum types:

```
multi_point: P1 = int_point(ix:2, iy:3);
multi_point: P2 = float_point(2.3, 5.6);
```

6.3.19 Non-flat Enum Access Expressions

Enum fields are accessed like record fields, with a '.'. However, enum field access expressions are only allowed within case expressions (described in Section 6.3.22). This makes it harder to access a field that does not exist in a particular non-flat enum value. If such an access does occur, it is a run-time error.

6.3.20 Annotation Literals

Literals of the `ann` type have this syntax:

⟨*ann-literal*⟩ ::= ⟨*ident*⟩ [`(` ⟨*expr*⟩ `,` ... `)`]

For example:

```
foo
cons(1, cons(2, cons(3, nil)))
```

There is no way to inspect or deconstruct annotation literals in a Zinc model; they are intended to be inspected only by an implementation, e.g. to direct compilation.

6.3.21 If-then-else Expressions

Zinc provides if-then-else expressions, which provide selection from two alternatives based on a condition. They have this syntax:

⟨*if-then-else-expr*⟩ ::= `if` ⟨*expr*⟩ `then` ⟨*expr*⟩
(`elseif` ⟨*expr*⟩ `then` ⟨*expr*⟩)*
`else` ⟨*expr*⟩ `endif`

For example:

```
if x <= y then x else y endif
if x < 0 then -1 elseif x > 0 then 1 else 0 endif
```

The presence of the `endif` avoids possible ambiguity when an if-then-else expression is part of a larger expression.

The type-inst of the "if" expression must be `par bool`. In the future we may allow it to also be `var bool`. The "then" and "else" expressions must have the same type-inst, or be coercible to the same type-inst, which is also the type-inst of the whole expression.

Evaluation of if-then-else expressions is lazy—the condition is evaluated, and then only one of the "then" and "else" branches are evaluated, depending on whether the condition succeeded or failed.

6.3.22 Case Expressions

Zinc provides case expressions for handling the different cases in an enum.

Case expressions have this syntax:

⟨*case-expr*⟩ ::= `case` ⟨*expr*⟩ `{` ⟨*case-expr-case*⟩ `,` ... `}`
⟨*case-expr-case*⟩ ::= ⟨*ident*⟩ `-->` ⟨*expr*⟩

For example, we can use the `multi_point` record in Section 7.2 with the following code:

```
multi_point: r;
int: M = case r {
    int_point   --> r.ix,
    float_point --> 0,
};
```

The comma after the final case is optional.

The type-inst of the case selection expression must be a fixed enum.

The type-inst of every result expression must be the same, or be coercible to the same type-inst, which is also the type-inst of the whole expression.

It is a static error if any case name is not covered by the case statement.

Currently Zinc does not support pattern matching. This may be supported in the future.

6.3.23 Let Expressions

Let expressions provide a way of introducing local names for one or more expressions that can be used within another expression. They are particularly useful in user-defined operations.

Let expressions have this syntax:

⟨*let-expr*⟩ ::= `let` { ⟨*var-decl-item*⟩ `,` ... } `in` ⟨*expr*⟩

For example:

```
let {int: x = 3, int: y = 4} in x + y;
```

The scope of a let local variable covers:

- The initialisation expressions of any subsequent variables within the let expression (but not the variable's own initialisation expression).
- The expression after the `in`, which is parsed as greedily as possible.

A variable can only be declared once in a let expression.

Thus in the following examples the first is acceptable but the rest are not:

```
let {int: x = 3, int: y = x} in x + y;  % ok
let {int: y = x, int: x = 3} in x + y;  % x not visible in y's defn.
let {int: x = x} in x;                  % x not visible in x's defn.
let {int: x = 3, int: x = 4} in x;      % x declared twice
```

The type-inst expressions can include type-inst variables if the let is within a function or predicate body in which the same type-inst variables were present in the function or predicate signature.

The initialiser for a let local variable can be omitted only if the variable is a decision variable. For example:

```
let {var int: x} in ...;    % ok
let {    int: x} in ...;    % illegal
```

The type-inst of the entire let expression is the type-inst of the expression after the `in` keyword.

There is a complication involving let expressions in negative contexts. A let expression occurs in a negative context if it occurs in an expression of the form `not X`, `X <-> Y`, or in the sub-expression `X` in `X -> Y` or `Y <- X`.

If a let expression is used in a negative context, then any let-local decision variables must be defined only in terms of non-local variables and parameters. This is because local variables are implicitly existentially quantified, and if the let expression occurred in a negative context then the local variables would be effectively universally quantified which is not supported by Zinc.

6.3.24 Call Expressions

Call expressions are used to call predicates and functions.

Call expressions have this syntax:

⟨*call-expr*⟩ ::= ⟨*ident-or-quoted-op*⟩ [`(` ⟨*expr*⟩ `,` ... `)`]

For example (using an example function defined in Section 7.10.3):

```
x = min_of_two(3, 5);
```

The type-insts of the expressions passed as arguments must match the argument types of the called predicate/function. The return type of the predicate/function must also be appropriate for the calling context.

Note that a call to a function or predicate with no arguments is syntactically indistinguishable from the use of a variable, and so must be determined during type-inst checking.

Evaluation of the arguments in call expressions is strict—all arguments are evaluated before the call itself is evaluated. Note that this includes Boolean operations such as `/\`, `\/`, `->` and `<-` which could be lazy in one argument. The one exception is `assert`, which is lazy in its third argument (Section A.9).

Rationale. *Boolean operations are strict because: (a) this minimises exceptional cases; (b) in an expression like `A -> B`, where `A` is not fixed and `B` causes an abort, the appropriate behaviour is unclear if laziness is present; and (c) if a user needs laziness, an if-then-else can be used.*

The order of argument evaluation is not specified. ***Rationale.*** *Because Zinc is declarative, there is no obvious need to specify an evaluation order, and leaving it unspecified gives implementors some freedom.*

6.3.25 Generator Call Expressions

Zinc has special syntax for certain kinds of call expressions which makes models much more readable.

Generator call expressions have this syntax:

⟨*gen-call-expr*⟩ ::= ⟨*ident-or-quoted-op*⟩ (⟨*comp-tail*⟩) (⟨*expr*⟩)

A generator call expression `P(Gs)(E)` is equivalent to the call expression `P([E | Gs])`. For example, the expression:

```
forall(i,j in Domain where i<j)
    (noattack(i, j, queens[i], queens[j]));
```

(in a model specifying the N-queens problem) is equivalent to:

```
forall( [ noattack(i, j, queens[i], queens[j])
        | i,j in Domain where i<j ] );
```

The parentheses around the latter expression are mandatory; this avoids possible confusion when the generator call expression is part of a larger expression.

The identifier must be the name of a unary predicate or function that takes an array argument.

The generators and where-expression (if present) have the same requirements as those in array comprehensions (Section 6.3.11).

7 Items

This section describes the top-level program items.

7.1 Type-inst Synonym Items

Type-inst synonym items have this syntax:

⟨*type-inst-syn-item*⟩ ::= `type` ⟨*ident*⟩ ⟨*annotations*⟩ `=` ⟨*ti-expr*⟩

For example:

```
type MyInt     = int;
type FloatPlus = (float: x where x >= 0);
type Domain    = 1..n;
```

It is a type-inst error if a type-inst synonym is declared and/or defined more than once in a model.

Type-inst synonym items can be annotated. Section 8 has more details on annotations.

All type-inst synonyms must be defined at instance-time.

7.2 Enum Items

Enumerated type items have this syntax:

⟨*enum-item*⟩ ::= `enum` ⟨*ident*⟩ ⟨*annotations*⟩ [`=` ⟨*enum-cases*⟩]
⟨*enum-cases*⟩ ::= `{` ⟨*enum-case*⟩ `,` ... `}`
⟨*enum-case*⟩ ::= ⟨*ident*⟩ [`(` ⟨*ti-expr-and-id*⟩ `,` ... `)`]

An example of a flat enum:

```
enum country = {Australia, Canada, China, England, USA};
```

An example of a non-flat enum:

```
enum multi_point = {
    int_point(int: ix, int: iy),
    float_point(float: fx, float: fy)
};
```

Each alternative is called an *enum case*. The identifier used to name each case (e.g. `Australia` and `float_point`) is called the *enum case name*.

Enums can be constrained by using a type-inst synonym (Section 7.1) and a case expression (Section 6.3.22). For example:

```
type multi_point2 =
    (multi_point: p where case p { int_point   --> p.ix > p.iy,
                                   float_point --> p.fx > p.fy });
```

Because enum case names all reside in the top-level namespace (Section 4.3), case names in different enums must be distinct. As for field names in non-flat enums, all field names in a single enum must be distinct to avoid possible ambiguities.

An enum can be declared but not defined, in which case it must be defined elsewhere. For non-flat enums, "elsewhere" must be within the model. For flat enums, "elsewhere" must be within the model, or in a data file. For example, a model file could contain this:

```
enum Workers;
enum Shifts;
```

and the data file could contain this:

```
enum Workers = { welder, driller, stamper };
enum Shifts  = { idle, day, night };
```

Sometimes it is useful to be able to refer to one of the enum case names within the model. This can be achieved by using a variable. The model would read:

```
enum Shifts;
Shifts idle;            % Variable representing the idle constant.
```

and the data file:

```
enum Shifts = { idle_const, day, night };
idle = idle_const;      % Assignment to the variable.
```

Although the constant `idle_const` cannot be mentioned in the model, the variable `idle` can be.

All enums must be defined at instance-time.

Enum items can be annotated. Section 8 has more details on annotations.

7.3 Include Items

Include items allow a model to be split across multiple files. They have this syntax:

⟨*include-item*⟩ ::= `include` ⟨*string-literal*⟩

For example:

```
include "foo.zinc";
```

includes the file `foo.zinc`.

Include items are particularly useful for accessing libraries or breaking up large models into small pieces. They are not, as Section 4.2 explains, used for specifying data files.

If the given name is not a complete path then the file is searched for in an implementation-defined set of directories. The search directories must be able to be altered with a command line option.

7.4 Variable Declaration Items

Variable declarations have this syntax:

⟨*var-decl-item*⟩ ::= ⟨*ti-expr-and-id*⟩ ⟨*annotations*⟩ [`=` ⟨*expr*⟩]

For example:

```
int: A = 10;
```

It is a type-inst error if a variable is declared and/or defined more than once in a model.

A variable whose declaration does not include an assignment can be initialised by a separate assignment item (Section 7.5). For example, the above item can be separated into the following two items:

```
int: A;
...
A = 10;
```

All variables that contain a parameter component must be defined at instance-time.

Variables can have one or more annotations. Section 8 has more on annotations.

7.5 Assignment Items

Assignments have this syntax:

⟨*assign-item*⟩ ::= ⟨*ident*⟩ `=` ⟨*expr*⟩

For example:

```
A = 10;
```

7.6 Constraint Items

Constraint items form the heart of a model. Any solutions found for a model will satisfy all of its constraints.

Constraint items have this syntax:

⟨*constraint-item*⟩ ::= `constraint` ⟨*expr*⟩

For example:

```
constraint a*x < b;
```

The expression in a constraint item must have type-inst `par bool` or `var bool`; note however that constraints with fixed expressions are not very useful.

7.7 Solve Items

Every model must have exactly one solve item. Solve items have the following syntax:

⟨*solve-item*⟩ ::= `solve` ⟨*annotations*⟩ `satisfy`
| `solve` ⟨*annotations*⟩ `minimize` ⟨*expr*⟩
| `solve` ⟨*annotations*⟩ `maximize` ⟨*expr*⟩

Example solve items:

```
solve satisfy;
solve maximize a*x + y - 3*z;
```

The solve item determines whether the model represents a constraint satisfaction problem or an optimisation problem. In the latter case the given expression is the one to be minimized/maximized.

The expression in a minimize/maximize solve item can have any type-inst. ***Rationale.*** *This is possible because all type-insts have a defined order.* Note that having an expression with a fixed type-inst in a solve item is not very useful as it means that the model requires no constraint solving.

Solve items can be annotated. Section 8 has more details on annotations.

7.8 Output Items

Output items are used to present the results of a model execution. They have the following syntax:

⟨*output-item*⟩ ::= `output` ⟨*expr*⟩

For example:

```
output ["The value of x is ", show(x), "!\n"];
```

The expression must have type-inst `array[int] of par string`. It can be composed using the built-in operator `++`, the built-in function `show`, and the built-in function `show_cond` (Section A).

The output is determined by concatenating the individual elements of the array.

Each model can have at most one output item. If a solution is found and an output item is present, it is used to determine the string to be printed. If a solution is found and no output item is present, the implementation should print all the global variables and their values in a readable format. If no solution is found, the implementation should print "No solution found"; it may also print some extra information about the cause of the failure, such as which constraints were violated.

7.9 Annotation Items

Annotation items are used to augment the `ann` type. They have the following syntax:

⟨*annotation-item*⟩ ::= `annotation` ⟨*ident*⟩ ⟨*params*⟩

For example:

```
annotation solver(SolverKind: kind);
```

It is a type-inst error if an annotation is declared and/or defined more than once in a model.

The use of annotations is described in Section 8.

7.10 User-defined Operations

Zinc models can contain user-defined operations. They have this syntax:

⟨*predicate-item*⟩ ::= `predicate` ⟨*operation-item-tail*⟩
⟨*test-item*⟩ ::= `test` ⟨*operation-item-tail*⟩
⟨*function-item*⟩ ::= `function` ⟨*ti-expr*⟩ `:` ⟨*operation-item-tail*⟩
⟨*operation-item-tail*⟩ ::= ⟨*ident*⟩ ⟨*params*⟩ ⟨*annotations*⟩ [`=` ⟨*expr*⟩]
⟨*params*⟩ ::= [`(` ⟨*ti-expr-and-id*⟩ `,` ... `)`]

The type-inst expressions can include type-inst variables in the function and predicate declaration.
For example, predicate `even` checks that its argument is an even number.

```
predicate even(int: x) =
    x mod 2 == 0;
```

Predicate `serial` constrains the resistor `z` to be equivalent to connecting the two resistors `x` and `y` in series (the fields `r` and `i` represent resistance and current respectively).

```
type Resistor = record(int: r, int: i);
predicate serial(Resistor: x, Resistor: y, Resistor: z) =
    z.r == x.r + y.r /\
    z.i == x.i       /\
    z.i == y.i;
```

A predicate supported natively by the target solver can be declared as follows:

```
predicate all_different(array [int] of var int: xs);
```

Predicate declarations in MiniZinc are restricted to using FlatZinc types (for instance, multi-dimensional and non-1-based arrays are forbidden).

Declarations for user-defined operations can be annotated. Section 8 has more details on annotations.

7.10.1 Basic Properties

The term "predicate" is generally used to refer to both test items and predicate items. When the two kinds must be distinguished, the terms "test item" and "predicate item" can be used.

The return type-inst of a test item is implicitly `par bool`. The return type-inst of a predicate item is implicitly `var bool`.

Predicates and functions are not allowed to be recursive. ***Rationale.*** *This ensures that the satisfiability of models is decidable.*

Predicates and functions introduce their own local names, being those of the formal arguments. The scope of these names covers the predicate/function body. Argument names cannot be repeated within a predicate/function declaration.

Zinc is mostly a first-order language, so operations cannot, in general, be used as values. The only exception to this is that they may be given as the first argument to `foldl` or `foldr`, and as arguments in annotation literals (see Section 5.8.2).

7.10.2 Ad-hoc polymorphism

Zinc supports ad-hoc polymorphism via overloading. Functions and predicates (both built-in and user-defined) can be overloaded. A name can be overloaded as both a function and a predicate.

It is a type-inst error if a single version of an overloaded operation with a particular type-inst signature is declared and/or defined more than once in a model. For example:

```
predicate p(1..5: x);
predicate p(1..5: x) = true;        % error:  repeated declaration
```

The combination of overloading and coercions can cause problems. Two overloadings of an operation are said to "overlap" if they could match the same arguments. For example, the following overloadings of `p` overlap, as they both match the call `p(3)`.

```
predicate p(par int: x);
predicate p(var int: x);
```

However, the following two predicates do not overlap because they cannot match the same argument:

```
predicate q(int:        x);
predicate q(set of int: x);
```

We avoid two potential overloading problems by placing some restrictions on overlapping overloadings of operations.

1. The first problem is ambiguity. Different placement of coercions in operation arguments may allow different choices for the overloaded function. For instance, if a Zinc function `f` is overloaded like this:

```
function int: f(int: x, float: y) = 0;
function int: f(float: x, int: y) = 1;
```

   then `f(3,3)` could be either 0 or 1 depending on coercion/overloading choices.

   To avoid this problem, any overlapping overloadings of an operation must be semantically equivalent with respect to coercion. For example, the two overloadings of the predicate `p` above must have bodies that are semantically equivalent with respect to overloading.

   Currently, this requirement is not checked and the modeller must satisfy it manually. In the future, we may require the sharing of bodies among different versions of overloaded operations, which would provide automatic satisfaction of this requirement.

2. The second problem is that certain combinations of overloadings could require a Zinc implementation to perform combinatorial search in order to explore different choices of coercions and overloading. For example, if function `g` is overloaded like this:

```
function tuple(float,int): g(tuple(int,float): t) = (t.2, t.1);
function tuple(int,float): g(tuple(float,int): t) = (t.2, t.1);
```

   then how the overloading of `g( (3,3) )` is resolved depends upon its context:

```
tuple(float,int): s = g( (3,3) );
tuple(float,int): t = g( g( (3,3) ) );
```

   In the definition of `s` the first overloaded definition must be used while in the definition of `t` the second must be used.

   To avoid this problem, all overlapping overloadings of an operation must be closed under intersection of their input type-insts. That is, if overloaded versions have input type-inst $(S_1, ..., S_n)$ and $(T_1, ..., T_n)$ then there must be another overloaded version with input type-inst $(R_1, ..., R_n)$ where each R_i is the greatest lower bound (*glb*) of S_i and T_i.

   Also, all overlapping overloadings of an operation must be monotonic. That is, if there are overloaded versions with input type-insts $(S_1, ..., S_n)$ and $(T_1, ..., T_n)$ and output type-inst S and T, respectively, then $S_i \preceq T_i$ for all i, implies $S \preceq T$. At call sites, the matching overloading that is lowest on the type-inst lattice is always used.

   For `g` above, the type-inst intersection (or *glb*) of `tuple(float,int)` and `tuple(float,int)` is `tuple(int,int)`. Thus, the overloaded versions are not closed under intersection and the user needs to provide another overloading for `g` with input type-inst `tuple(int,int)`. The natural definition is:

```
function tuple(int,int): g(tuple(int,int): t) = (t.2, t.1);
```

   Once `g` has been augmented with the third overloading, it satisfies the monotonicity requirement because the output type-inst of the third overloading is `tuple(int,int)` which is less than the output type-inst of the original overloadings.

   Monotonicity and closure under type-inst conjunction ensure that whenever an overloaded function or predicate is reached during type-inst checking, there is always a unique and safe "minimal" version to choose, and so the complexity of type-inst checking remains linear. Thus in our example `g((3,3))` is always resolved by choosing the new overloaded definition.

7.10.3 Parametric Polymorphism

Zinc supports parametric polymorphic of functions and predicates via type-inst variables.

For example, function `min_of_two` takes two parameters and gives their minimum.

```
function $T:min_of_two($T: x, $T: y) =
    if x <= y then x else y endif;
```

This function is possible because every type has a built-in ordering.

Section 5.8.1 explained that type-inst variables can have no prefix (or, equivalently, a `par` prefix) or an `any` prefix. The prefixes are necessary for precise type-inst signatures of some user-defined operations. For example, consider the following two definitions of a function `between`:

```
par bool: function between(par $T: x, par $T: y, par $T: z) =
    (x <= y /\ y <= z) \/ (z <= y /\ y <= x);
var bool: function between(any $T: x, any $T: y, any $T: z) =
    (x <= y /\ y <= z) \/ (z <= y /\ y <= x);
```

The first version has a more precise return type-inst. The `par` and `any` prefixes are needed to express the difference between these two versions.

Note that although `par $T` (and also `$T`) only *matches* fixed type-insts, it does not mean that the type-inst variable `$T` must be *bound* to a fixed type-inst. For example, with these variables and predicate:

```
par int: pi;
var int: vi;
predicate p(par $T: x, any $T: y);
```

the first two of the following are acceptable, but the last two are errors:

```
constraint p(pi, pi);     % ok:  $T bound to 'par int'
constraint p(pi, vi);     % ok:  $T bound to 'var int'
constraint p(vi, pi);     % error
constraint p(vi, vi);     % error
```

7.10.4 Local Variables

Local variables in operation bodies are introduced using let expressions. For example, the predicate `have_common_divisor` takes two integer values and checks whether they have a common divisor greater than one:

```
predicate have_common_divisor(int: A, int: B) =
    let {
        var 2..min2(A,B): C
    } in
        A mod C == 0 /\
        B mod C == 0;
```

However, as Section 6.3.23 explained, because `C` is not defined, this predicate cannot be called in a negative context. The following is a version that could be called in a negative context:

```
predicate have_common_divisor(int: A, int: B) =
    exists(C in 2..min2(A,B))
        (A mod C == 0 /\ B mod C == 0);
```

8 Annotations

Annotations—values of the `ann` type—allow a modeller to specify non-declarative and solver-specific information that is beyond the core language. Annotations do not change the meaning of a model, however, only how it is solved.

Annotations can be attached to variables (on their declarations), expresssions, type-inst synonyms, enum items, solve itmes and on user defined operations. They have the following syntax:

⟨*annotations*⟩ ::= (`::` ⟨*annotation*⟩)*
⟨*annotation*⟩ ::= ⟨*expr-atom-head*⟩ ⟨*expr-atom-tail*⟩

For example:

```
enum c :: foo = {r, g, b};
type size :: foo = int;
int: x::foo;
x = (3 + 4)::bar("a", 9)::baz("b");
solve :: blah(4)
    minimize x;
```

The types of the argument expressions must match the argument types of the declared annotation. Unlike user-defined predicates and functions, annotations cannot be overloaded. ***Rationale.*** *There is no particular strong reason for this, it just seemed to make things simpler.*

Annotation signatures can contain type-inst variables.

The order and nesting of annotations do not matter. For the expression case it can be helpful to view the annotation connector `'::'` as an overloaded operator:

```
ann: '::'(any $T: e, ann: a);       % associative
ann: '::'(ann:    a, ann: b);       % associative + commutative
```

Both operators are associative, the second is commutative. This means that the following expressions are all equivalent:

```
e :: a :: b
e :: b :: a
(e :: a) :: b
(e :: b) :: a
e :: (a :: b)
e :: (b :: a)
```

This property also applies to annotations on solve items and variable declaration items. ***Rationale.*** *This property make things simple, as it allows all nested combinations of annotations to be treated as if they are flat, thus avoiding the need to determine what is the meaning of an annotated annotation. It also makes the Zinc abstract syntax tree simpler by avoiding the need to represent nesting.*

Zinc's built-in annotations are listed in Appendix C. Moreover, an implementation is likely to define a number of its own annotations for a variety of purposes.

9 Partiality

The presence of constrained type-insts in Zinc means that various operations are potentially *partial*, i.e. not clearly defined for all possible inputs. For example, what happens if a function expecting a positive argument is passed a negative argument? What happens if a variable is assigned a value that does not satisfy its type-inst constraints? What happens if an array index is out of bounds? This section describes what happens in all these cases.

In general, cases involving fixed values that do not satisfy constraints lead to run-time aborts. ***Rationale.*** *Our experience shows that if a fixed value fails a constraint, it is almost certainly due to a programming error. Furthermore, these cases are easy for an implementation to check.*

But cases involving unfixed values vary, as we will see. ***Rationale.*** *The best thing to do for unfixed values varies from case to case. Also, it is difficult to check constraints on unfixed values, particularly because during search a decision variable might become fixed and then backtracking will cause this value to be reverted, in which case aborting is a bad idea.*

9.1 Partial Assignments

The first operation involving partiality is assignment. There are four distinct cases for assignments.

- A value assigned to a fixed, constrained global variable is checked at run-time; if the assigned value does not satisfy its constraints, it is a run-time error. In other words, this:

  ```
  1..5: x = 3;
  ```

  is equivalent to this:

  ```
  int: x = 3;
  constraint assert(x in 1..5,
                    "assignment to global parameter 'x' failed")
  ```

- A value assigned to an unfixed, constrained global variable makes the assignment act like a constraint; if the assigned value does not satisfy the variable's constraints, it causes a run-time model failure. In other words, this:

  ```
  var 1..5: x = 3;
  ```

  is equivalent to this:

  ```
  var int: x = 3;
  constraint x in 1..5;
  ```

  Rationale. *This behaviour is easy to understand and easy to implement.*

- A value assigned to a fixed, constrained let-local variable is checked at run-time; if the assigned value does not satisfy its constraints, it is a run-time error. In other words, this:

  ```
  let { 1..5: x = 3 } in x+1
  ```

  is equivalent to this:

  ```
  let { int: x = 3 } in
      assert(x in 1..5,
             "assignment to let parameter 'x' failed", x+1)
  ```

- A value assigned to an unfixed, constrained let-local variable makes the assignment act like a constraint; if the assigned value does not statically match the variable's constraint at run-time it fails, and the failure "bubbles up" to the nearest enclosing Boolean scope, where it is interpreted as `false`.

  Rationale. *This behaviour is consistent with assignments to global variables.*

Note that in cases where a value is partly fixed and partly unfixed, e.g. some tuples, the different elements are checked according to the different cases, and fixed elements are checked before unfixed elements. For example:

```
tuple(var 1..5, par 1..5): t = (6, 6);
```

This causes a run-time abort, because the second, fixed element is checked before the first, unfixed element. This ordering is true for the cases in the following sections as well. ***Rationale.*** *This ensures that failures cannot mask aborts, which seems desirable.*

9.2 Partial Predicate/Function and Annotation Arguments

The second kind of operation involving partiality is calls and annotations. The behaviour for these operations is simple: constraints on arguments are ignored.

Rationale. *This is easy to implement and easy to understand. It is also justifiable in the sense that predicate/function/annotation arguments are values that are passed in from elsewhere; if those values are to be constrained, that could be done earlier. (In comparison, when a variable with a constrained type-inst is declared, any assigned value must clearly respect that constraint.)*

It is possible in the future that this behaviour may be changed to be more restrictive, like assignments: fixed arguments that fail their constraints will cause aborts, and unfixed arguments that fail their constraints will cause failure, which bubbles up to the nearest enclosing Boolean scope.

9.3 Partial Array Accesses

The third kind of operation involving partiality is array access. There are two distinct cases.

- A fixed value used as an array index is checked at run-time; if the index value is not in the index set of the array, it is a run-time error.
- An unfixed value used as an array index makes the access act like a constraint; if the access fails at run-time, the failure "bubbles up" to the nearest enclosing Boolean scope, where it is interpreted as `false`. For example:

  ```
  array[1..3] of int: a = [1,2,3];
  var int: i;
  constraint (a[i] + 3) > 10 \/ i == 99;
  ```

  Here the array access fails, so the failure bubbles up to the disjunction, and `i` is constrained to be 99. ***Rationale.*** *Unlike predicate/function calls, modellers in practice sometimes do use array accesses that can fail. In such cases, the "bubbling up" behaviour is a reasonable one.*

A Built-in Operations

This appendix lists built-in operators, functions and predicates. They may be implemented as true built-ins, or in libraries that are automatically imported for all models. Many of them are overloaded.

Operator names are written within single quotes when used in type signatures, e.g. `bool: '\/'(bool, bool)`.

We use the syntax `TI: f(TI1,...,TIn)` to represent an operation named `f` that takes arguments with type-insts `TI,...,TIn` and returns a value with type-inst `TI`. This is slightly more compact than the usual Zinc syntax, in that it omits argument names.

A.1 Comparison Operations

Less than. Other comparisons are similar: greater than (`>`), less than or equal (`<=`), greater than or equal (`>=`), equality (`==`, `=`), and disequality (`!=`).

```
     bool: '<'(    $T,     $T)
 var bool: '<'(any $T, any $T)
```

A.2 Arithmetic Operations

Addition. Other numeric operations are similar: subtraction (`-`), and multiplication (`*`).

```
     int:   '+'(    int,       int)
 var int:   '+'(var int,   var int)
     float: '+'(    float,     float)
 var float: '+'(var float, var float)
```

Unary minus. Unary plus (`+`) is similar.

```
     int:   '-'(    int)
 var int:   '-'(var int)
     float: '-'(    float)
 var float: '-'(var float)
```

Integer and floating-point division and modulo.

```
     int:   'div'(    int,       int)
 var int:   'div'(var int,   var int)
     int:   'mod'(    int,       int)
 var int:   'mod'(var int,   var int)
     float: '/'  (    float,     float)
 var float: '/'  (var float, var float)
```

The result of the modulo operation, if non-zero, always has the same sign as its second operand. The integer division and modulo operations are connected by the following identity:

```
x == (x div y) * y + (x mod y)
```

Some illustrative examples:

```
 7 div  4 =  1        7 mod  4 =  3
-7 div  4 = -2       -7 mod  4 =  1
 7 div -4 = -2        7 mod -4 = -1
-7 div -4 =  1       -7 mod -4 = -3
```

Sum multiple numbers. Product (`product`) is similar. Note that the sum of an empty array is 0, and the product of an empty array is 1.

```
     int:   sum(array[$T]  of     int  )
 var int:   sum(array[$T]  of var int  )
     float: sum(array[$T]  of     float)
 var float: sum(array[$T]  of var float)
```

Minimum of two values; maximum (`max`) is similar.

```
any $T:    min(any $T,    any $T )
```

Minimum of an array of fixed values; maximum (`max`) is similar.

```
any $U:    min(array[$T]  of any $U)
```

Minimum of a fixed set; maximum (`max`) is similar.

```
  $T:    min(set of $T)
```

Absolute value of a number.

```
     int:   abs(    int)
 var int:   abs(var int)
     float: abs(    float)
 var float: abs(var float)
```

Square root of a float. Aborts if argument is negative.

```
     float: sqrt(    float)
 var float: sqrt(var float)
```

Power operator. E.g. `pow(2, 5)` gives 32.

```
   int: pow(int,       int)
 float: pow(float,     float)
```

Natural exponent.

```
     float: exp(float)
 var float: exp(var float)
```

Natural logarithm. Logarithm to base 10 (`log10`) and logarithm to base 2 (`log2`) are similar.

```
     float: ln(float)
 var float: ln(var float)
```

General logarithm; the first argument is the base.

```
 float: log(float, float)
```

Sine. Cosine (`cos`), tangent (`tan`), inverse sine (`asin`), inverse cosine (`acos`), inverse tangent (`atan`), hyperbolic sine (`sinh`), hyperbolic cosine (`cosh`), hyperbolic tangent (`tanh`), inverse hyperbolic sine (`asinh`), inverse hyperbolic cosine (`acosh`) and inverse hyperbolic tangent (`atanh`) are similar.

```
     float: sin(float)
 var float: sin(var float)
```

A.3 Logical Operations

Conjunction. Other logical operations are similar: disjunction (`\/`) reverse implication (`<-`), forward implication (`->`), bi-implication (`<->`), exclusive disjunction (`xor`), logical negation (`not`).

Note that the implication operators are not written using `=>`, `<=` and `<=>` as is the case in some languages. This allows `<=` to instead represent "less than or equal".

```
     bool: '/\'(    bool,     bool)
 var bool: '/\'(var bool, var bool)
```

Universal quantification. Existential quantification (`exists`) is similar. Note that, when applied to an empty list, `forall` returns `true`, and `exists` returns `false`.

```
     bool: forall(array[$T]  of     bool)
 var bool: forall(array[$T]  of var bool)
```

A.4 Set Operations

Set membership.

```
     bool: 'in'(    $T,       set of $T )
 var bool: 'in'(any $T,   var set of $T )
```

Non-strict subset. Non-strict superset (`superset`) is similar.

```
     bool: 'subset'(    set of $T,     set of $T)
 var bool: 'subset'(var set of $T, var set of $T)
```

Set union. Other set operations are similar: intersection (`intersect`), difference (`diff`), symmetric difference (`symdiff`).

```
     set of $T: 'union'(    set of $T,     set of $T )
 var set of $T: 'union'(var set of $T, var set of $T )
```

Set range. If the first argument is larger than the second (e.g. `1..0`), it returns the empty set.

```
set of int: '..'(int, int)
```

Cardinality of a set.

```
    int: card(    set of $T)
var int: card(var set of $T)
```

Union of an array of sets. Intersection of multiple sets (`array_intersect`) is similar.

```
    set of $U:    array_union(array[$T]  of     set of $U)
var set of $U:    array_union(array[$T]  of var set of $U)
```

Power set.

```
set of set of $T: powerset(set of $T)
```

Cartesian product of sets. This list is only partial, it extends in the obvious way, for greater numbers of sets.

```
set of tuple($T1, $T2):      cartesian_product(set of $T1, set of $T2)
set of tuple($T1, $T2, $T3): cartesian_product(set of $T1, set of $T2,
                                               set of $T3)
...
```

A.5 Array Operations

Length of an array.

```
int: length(array[$T] of any $U)
```

List concatenation. Returns the list (integer-indexed array) containing all elements of the first argument followed by all elements of the second argument, with elements occurring in the same order as in the arguments. The resulting indices are in the range `1..n`, where `n` is the sum of the lengths of the arguments. ***Rationale.*** *This allows list-like arrays to be concatenated naturally and avoids problems with overlapping indices. The resulting indices are consistent with those of implicitly indexed array literals.* Note that `'++'` also performs string concatenation.

```
array[int] of any $T: '++'(array[int] of any $T, array[int] of any $T)
```

Array concatenation. Does not change any indices. If any index is repeated in the result, it is a run-time error. Note that it may result in an interleaving of the elements, e.g. the concatenation of `[1:1, 3:3]` and `[2:2, 4:4]` is `[1:1, 2:2, 3:3, 4:4]`.

```
array[$T] of any $U: concat(array[$T] of any $U, array[$T] of any $U)
```

Index sets of arrays. If the argument is a literal, returns `1..n` where `n` is the (sub-)array length. Otherwise, returns the declared or inferred index set. This list is only partial, it extends in the obvious way, for arrays of higher dimensions.

```
set of $T:  index_set     (array[$T]      of any $V)
set of $T:  index_set_1of2(array[$T, $U]  of any $V)
set of $U:  index_set_2of2(array[$T, $U]  of any $V)
...
```

Get the first and last elements of an array, and the tail of an array (i.e. all elements except the first). All of them abort if the array is empty.

```
any $U:              head(array[$T] of any $U)
any $U:              last(array[$T] of any $U)
array[$T] of any $U: tail(array[$T] of any $U)
```

Replace the indices of the array given by the last argument with the cartesian product of the sets given by the previous arguments. Similar versions exist for arrays up to 6 dimensions.

```
array[$T1] of any $V: array1d(set of $T1, array[$U] of any $V)
array[$T1,$T2] of any $V:
    array2d(set of $T1, set of $T2, array[$U] of any $V)
array[$T1,$T2,$T3] of any $V:
    array3d(set of $T1, set of $T2, set of $T3, array[$U] of any $V)
```

Condenses an array-of-arrays into an array, by folding `concat` over the array-of-arrays. It is a run-time error if any of the indices are repeated in the result.

```
array[$U] of $V: condense(array[$T] of array[$U] of $V)
```

Condenses an array-of-arrays into an array, by folding `++` over the array-of-arrays. This means the array indices are ordered contiguously after the final element of the first array in the array-of-arrays.

```
array[int] of $T: condense_int_index(array[$U] of array[int] of $T)
```

A.6 Coercion Operations

Round a float towards $+\infty$, $-\infty$, and the nearest integer, respectively.

```
int: ceil (float)
int: floor(float)
int: round(float)
```

Explicit casts from one type-inst to another.

```
    int:           bool2int(    bool)
var int:           bool2int(var bool)
    float:         int2float(    int)
var float:         int2float(var int)
array[int] of $T: set2array(set of $T)
```

A.7 String Operations

To-string conversion. Converts any value to a string for output purposes. The exact form of the resulting string is implementation-dependent.

```
string: show(any $T)
```

Conditional to-string conversion. If the first argument is not fixed, it aborts; if it is fixed to `true`, the second argument is converted to a string; if it is fixed to `false` the third argument is converted to a string. The exact form of the resulting string is implementation-dependent, but same as that produced by `show`.

```
string: show_cond(var bool, any $T, any $U)
```

To-string conversion, for floats. The exact form of the resulting string is implementation-dependent, like `show`, but the integer argument should preferably be used to dictate the upper limit on the number of decimal places that are shown.

```
string: show_float(var float, int)
```

String concatenation. Note that `'++'` also performs array concatenation.

```
string: '++'(string, string)
```

A.8 Bound and Domain Operations

Note that these operations can produce different results depending on when they are evaluated, and what form the argument takes. For example, consider the numeric lower bound operation.

- If the argument is a fixed expression, the result is the argument's value.
- If the argument is a decision variable name, then the result depends on the circumstance.
  - If the variable has a current lower bound, the result is that lower bound. The current lower bound may be from the variable's declaration (e.g. if evaluated at instance-time), or higher than that due to constraint solving (e.g. if evaluated at run-time), or from an implementation-defined lower bound (e.g. if it was declared with no lower bound, but the implementation imposes a lowest possible bound).
  - If the variable has no current lower bound (e.g. because the variable was declared with no lower bound and the implementation is able to represent $-\infty$), the operation aborts.
- If the argument is any other kind of unfixed expression, the operation aborts.

All of the following operations operate in this basic manner.

In Zinc, but not MiniZinc, the bound operations are polymorphic.

```
$T:   lb(any $T)
$T:   ub(any $T)
```

For numeric types, they return the lower/upper bound, i.e. the lowest/highest value the number can currently take.

For set types, they return the set lower/upper bound, i.e. the intersection/union of all current possible values of the set.

For array types, they return an array containing the lower/upper bound of each array element.

MiniZinc numeric current lower/upper bound, i.e. the lowest/highest value the number can currently take.

```
int:   lb(var int)
float: lb(var float)
int:   ub(var int)
float: ub(var float)
```

MiniZinc set lower/upper bound, i.e. the intersection/union of all current possible values of the set.

```
set of int: lb(var set of int)
set of int: ub(var set of int)
```

For example, the lower bound of a set variable that could equal {1,3,5} or {1,2,4} is {1}, and the upper bound is {1,2,3,4,5}.

MiniZinc array lower/upper bound, i.e. the lowest/highest of all the current lower/upper bounds of all the elements in the array (like finding the lower/upper bound of every element in the list and then folding `min`/`max` over them; or for the set version, like folding `intersection`/`union` over them).

```
int:        lb(array[$T] of var int)
float:      lb(array[$T] of var float)
set of int: lb(array[$T] of var set of int)
int:        ub(array[$T] of var int)
float:      ub(array[$T] of var float)
set of int: ub(array[$T] of var set of int)
```

The above MiniZinc bound operations on arrays are deprecated. They will be removed in a future version of MiniZinc. Models that require the existing behaviour should instead use the following (equivalent) operations:

```
int:        lb_array(array[int] of var int)
float:      lb_array(array[int] of var float)
set of int: lb_array(array[int] of var set of int)
int:        ub_array(array[int] of var int)
float:      ub_array(array[int] of var float)
set of int: ub_array(array[int] of var set of int)
```

In Zinc, but not MiniZinc, the domain operation is polymorphic.

```
set of $T: dom(any $T)
```

For integer values, it returns the current possible values for the integer.

For array value, it returns an array containing the set of of possible values for each element.

MiniZinc integer domain, i.e. the set of current possible values for the integer.

```
set of int: dom(var int)
```

MiniZinc integer array domain, i.e. the union of the current domains of all elements in the array.

```
set of int: dom(array[$T] of var int)
```

The above operation is deprecated (as with the array bound operations). Models that require the existing behaviour should instead use the following (equivalent) operation:

```
set of int: dom_array(array[int] of var int)
```

Domain size, for integers and arrays; `dom_size(x)` is equivalent to `card(dom(x))`, but is likely to be much faster.

```
int: dom_size(var int)
int: dom_size(array[$T] of var int)
```

A.9 Other Operations

Check a Boolean expression is true, and abort if not, printing the second argument as the error message. The first one returns the third argument, and is particularly useful for sanity-checking arguments to predicates and functions; importantly, its third argument is lazy, i.e. it is only evaluated if the condition succeeds. The second one returns `true` and is useful for global sanity-checks (e.g. of instance data) in constraint items.

```
any $T:   assert(bool: c, string: s, any $T: val)
par bool: assert(bool: c, string: s)
```

Abort evaluation, printing the given string.

```
any $T: abort(string: s)
```

Check if the argument's value is fixed at this point in evaluation. If not, abort; if so, return its value. This is most useful in output items when decision variables should be fixed—it allows them to be used in places where a fixed value is needed, such as if-then-else conditions.

```
$T: fix(any $T)
```

Extract the first or second element from a two-element tuple. `fst` and `snd` are synonyms.

```
any $T: first (tuple(any $T, any $U))
any $U: second(tuple(any $T, any $U))
```

Fold a binary function over an array in a left-associative manner. For example, `foldl('+', 0, xs)` is `sum`, and `foldl('and', true, xs)` is `forall`.

```
any $T: foldl(any $T:(any $T,any $U), any $T, array[$V] of any $U)
```

Fold a binary function over an array in a right-associative manner.

```
any $T: foldr(any $T:(any $U,any $T), any $T, array[$V] of any $U)
```

B Libraries

This section describes some of the Zinc. For full details, please see the comments in the library files themselves.

B.1 `globals.zinc`

The Zinc global constraints library contains a number of global constraints: `all_different`, `disjoint`, `less_leq`, etc.

C Standard Annotations

C.1 Annotations

In addition to the annotations listed in this section, Zinc also supports the FlatZinc annotations (see the FlatZinc specification for details).

C.1.1 Null Annotation

```
null
```

The null annotation has no meaning. It can be useful as a place-holder.

C.1.2 Solve Annotations

Specifies that the model should be solved using a backtracking tree search. `a` is the list of variables to search over, `selector` is the name of a function that takes a list of non-ground variables and decides which one should be considered next, and `brancher` is the name of a function that returns an array of constraints that define the branches of the subsequent search tree.

```
tree_search(array[$K1] of any $V: a,
            op(any $V:(array[$K1] of any $V)): selector,
            op(array[$K2] of var bool:(any $V)): brancher)
```

D Zinc Grammar

Section 3.3 describes the notation used in the following Zinc grammar.

D.1 Items

⟨*model*⟩ ::= [⟨*item*⟩ ; ...]

⟨*item*⟩ ::= ⟨*type-inst-syn-item*⟩
| ⟨*enum-item*⟩
| ⟨*include-item*⟩
| ⟨*var-decl-item*⟩
| ⟨*assign-item*⟩
| ⟨*constraint-item*⟩
| ⟨*solve-item*⟩
| ⟨*output-item*⟩
| ⟨*predicate-item*⟩
| ⟨*test-item*⟩
| ⟨*function-item*⟩
| ⟨*annotation-item*⟩

⟨*type-inst-syn-item*⟩ ::= `type` ⟨*ident*⟩ ⟨*annotations*⟩ `=` ⟨*ti-expr*⟩

⟨*enum-item*⟩ ::= `enum` ⟨*ident*⟩ ⟨*annotations*⟩ [`=` ⟨*enum-cases*⟩]
⟨*enum-cases*⟩ ::= `{` ⟨*enum-case*⟩ `,` ... `}`
⟨*enum-case*⟩ ::= ⟨*ident*⟩ [`(` ⟨*ti-expr-and-id*⟩ `,` ... `)`]
⟨*ti-expr-and-id*⟩ ::= ⟨*ti-expr*⟩ `:` ⟨*ident*⟩

⟨*include-item*⟩ ::= `include` ⟨*string-literal*⟩

⟨*var-decl-item*⟩ ::= ⟨*ti-expr-and-id*⟩ ⟨*annotations*⟩ [`=` ⟨*expr*⟩]

⟨*assign-item*⟩ ::= ⟨*ident*⟩ `=` ⟨*expr*⟩

⟨*constraint-item*⟩ ::= `constraint` ⟨*expr*⟩

⟨*solve-item*⟩ ::= `solve` ⟨*annotations*⟩ `satisfy`
| `solve` ⟨*annotations*⟩ `minimize` ⟨*expr*⟩
| `solve` ⟨*annotations*⟩ `maximize` ⟨*expr*⟩

⟨*output-item*⟩ ::= `output` ⟨*expr*⟩

⟨*annotation-item*⟩ ::= `annotation` ⟨*ident*⟩ ⟨*params*⟩

⟨*predicate-item*⟩ ::= `predicate` ⟨*operation-item-tail*⟩
⟨*test-item*⟩ ::= `test` ⟨*operation-item-tail*⟩
⟨*function-item*⟩ ::= `function` ⟨*ti-expr*⟩ `:` ⟨*operation-item-tail*⟩
⟨*operation-item-tail*⟩ ::= ⟨*ident*⟩ ⟨*params*⟩ ⟨*annotations*⟩ [`=` ⟨*expr*⟩]
⟨*params*⟩ ::= [`(` ⟨*ti-expr-and-id*⟩ `,` ... `)`]

D.2 Type-Inst Expressions

⟨*ti-expr*⟩ ::= `(` ⟨*ti-expr*⟩ `:` ⟨*ident*⟩ `where` ⟨*expr*⟩ `)`
| ⟨*base-ti-expr*⟩
⟨*base-ti-expr*⟩ ::= ⟨*var-par*⟩ ⟨*base-ti-expr-tail*⟩
⟨*var-par*⟩ ::= `var` | `par` | ϵ
⟨*base-ti-expr-tail*⟩ ::= ⟨*ident*⟩
| `bool`
| `int`
| `float`
| `string`
| ⟨*set-ti-expr-tail*⟩
| ⟨*array-ti-expr-tail*⟩
| ⟨*tuple-ti-expr-tail*⟩
| ⟨*record-ti-expr-tail*⟩

| ⟨*ti-variable-expr-tail*⟩
| ann
| ⟨*op-ti-expr-tail*⟩
| { ⟨*expr*⟩ , ... }
| ⟨*num-expr*⟩ .. ⟨*num-expr*⟩

⟨*set-ti-expr-tail*⟩ ::= set of ⟨*ti-expr*⟩

⟨*array-ti-expr-tail*⟩ ::= array [⟨*ti-expr*⟩ , ...] of ⟨*ti-expr*⟩
| list of ⟨*ti-expr*⟩

⟨*tuple-ti-expr-tail*⟩ ::= tuple (⟨*ti-expr*⟩ , ...)

⟨*record-ti-expr-tail*⟩ ::= record (⟨*ti-expr-and-id*⟩ , ...)

⟨*ti-variable-expr-tail*⟩ ::= [any] `$[A-Za-z][A-Za-z0-9_]*`

⟨*op-ti-expr-tail*⟩ ::= op (⟨*ti-expr*⟩ : (⟨*ti-expr*⟩ , ...))

D.3 Expressions

⟨*expr*⟩ ::= ⟨*expr-atom*⟩ ⟨*expr-binop-tail*⟩
⟨*expr-atom*⟩ ::= ⟨*expr-atom-head*⟩ ⟨*expr-atom-tail*⟩ ⟨*annotations*⟩
⟨*expr-binop-tail*⟩ ::= [⟨*bin-op*⟩ ⟨*expr*⟩]
⟨*expr-atom-head*⟩ ::= ⟨*builtin-un-op*⟩ ⟨*expr-atom*⟩
| (⟨*expr*⟩)
| ⟨*ident-or-quoted-op*⟩
| _
| ⟨*bool-literal*⟩
| ⟨*int-literal*⟩
| ⟨*float-literal*⟩
| ⟨*string-literal*⟩
| ⟨*set-literal*⟩
| ⟨*set-comp*⟩
| ⟨*simple-array-literal*⟩
| ⟨*simple-array-literal-2d*⟩
| ⟨*indexed-array-literal*⟩
| ⟨*simple-array-comp*⟩
| ⟨*indexed-array-comp*⟩
| ⟨*tuple-literal*⟩
| ⟨*record-literal*⟩
| ⟨*enum-literal*⟩
| ⟨*ann-literal*⟩
| ⟨*if-then-else-expr*⟩
| ⟨*case-expr*⟩
| ⟨*let-expr*⟩
| ⟨*call-expr*⟩
| ⟨*gen-call-expr*⟩
⟨*expr-atom-tail*⟩ ::= ϵ
| ⟨*array-access-tail*⟩ ⟨*expr-atom-tail*⟩
| ⟨*tuple-access-tail*⟩ ⟨*expr-atom-tail*⟩
| ⟨*record-access-tail*⟩ ⟨*expr-atom-tail*⟩

⟨*num-expr*⟩ ::= ⟨*num-expr-atom*⟩ ⟨*num-expr-binop-tail*⟩
⟨*num-expr-atom*⟩ ::= ⟨*num-expr-atom-head*⟩ ⟨*expr-atom-tail*⟩ ⟨*annotations*⟩
⟨*num-expr-binop-tail*⟩ ::= [⟨*num-bin-op*⟩ ⟨*num-expr*⟩]
⟨*num-expr-atom-head*⟩ ::= ⟨*builtin-num-un-op*⟩ ⟨*num-expr-atom*⟩
| (⟨*num-expr*⟩)
| ⟨*ident-or-quoted-op*⟩
| ⟨*int-literal*⟩
| ⟨*float-literal*⟩
| ⟨*if-then-else-expr*⟩
| ⟨*case-expr*⟩
| ⟨*let-expr*⟩

| ⟨*call-expr*⟩
| ⟨*gen-call-expr*⟩

⟨*builtin-op*⟩ ::= ⟨*builtin-bin-op*⟩
| ⟨*builtin-un-op*⟩
⟨*bin-op*⟩ ::= ⟨*builtin-bin-op*⟩
| ‘ ⟨*ident*⟩ ‘
⟨*builtin-bin-op*⟩ ::= `<->` | `->` | `<-` | `\/` | `xor` | `/\`
| `<` | `>` | `<=` | `>=` | `==` | `=` | `!=`
| `in` | `subset` | `superset` | `union` | `diff` | `symdiff`
| `..` | `intersect` | `++` | ⟨*builtin-num-bin-op*⟩
⟨*builtin-un-op*⟩ ::= `not` | ⟨*builtin-num-un-op*⟩

⟨*num-bin-op*⟩ ::= ⟨*builtin-num-bin-op*⟩
| ‘ ⟨*ident*⟩ ‘
⟨*builtin-num-bin-op*⟩ ::= `+` | `-` | `*` | `/` | `div` | `mod`
⟨*builtin-num-un-op*⟩ ::= `+` | `-`

⟨*bool-literal*⟩ ::= `false` | `true`

⟨*int-literal*⟩ ::= `[0-9]+`
| `0x[0-9A-Fa-f]+`
| `0o[0-7]+`

⟨*float-literal*⟩ ::= `[0-9]+.[0-9]+`
| `[0-9]+.[0-9]+[Ee][-+]?[0-9]+`
| `[0-9]+[Ee][-+]?[0-9]+`

⟨*string-literal*⟩ ::= `"[^"\n]*"`

⟨*set-literal*⟩ ::= { [⟨*expr*⟩ , ...] }

⟨*set-comp*⟩ ::= { ⟨*expr*⟩ | ⟨*comp-tail*⟩ }
⟨*comp-tail*⟩ ::= ⟨*generator*⟩ , ... [`where` ⟨*expr*⟩]
⟨*generator*⟩ ::= ⟨*ident*⟩ , ... `in` ⟨*expr*⟩

⟨*simple-array-literal*⟩ ::= `[` [⟨*expr*⟩ , ...] `]`

⟨*simple-array-literal-2d*⟩ ::= `[|` [(⟨*expr*⟩ , ...) | ...] `|]`

⟨*simple-array-comp*⟩ ::= `[` ⟨*expr*⟩ | ⟨*comp-tail*⟩ `]`

⟨*indexed-array-literal*⟩ ::= `[` [⟨*index-expr*⟩ , ...] `]`
⟨*index-expr*⟩ ::= ⟨*expr*⟩ : ⟨*expr*⟩

⟨*indexed-array-comp*⟩ ::= `[` ⟨*index-expr*⟩ | ⟨*comp-tail*⟩ `]`

⟨*array-access-tail*⟩ ::= `[` ⟨*expr*⟩ , ... `]`

⟨*tuple-literal*⟩ ::= `(` ⟨*expr*⟩ , ... `)`

⟨*tuple-access-tail*⟩ ::= `.` ⟨*int-literal*⟩

⟨*record-literal*⟩ ::= `(` ⟨*named-expr*⟩ , ... `)`
⟨*named-expr*⟩ ::= ⟨*ident*⟩ : ⟨*expr*⟩

⟨*record-access-tail*⟩ ::= `.` ⟨*ident*⟩

⟨*enum-literal*⟩ ::= ⟨*ident*⟩ `(` ⟨*named-expr*⟩ , ... `)`
| ⟨*ident*⟩ `(` ⟨*expr*⟩ , ... `)`
| ⟨*ident*⟩

⟨*ann-literal*⟩ ::= ⟨*ident*⟩ [`(` ⟨*expr*⟩ , ... `)`]

⟨*if-then-else-expr*⟩ ::= `if` ⟨*expr*⟩ `then` ⟨*expr*⟩
(`elseif` ⟨*expr*⟩ `then` ⟨*expr*⟩)*
`else` ⟨*expr*⟩ `endif`

⟨*case-expr*⟩ ::= `case` ⟨*expr*⟩ `{` ⟨*case-expr-case*⟩ `,` ... `}`
⟨*case-expr-case*⟩ ::= ⟨*ident*⟩ `-->` ⟨*expr*⟩

⟨*call-expr*⟩ ::= ⟨*ident-or-quoted-op*⟩ [`(` ⟨*expr*⟩ `,` ... `)`]

⟨*let-expr*⟩ ::= `let` `{` ⟨*var-decl-item*⟩ `,` ... `}` `in` ⟨*expr*⟩

⟨*gen-call-expr*⟩ ::= ⟨*ident-or-quoted-op*⟩ `(` ⟨*comp-tail*⟩ `)` `(` ⟨*expr*⟩ `)`

D.4 Miscellaneous Elements

⟨*ident*⟩ ::= `[A-Za-z][A-Za-z0-9_]*` % excluding keywords
⟨*ident-or-quoted-op*⟩ ::= ⟨*ident*⟩
| `'`⟨*builtin-op*⟩`'`

⟨*annotations*⟩ ::= (`::` ⟨*annotation*⟩)*
⟨*annotation*⟩ ::= ⟨*expr-atom-head*⟩ ⟨*expr-atom-tail*⟩

E MiniZinc

MiniZinc is modelling language that is a subset of Zinc. It is easier to implement, and it can be flattened into FlatZinc in a straightforward manner. For more details on the goals of MiniZinc and FlatZinc, please read *MiniZinc: Towards a Standard CP Modelling Language*, by Nethercote, Stuckey, Becket, Brand, Duck and Tack.

This section defines MiniZinc by describing the features from Zinc that it does not support. The two languages have identical grammars, although various syntactically-correct constructs are not valid in MiniZinc programs, and should be rejected by post-parsing checks.

Rationale. *In the past, the two languages did have distinct grammars, but combining them makes this document much simpler. It also makes the implementation of MiniZinc a little simpler. It can also allow for better error messages if a Zinc-only feature is used in a MiniZinc program; for example, if an enum item appears in a MiniZinc program, instead of a syntax error such as "syntax error at* `enum`*", an more helpful error such as "MiniZinc does not allow enum items" can be emitted. And it allows Zinc features to be added to MiniZinc more easily later on, should we desire.*

E.1 Items

MiniZinc has the following restrictions on items.

- Type-inst synonym items are not supported.
- Enum items are not supported.
- User-defined function items are not supported.
- The expression in a minimize/maximize solve item must have type-inst `int`, `float`, `var int` or `var float`. The first two of these are not very useful as they mean that the model requires no constraint solving.

E.2 Type-insts and Expressions

MiniZinc has the following restrictions on type-insts and expressions.

- Sets can only contain Booleans, integers, and floats. Sets of integers may be fixed or unfixed; other sets must be fixed.
- Arrays must have indices that are contiguous integer ranges (e.g. `0..3`, `10..12`, or the name of a set variable assigned in the model with a range value), or a tuple of contiguous integer ranges.

  Furthermore, MiniZinc arrays must be declared with index types that are explicit ranges, or variables that are assigned a range value. Because these types are finite, only explicitly-indexed array variables can be declared in MiniZinc. The one exception is that implicitly-indexed arrays are allowed as arguments to predicates and annotations.
- Arrays can only contain Booleans, integers, floats or sets of integers (all fixed or unfixed), or fixed sets of Booleans or floats, or fixed strings. Arrays-of-arrays are not supported.
- Indexed array literals and comprehensions are not supported.
- Tuples can only be used as array indices, and must contain integers, and must be fixed. Furthermore, neither tuple literals nor tuple accesses are supported.
- Records are not supported. Therefore, neither record literals nor record accesses are supported. Furthermore, there are no local namespaces for record field names.
- Enums are not supported, including in data files. Therefore, case expressions are not supported. Furthermore, there are no local namespaces for (non-flat) enum field names.
- The language is entirely first-order; no higher-order types are supported, operation type-inst expressions are not supported, and operations cannot be used as values.
- Arbitrarily constrained type-insts are not supported.
- Implicit type coercions (e.g. int-to-float, set-to-array) are not supported, with one exception: set-to-array coercions are allowed in comprehension generators. This allows sets to be used as generator expressions, which is very convenient.

  However, explicit type coercions are supported (e.g. `int2float`), as are implicit inst coercions (e.g. par-int-to-var-int).
- Type-inst variables are not supported in user-defined predicates and functions. However, many of the built-in operations, e.g. `show`, have signatures that feature type-inst variables, and they work with all valid matching MiniZinc types.

E.3 Built-in Operations and Annotations

MiniZinc has the following restrictions on built-in operations and annotations.

- The built-in comparison operators (`'<'`, `'=='`, etc.) are not supported for arrays, i.e. only the following signatures of `'<'` are supported (and likewise for the other comparison operators):

```
    bool: '<'(    int,              int)
var bool: '<'(var int,          var int)
    bool: '<'(    float,            float)
var bool: '<'(var float,        var float)
    bool: '<'(    bool,             bool)
var bool: '<'(var bool,         var bool)
    bool: '<'(    string,           string)
    bool: '<'(    set of int,       set of int)
    bool: '<'(    set of bool,      set of bool)
    bool: '<'(    set of float,     set of float)
var bool: '<'(var set of int,   var set of int)
```

- Only the following signatures of `min` are supported (and likewise for `max`):

```
    int:   min(    int,       int)
var int:   min(var int,   var int)
    float: min(    float,     float)
var float: min(var float, var float)
    int:   min(array[int] of     int)
var int:   min(array[int] of var int)
    float: min(array[int] of     float)
var float: min(array[int] of var float)
    int:   min(set of int)
    float: min(set of float)
```

- Only the following signatures of `'in'` are supported:

```
    bool: 'in'(    int,      set of int)
    bool: 'in'(    bool,     set of bool)
    bool: 'in'(    float,    set of float)
var bool: 'in'(var int,  var set of int)
```

- For `array1d`, `array2d`, etc., the set arguments must be contiguous integer sets, otherwise it is a run-time error.
- The following built-in operations are not supported: `powerset`, `cartesian_product`, `concat`, `head`, `last`, `tail`, `condense`, `condense_int_index`, `show_float`, `first`, `second`, `fst`, `snd`, `foldl`, `foldr`.
- The following built-in annotations are not supported: `tree_search`.

E.4 Other

MiniZinc has the following other restrictions.

- `globals.mzn` is the equivalent to the Zinc `globals.zinc` library.