The Design of the Zinc Modelling Language

KIM MARRIOTT kim.marriott@infotech.monash.edu.au
Clayton School of IT, Monash University, Vic. 3800, Australia

NICHOLAS NETHERCOTE njn@csse.unimelb.edu.au
National ICT Australia, Melbourne, Vic. 3010, Australia

REZA RAFEH reza.rafeh@infotech.monash.edu.au
Clayton School of IT, Monash University, Vic. 3800, Australia

PETER J. STUCKEY pjs@csse.unimelb.edu.au
National ICT Australia, Melbourne, Vic. 3010, Australia

MARIA GARCIA DE LA BANDA mbanda@infotech.monash.edu.au
Clayton School of IT, Monash University, Vic. 3800, Australia

MARK WALLACE mark.wallace@infotech.monash.edu.au
Clayton School of IT, Monash University, Vic. 3800, Australia

Abstract. Zinc is a new modelling language developed as part of the G12 project. It has four important characteristics. First, Zinc allows specification of models using a natural mathematical-like notation. To do so it supports overloaded functions and predicates and automatic coercion and provides arithmetic, finite domain and set constraints. Second, while Zinc is a relatively simple and small language, it can be readily extended to different application areas by means of powerful language constructs such as user-defined predicates and functions and constrained types. Third, Zinc provides sophisticated type and instantiation checking which allows early detection of errors in models. Finally, perhaps the main novelty in Zinc is that it is designed to support a modelling methodology in which the same conceptual model can be automatically mapped into different design models, thus allowing modellers to easily "plug and play" with different solving techniques and so choose the most appropriate for that problem. We describe in detail the various language features of Zinc and the many trade-offs we faced in its design.

Keywords: Language design, modelling, CSP, optimisation, constraint programming

1. Introduction

Solving combinatorial problems is a remarkably difficult task. Formulating the problem precisely is surprisingly hard and typically requires many iterations, while efficient solving often requires the development of tailored algorithms that exploit the structure of the problem. Typically, extensive experimentation is required to determine which technique(s) is most appropriate for a particular problem.

Because of this, the task is often divided into two (hopefully simpler) steps. The first step is to develop a *conceptual model* which specifies the problem without describing how to actually solve it. The second step is to map this conceptual model down to a *design model* which can be directly solved. Ideally, the same

conceptual model can be transformed into different design models, thus allowing modellers to easily "plug and play" with different solving techniques [12, 10, 19].

In this paper we introduce a new modelling language, Zinc, specifically designed to support this methodology. We had four distinct aims in designing Zinc.

1. To support **natural modelling**, allowing the development of clear and concise models using high-level, mathematical-like notation.

2. To support **extensible modelling**, and thus support modelling in a wide variety of applications. Such extensibility is a major innovation in modelling language design and contrasts with, say, OPL which provides ad-hoc built-in types and predicates for resource allocation, but cannot be extended to model new application areas without redefining OPL itself.

3. To support **software engineering** practices that lead to more correct and maintainable models. In particular, the design should allow at least some checking of a model independently of its instance data, with more thorough checking being performed once the data is provided.

4. To have **practical solver-independence**, so that a single conceptual model can be mapped to a design model that uses the most appropriate technique, be it local search, mathematical modelling, constraint programming, or a combination of the above.

Of these, aims 2 and 4 are the most ambitious, since they represent the furthest leaps from existing modelling languages.

These four aims have influenced all aspects of Zinc's design. The following list gives an overview of the most important Zinc features that support these aims. While some of these are found in other modelling languages, many are innovative features that are unique to Zinc.

1. **Natural modelling.** Zinc is a strongly-typed, mostly first-order, functional language with mathematical-like syntax, simple declarative semantics, and support for a range of high-level data structures such as sets, arrays, tuples, records and enumerated types.

   Zinc's first innovation is to provide the widest range of different constraint domains: Boolean, finite domain, sets, and linear and nonlinear constraints on reals and integers. Previous modelling languages have provided only a subset of these.

2. **Extensible modelling.** Zinc's second innovation is to allow the modeller to define new predicates and functions. This is the primary language feature for allowing the modeller to extend Zinc to new application domains. Polymorphism and context-free overloading is supported so as to support natural syntax.

   Zinc's third innovation is to support user-defined constrained types, i.e., to allow constraints to be associated with data types. This further supports modelling in new domains and can lead to very succinct models.

3. **Software engineering.** Zinc's fourth innovation is to provide sophisticated type and instantiation checking of models and data files. (Instantiations represent whether a variable has a fixed value or not.) Such checking identifies many errors early in the development process. In order to allow natural, mathematical-like syntax, the type-inst system supports polymorphic predicates and functions, overloading of functions and predicates, and powerful automatic type coercions (e.g., integers to floats) and instantiation coercions (parameters to decision variables). While most modelling languages provide some overloaded functions and limited coercion, we believe Zinc is unique in its flexibility. Indeed, the combination of coercions and parametric polymorphic overloading allowed in Zinc is novel in any programming language, not just a modelling language.

   Like most modelling languages, Zinc also supports separation of data from model, which means that models can be reused for different instances of a problem without change. And like OPL, Zinc supports data checking by providing assertions.

4. **Practical solver-independence.** Zinc has two crucial restrictions in its expressiveness. First, it is not Turing-complete, as it restricts recursion to iteration over fixed-length data structures. Second, non-global decision variables (e.g., those in user-defined predicates) cannot appear within negative contexts unless they are functionally dependent on non-local variables, thus preventing decision variables from becoming universally quantified. These restrictions ensure that there is a straightforward translation, guaranteed to terminate, from a problem instance (i.e., model plus data) into a satisfaction or optimization problem comprising a (finite) conjunction of (possibly reified) constraints over a global set of existentially quantified variables, a form that can be handled by current solving techniques.

   Zinc's fifth innovation is to provide variable and constraint annotations that specify non-declarative and solver-specific information, and thus allow the modeller to direct the mapping to specific solvers. Annotations are an integral part of Zinc, as they can be declared by the user, and are type-checked.

   We also note that user-defined predicates and functions support solver-specific specialisation of library predicates and functions by allowing different definitions to be used for different solvers.

Zinc is the modelling language for the G12 Project [25] and it is thus intended to be widely used. Therefore, we have spent considerable effort over the last three years on refining and clarifying its design, letting group members use it to model dozens of representative problems, reconsidering design decisions in light of user feedback, and evaluating how Zinc meets its aims.

The aim of this paper is to present the resulting design for Zinc, illustrate how it achieves its four aims, and discuss its main trade-offs and design decisions. We believe this is of interest to designers of future modelling languages, programming language theorists, and solver implementers. Note that although the focus of this

paper is on the design of Zinc rather than on its implementation, we briefly discuss our prototype implementation.

The rest of the paper is organized as follows. Section 2 discusses related work. Sections 3–6 introduce Zinc's features, grouped according to the four aims. Section 7 presents more detailed examples that further illustrate some of the earlier points, and Section 8 concludes.

2. Related Work

The practical importance of combinatorial optimisation problems has resulted in many interfaces and systems for stating and solving them. These range from application-specific, custom-designed interfaces fully integrated with specific algorithms, to generic formal specification languages capable of expressing any problem but of solving none.

2.1. Specification Languages

Generic specification languages, such as Z [3] and B [22], are designed to model a wide variety of computer applications and, therefore, are considerably more expressive than Zinc. For instance, Z is at heart a typed first-order set theory. It includes variables, logic and sets, like Zinc, but it also provides relational algebra, transitive closure, injective and surjective functions, lambda calculus, and even schemas and states. This expressiveness comes, however, at a price. Typically, generic specification languages are Turing-complete and models must be evaluated using theorem proving and other symbolic evaluation techniques. This makes these languages unable to handle large scale industrial applications of combinatorial problems. The (relative) simplicity of Zinc makes it amenable to efficient evaluation, even with large data sets.

There are several less expressive formalisms for various NP-complete problem classes, such as the DIMACS representation for propositional satisfiability (SAT) problems [6], and the standard MPS representation for integer/linear programming [20]. While these are designed to be read in by efficient solving packages, they are application-independent and even algorithm-independent. Also, they are restricted to a small set of constraints, which makes modelling unnatural and difficult. Furthermore, simple problems can become hard when encoded in these languages because the original structure is lost. For example, when encoded into DIMACS form, a "pigeonhole" problem [13] loses its structure, which could have been preserved in the MPS representation. On the other hand, Einstein's Zebra puzzle is represented nicely in DIMACS but it is very hard to represent in MPS [30]. Zinc strictly subsumes both DIMACS and MPS.

The CSP formalism is a more powerful and natural framework than DIMACS or MPS for expressing constraint satisfaction problems. In short, a CSP is of the form $\langle V, D, C\rangle$ and specifies the problem variables V, their domains D and the problem constraints C. Unfortunately, the CSP formalism misses many modelling features. For example, it has to be extended in order to be able to specify optimisation

problems and does not support records, lists and arrays, iteration, or constraint or function abstraction. Moreover, there is no separation between a CSP model and its data—another essential facility for problem modelling. Zinc provides all of these features, and allows CSPs to be naturally encoded using sets.

2.2. Modelling Languages

There has been a considerable body of research into problem modelling which has resulted in a progression of modelling languages including MOLGEN [24], AMPL [8], OPL [26], ESRA [7] and ESSENCE [11].

Several of these languages (including AMPL, GAMS, LINGO, AIMMS, and Mosel, see [17] for a discussion of mathematical modelling languages) model problems in terms of numbers and predefined numerical constraints supported by a range of mathematical programming solvers (e.g., see [1]). These languages support some of the same features as Zinc including arrays of variables, iterations over variables and constraints, and the separation of model and data. However, the more challenging extensions, such as sets and user-defined functions, are only supported for *parameters*, so they can be evaluated before any constraint solving takes place.

There is also a variety of constraint programming languages that offer many of the same features as Zinc—and some that Zinc does not offer. A number of these are listed on the CSPLib web site [14]. Two such languages of particular interest in comparison with Zinc are ECLiPSe [2] and Comet [27].

ECLiPSe, like Zinc, has the objective of separating problem modelling from problem solving. Like Zinc, ECLiPSe programs can be mapped down to different solvers by annotating the constraints appropriately. The main difference is that ECLiPSe is ultimately a full-fledged programming language designed to allow specification of design models as well as conceptual models. Thus, it is Turing-complete and its semantics is inherently procedural. This contrasts with Zinc which is intended to support only specification of conceptual models and, hence, is decidable and declarative. A further difference is the strong type checking provided by Zinc.

Comet, like ECLiPSe, is a procedural, Turing-complete programming language intended to allow the specification of both conceptual and design models. However, it is currently restricted to local search based algorithms.

The last three comparable modelling languages are ESSENCE [11], ESRA [7] and OPL [26]. Like ESSENCE and ESRA, Zinc is a direct descendant of OPL. Consequently, the three languages share many similarities. However, there are also important differences.

ESSENCE and ESRA are designed to support high-level problem models that include, in particular, variables representing sets, relations and functions. Thus, they are akin to specification languages. ESSENCE is strongly typed and includes built-in sets, multi-sets, relations and functions of different kinds (partial/total) and classes (injective, surjective, bijective) and partitions. ESSENCE, like Zinc, has basic types and an orthogonal set of type constructors. One important difference is that, unlike Zinc, ESSENCE and ESRA do not allow user-defined predicates, functions or constrained types and so cannot be readily extended to new application

Table 1. Modelling languages and features. The meaning of each row is as follows. Decidable: all models are solvable. Typed: language is typed. High-level: admits data structures such as records and sets. Con-types: constraints can be associated with (all variables/values of) a type. Coercions: support for both overloading and type coercions. Extensible: core language provides extensible features. Sep-model: model and data can be provided separately. Platforms: models can be mapped to different underlying implementations. Domains: supported constraint domains—continuous arithmetic (C), discrete arithmetic (D), discrete symbolic (E), Booleans (B), sets (S).

	Z	CSP	AMPL	Comet	ECLiPSe	OPL	ESRA	ESSENCE	Zinc
Decidable	-	*Yes*	*Yes*	*Yes*	-	*Yes*	*Yes*	*Yes*	*Yes*
Typed	*Yes*	-	-	*Yes*	-	*Yes*	*Yes*	*Yes*	*Yes*
High-level	*Yes*	-	-	*Yes*	-	*Yes*	*Yes*	*Yes*	*Yes*
Con-types	-	-	-	-	-	-	-	-	*Yes*
Coercions	-	-	-	-	-	-	-	-	*Yes*
Extensible	-	-	-	*Yes*	*Yes*	-	-	-	*Yes*
Sep-model	-	-	*Yes*	*Yes*	-	*Yes*	*Yes*	*Yes*	*Yes*
Platforms	-	-	*Yes*	-	*Yes*	-	-	*Yes*	*Yes*
Domains	DEBS	E	CD	DBS	CDEBS	CDE	DEBS	DEBS	CDEBS

domains. Another difference is that ESRA and ESSENCE are designed to be specification languages for CSPs. Thus, they have only finite domains and are designed so that problems expressed in these languages can be mapped down to CSPs. Zinc, by contrast, includes continuous variables, which are used in many real-world problems to model dimensions, speeds, etc.

As for OPL, Zinc inherited some of its syntax, its support for enumerated domains, its type declarations, some of its data structures, its use of variable array indices, and its support for both continuous and discrete variables. However, Zinc also extends OPL in a number of ways. The most important is that Zinc supports user-defined functions and constraints, and constrained types. Together with user-defined types, these can be used to build libraries providing language support for specific applications. This contrasts with OPL which has specialised support for certain applications, such as resource scheduling, but cannot readily be extended to new application domains. (Section 7 gives an example of resource scheduling with Zinc.) The ability to aggregate multiple constraints in a predicate and reuse it is also useful and promotes good software engineering. Other differences are that Zinc provides set constraints and supports solver independence while OPL is tied to constraint programming or MIP solving techniques.

At the risk of oversimplifying, Table 1 summarises the features supported by different modelling formalisms and languages.

There are some very important languages that we have not discussed in this section, including the "mother of all" languages for modelling combinatorial problems, Alice [18]. Two excellent comparisons of previous and related work in this area are presented in the papers on ESRA [7] and ESSENCE [9].

```
type PosInt = (int: x where x > 0);
PosInt: sizeBase;          % instantiated in the data file

type Square = record(var 1..sizeBase: x, var 1..sizeBase: y, PosInt: size);
list of Square: squares;    % instantiated in the data file

constraint
    forall(s in squares) (
         s.x + s.size <= sizeBase + 1 /\
         s.y + s.size <= sizeBase + 1
    );

predicate
    nonOverlap(Square: s, Square: t) =
        s.x + s.size <= t.x  \/  t.x + s.size <= s.x  \/
        s.y + s.size <= t.y  \/  t.y + s.size <= s.y;

constraint
    forall(i, j in 1..length(squares) where i < j) (
        nonOverlap(squares[i], squares[j])
    );

constraint
    assert(sum(s in squares) (s.size * s.size) == sizeBase * sizeBase,
           "Squares do not cover the base exactly");

solve satisfy;

output [show(squares)];
```

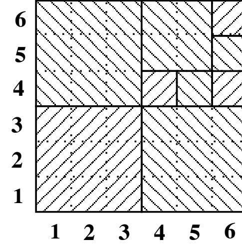

Figure 1. Perfect Squares model and a graphical representation of example data.

3. Natural Modelling

In this section we describe the basic modelling features provided in Zinc, with an emphasis on why each feature was included. We will discuss more advanced modelling features in subsequent sections.

3.1. Overview

We introduce Zinc via two examples. We will discuss the language features present in the examples in more detail in the following sections.

EXAMPLE: Consider the perfect squares problem [29], an instance of which is shown graphically on the right-hand side of Figure 1. The aim is to place multiple small squares, with no overlapping, onto a large base square.

A Zinc model for this problem is given on the left-hand side of Figure 1. The model consists of a sequence of *items*. The first item defines a new type-inst (a type-inst is a type with instantiation information; we will discuss them in more detail in the next section). The type-inst defined, `PosInt`, is constrained to be a positive integer. The second item declares the parameter `sizeBase`, which models the size of the base square as a `PosInt`.

The third item defines a record type, `Square`, which models each of the squares. Its `x` and `y` fields model the (unknown and, thus, a decision variable) position of the lower left corner of the square, and its `size` field models its (known and, thus, a parameter) side length. The fourth item declares `squares`, which represents the small squares, as a list of `Square` elements.

The first constraint in the model ensures that each square is inside the base (`\/` and `/\` denote disjunction and conjunction, respectively). The model contains a user-defined predicate `nonOverlap`, which is used in the second constraint to ensure that no two squares overlap.

The squares provided as input data are assumed to fit in the base exactly. To check this assumption, the model includes an assertion that equates their combined areas with the base's area. An assertion is just a particular kind of constraint used to statically check the input data and to provide an adequate error message whenever such test fails.

The model ends with two final items. The `solve` item indicates that this is a satisfaction problem. The `output` item determines the output (an array of strings, which get concatenated) printed when a solution is found; `show` is a built-in function that converts its argument to a string.

A problem instance is specified by a model and optional data files giving values for the model parameters. In the case of our example, a sample data file is:

```
sizeBase = 6;
squares = [ (x:_, y:_, size:s) | s in [3,3,3,2,1,1,1,1,1] ];
```

The identifier '`_`' represents an unconstrained, anonymous decision variable (of any type), which is most useful when partially initialising data structures. □

Example: As another example of Zinc consider the model for the Minimisation of Open Stacks Problem (MOSP) [15] shown in Figure 2. In MOSP, a factory can manufacture a number of products. Once a product in a customer's order starts being manufactured, a stack is opened for that customer to store the products. Once all the products for a customer are manufactured, the order is sent and the stack closed. The MOSP aims at determining the time sequence in which products should be manufactured to minimise the maximum number of open stacks.

This model has a similar structure to the previous example, but some important differences. First, the `include` item on the first line imports the global constraints library, which defines the `all_different` constraint.

Second, there are two enumerated types: `Customers` and `Products`. While these two types are declared, their elements are not defined in the model and, therefore, must be defined in a data file. Allowing enumerated types of this kind to be defined

in data files gives models flexibility—in this case, it makes the model independent of the exact customers and products.

Third, the model uses sets—both as parameters (in the `ordered` array) and as decision variables (in the `openStacks` array).

Fourth, the `Time` type is an example of a range type, which specifies an integer that is constrained to be in a particular range.

Fifth, `multi_union` is a user-defined function that unions together multiple sets. It is polymorphic (`$T` is a type-inst variable) and uses the built-in higher-order `foldr` function. It is called using the mathematical-like *generator call* syntax, which combines an array comprehension with a call, and so the call:

```
multi_union(ti in 1..t) (ordered[assign[ti]])
```

is syntactic sugar for:

```
multi_union( [ordered[assign[ti]] | ti in 1..t] )
```

Sixth, the `solve` item indicates that this is an optimisation problem, and gives the expression to be minimised. □

3.2. Types, Insts, and Related Expressions

Zinc provides a rich variety of built-in types: integers, floats, strings (for output), Booleans, arrays, sets, tuples and records. It also provides user-defined enumerated types (*enums* for short); these define a set of named alternatives, but, unlike some languages, each alternative can have arguments and so they are more like discriminated union types. Zinc also has *constrained types*, but we defer discussion of them to Section 4.

Every variable in a model has both a type and an *instantiation* (often abbreviated to *inst*), which indicates whether the variable is fixed in the model to a known value (and is thus a parameter, shortened to *par*) or not (and is thus a decision variable, shortened to *var*). A pairing of a type and an inst is called a *type-inst*. We will see that instantiations are written using `par` and `var` prefixes; where a prefix is omitted, it is equivalent to `par`.

> ***Design Decision.*** Requiring the modeller to distinguish between decision variables and parameters is typical in modelling languages, but unusual in most CLP languages (with HAL [5] being the exception). It allows Zinc to check that all parameters are initialised in either the model or in the instance, and it ensures the modeller documents which variables are parameters and which are not. For readability and to simplify error checking, we decided to combine instantiation information with types, rather than introducing another kind of declaration.

Each type has one or more *expressions*, i.e., ways of writing values of that type. For example, sets can be expressed as literals (e.g., `{1,2,3}`) or comprehensions (e.g., `{ i * i | i in 1..5 }`).

```
include "globals.zinc";

enum Customers;
enum Products;
array[Products] of set of Customers: ordered;

type Time = 1..card(Products);
array[Time] of var Products: assign;
array[Time] of var set of Customers: openStacks;

constraint
    all_different([assign[t] | t in Time]);

constraint
    forall(t in Time) (
        openStacks[t] ==
            multi_union(ti in 1..t             ) (ordered[assign[ti]])
            intersect
            multi_union(ti in t..card(Products)) (ordered[assign[ti]])
    );

function var set of $U: multi_union(array[$T] of var set of $U: a) =
    foldr('union', {}, a);

solve minimize max(t in Time) (card(openStacks[t]));

output [show(assign)];
```

Figure 2. Minimization of Open Stacks Problem model.

> ***Design Decision.*** All types have a total order on their elements, thus facilitating the specification of symmetry breaking and of polymorphic predicates and functions. In the case of compound types, the total order on the type is the natural lexicographic ordering based on their component types.

Thus, there are several built-in comparison predicates (e.g. `<` and `==`) that work with every type.

In the rest of this section we discuss the various types, how they are written, their expressions, and why we included them in Zinc. We also discuss the remaining expressions that are not tied to specific types (such as if-then-else expressions). We do not discuss the details of the type-inst checking until Section 5.

3.2.1. Numbers Zinc has integers and floats, both of which are obviously crucial for modelling most problems. They may be *par* or *var*. For example, we might have `par int` or `var float`. Integer and float literals have typical forms, e.g., `23`, `0x4a`,

-44.5, 2.3e-05, with integers being automatically coerced to floats when used in a float context (Section 5 will discuss coercions in more detail).

There are a range of built-in arithmetic operations, e.g. +, *, min, round. Many are overloaded to work on both integers and floats, and all are overloaded to work with both parameters and decision variables.

3.2.2. Booleans Booleans are useful to model many problems. They can be *par* or *var*, and are written as par bool or var bool. The Boolean literals are true and false.

Design Decision. Some solvers represent Booleans as integers. We chose to represent Booleans distinctly because it is less error-prone—integers and Booleans cannot be accidentally combined—and because we believe abstracting away from such implementation details is appropriate in a high-level language and makes it easier to map to different design models.

There are several built-in logical operations, e.g. /\, \/, xor, not.

3.2.3. Strings Zinc supports strings to facilitate output (see Section 3.3.6). All strings must be *par* as, currently, no solvers deal with strings. String literals are written as in C, e.g., "one two three\n".

3.2.4. Sets Sets are crucial for naturally modelling many problems and so Zinc supports them directly. Zinc supports both *par* and *var* sets, but requires set elements to be *par*. For example, var set of bool is legal (it represents a variable which takes as its value a subset of bool), but set of var bool is not (it would represent a set of variables each taking a value from bool).

Design Decision. Currently, we do not directly support sets of *var* elements because this makes it difficult to find the domain for the set.

However, in practice, fixed cardinality sets of *var* elements ranging over a finite domain can be modelled by linking set variables to decision variables via membership constraints:

```
var 1..n: x;  var 1..n: y;  var 1..n: z;
var set of 1..n: s;
constraint card(s) == 3 /\ x in s /\ y in s /\ z in s /\
           all_different([x,y,z])
```

Sets are written using set literals or set comprehensions. For example:

```
set of int: s1 = {1,2,3};
set of int: s2 = { i * j | i,j in 1..10 where i != j }
```

where the 1..10 expression uses the '..' range operator to represent the set containing all integers from 1 to 10. Comprehensions in Zinc are similar to those in OPL. They can have multiple variables per generator, multiple generators, and each

generator can have a filtering `where` clause. The generator variables (`i` and `j` in the example) are local variables that overshadow any global variables of the same name; their types are inferred.

Currently, comprehension generators can be *par* sets but not *var* sets. Thus, the constraint in the following example is not allowed:

```
var set of 1..10: p1;  var set of 1..10: p2;
constraint sum({s | s in p1}) == sum({s | s in p2});    % illegal
```

Instead, the modeller must write the constraint as:

```
constraint sum({bool2int(s in p1)*s | s in 1..10}) ==
           sum({bool2int(s in p2)*s | s in 1..10});
```

where the built-in `bool2int` converts `true` to 1 and `false` to 0. Note here that the first `in` is the set membership operator (i.e., `s in p1` is a Boolean expression), but that the second `in` is part of the comprehension generator.

Design Decision. This restriction simplifies the translation into decision models. However, it has proven to be an initial source of confusion to many Zinc modellers, so we may lift it in the future. In general, it is preferable if the allowed language constructs are independent of variable instantiations.

There are a number of built-in set operations, e.g. `union`, `subset`, `in` (membership), `card` (cardinality).

One important use of sets is for modelling CSP problems involving arbitrary finite domain constraints. For example:

```
var int: x; var float: y;
constraint (x,y) in {(0, 0.0), (1, 1.0), (2, 1.414), (3, 1.732), (4, 2.0)}
```

uses an ad hoc binary relation constraint to approximately model the constraint $y = \sqrt{x}$.

More generally, *var* sets of tuples can be used to model unknown relations over a fixed domain.

3.2.5. Arrays Arrays are not restricted to integer indexes, and so they are actually maps from one type to another. Array elements can have any instantiation, but an array index set must be a *par* set, and arrays themselves cannot be *var*.

Design Decision. This disallows arrays of variable length and arrays with variable index sets, and avoids the possibility of non-terminating iteration over an array decision variable. Multi-dimensional arrays can be simulated by arrays with tuple indices, and there is some syntactic sugar to simplify their use.

Arrays with *var* array elements can be used to model unknown functions over a fixed domain.

Arrays can be declared in two different ways. *Explicitly-indexed* arrays provide indices in their declaration, while *implicitly-indexed* arrays do not (their index sets are computed from the array initialisation). Array literals and comprehensions

come in two forms: indexed and non-indexed. If the indices are integral and not specified, they are assumed to begin at 0. Indexed array literals and comprehensions can also have a default element which allows the user to avoid repetitive literals. Examples include:

```
array[1..3]   of int: a1 = [5, 6, 7];      % explicitly-indexed
array[int]    of int: a2 = [0:8, 1:9];     % implicitly-indexed
array[1..100] of int: a3 = [1:1, 2:2] default 0;
```

In the indexed array literals, the '`:`' separates each index from its value, i.e. `index:value`.

Array accesses are written with the standard notation, for example: `a1[x+y]`. While the array index sets must be *par*, decision variables (and expressions involving decision variables) can be used as array indices in array accesses.

A *par* set used in an array context is automatically coerced to an array if the array context has an integer index and the element types match. The resulting array is sorted and has indices `0..n-1`, where `n` is the cardinality of the set.

> ***Design Decision.*** Earlier versions of Zinc had lists as well as arrays, but there was little difference between lists and arrays with integer indices, so lists were removed. However, we allow `list of <T>` as syntactic sugar for `array[int] of <T>`, because it is more concise. Providing syntactic sugar like this is simpler than having an extra built-in type.

3.2.6. Tuples and Records Zinc has tuples and records, which are fixed-size, heterogeneous collections. The two are similar, but records allow fields to be named. They are both useful for combining related pieces of data into a single entity, and so aid model clarity. Both can contain elements of any instantiation. Tuples used in record contexts are automatically coerced, if the element type-insts match; this can make initialisation more convenient. Examples include:

```
tuple(int, int):          t  = (3, 4);
record(int: x, int: y): r  = (x:3, y:4);
record(int: x, int: y): r2 = (3, 4);    % tuple-to-record coercion
```

> ***Design Decision.*** The order of record fields matters. This is necessary to avoid ambiguity when tuples are coerced to records.

For example, `record(int:x, int:y)` is distinct from and incompatible with `record(int:y, int:x)`. If field ordering didn't 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)`.

Record fields are accessed using a '`.`' followed by the field-name. Tuple fields are accessed similarly, but using field numbers instead of names. For example:

```
int: x2 = r.x;
int: x3 = t.1;
```

Note that the field names and numbers have to appear literally, i.e., they cannot be variables. There are also built-in functions `fst` and `snd` that extract the first and second elements of a pair.

3.2.7. Enumerated Types Enumerated types are user-defined types that provide a set of named alternatives. They are very useful in finite domain problems. We distinguish between *flat* enums, for which none of the alternatives have arguments, and *non-flat* enums, for which one or more alternatives do have arguments.

Flat enums can be *par* or *var*. For example, the following lines define a flat enum and declare a decision variable of that type:

```
enum Colour = { Red, Green, Blue };   % define new enumerated type 'Colour'
var Colour: c;                        % use new type
```

> ***Design Decision.*** Flat enums have two interesting features that make them more useful. First, their names can be used as set expressions. For example, `Colour` can be used in an expression to mean the three-element set `{ Red, Green, Blue }`. Second, flat enums can be declared in a model and defined separately in a data file. Together these features allow more general models.

For example, a model might declare an enumerated type `Product`, and then iterate over the elements of that type (in this case fixing their minimum price):

```
enum Product;       % declares 'Product', does not define it
array[Product] of var int: price;
constraint forall(p in Product) (price[p] > 10);
```

Note that the number of products or their actual names is not specified in the model. Thus, different data files might have different products.

The following lines show an example non-flat enum definition and two corresponding declarations:

```
enum multi_point = {
    int_point(int: xi, int: yi),
    float_point(float: xf, float: yf)
};
multi_point: P1 = int_point(xi:2, yi:3);
multi_point: P2 = float_point(2.3, 5.6);
```

The field names are mandatory in the declarations, but optional in the expressions, as `P2` shows.

> ***Design Decision.*** Enum case names (such as `Red` and `Green`) all inhabit the global namespace. This means that two different enumerated types cannot share any case names. This makes type-inst checking easier, as the type of an enumerated case name is known just from its name. However, for enum field names, each case represents a separate namespace. This means a field name can be used in multiple cases of a single (non-flat) enum.

Non-flat enums have some restrictions than flat enums do not: they cannot be *var*, they cannot be used as set values, and they cannot be defined in a data file. All of these uses become less sensible when arguments are present.

Enums cannot be recursive. For example, the following tree structure is not allowed:

```
enum tree = {
    node(int: value, tree: left, tree: right),  % illegal
    leaf
};
```

Design Decision. The restriction of disallowing recursive enums naturally follows from the decision to disallow recursive functions and predicates (see Section 4.1). Without such capabilities, recursive data structures are of little use as there is no general way to traverse them.

The possible alternatives of an enum (flat or non-flat) can be handled by a case expression, which takes an enum and returns a value. For example:

```
int: M = case P2 {
    int_point   --> r.x,
    float_point --> 0
};
```

The case selection expression (`P2` in this case) must be *par*. Field access of non-flat enums is done via '`.`'; this is only allowed within case expressions so that uses of a field name from the wrong case can be detected statically. Current implementations of Zinc do not provide pattern matching, but this is something we hope to provide in the future.

3.2.8. Function and Predicate Calls Normal function and predicate calls have a very standard syntax, including those built-in functions that are *operators* (i.e., are non-alphanumeric and can be written in an infix or prefix manner). Binary functions can be used in infix form by back-quoting them, and operators can be used like normal identifiers by single-quoting them. Examples include:

```
plus(3, 4)          3 + 4          '+'(3, 4)          3 `plus` 4
```

As Section 3.1 showed, any predicate or function `P` (including user-defined functions or predicates) that takes a single array argument can be called with special, mathematical-like syntax that combines an array comprehension with a call. A *generator call* `P(Gs)(E)` is equivalent to `P([E | Gs])`. The parentheses around the `E` are mandatory; this avoids ambiguity if the generator call is part of a larger expression.

Design Decision. Both the generator calls and the `` `plus` ``-style infix calls are aimed at making Zinc's syntax more mathematical and easier to read.

3.2.9. If-then-else Expressions If-then-else expressions (e.g. `if C1 then A else B endif`) allow conditional choices. Note that `A` and `B` can be any expression (as long as they have the same type). For simplicity, the condition of an if-then-else must currently be *par*, although we may lift this restriction in the future. An example usage is:

```
[ if i mod 2 == 0 then i else -i endif | i in 1..100 ]
```

3.2.10. Let Expressions Let expressions can be used to introduce local variables:

```
let { int: x = 1, int: y = 2 } in x + y
```

The let-local variables (`x` and `y` in the example) overshadow any global variables of the same name. They are most often used within predicates to declare local variables (see Section 4.1).

> ***Design Decision.*** One complication is that local variables inside a negation implicitly become universally quantified. Since current continuous arithmetic solvers cannot handle unrestricted universal quantification, we restrict the kinds of local variables that can occur in a negative context.

To see the problem consider the constraint:

```
var float: x;
not (let { var float: y } in y > 0 /\ x > y);
```

This is not equivalent to

```
var float: x, y;
not (y > 0 /\ x > y);
```

and cannot be directly handled with a linear constraint solver.

A `let` expression is in a *negative context* if it occurs in a formula of the form $\neg F$, $F_1 \leftrightarrow F_2$ or in the sub-formula F_1 in $F_1 \rightarrow F_2$ or $F_2 \leftarrow F_1$. A local variable introduced by a let expression in a negated context must have an associated assignment which ensures that the variable is functionally-dependent on non-local variables and parameters, e.g., `let { int: x = max(a,b) } in ...`

Functionally-dependent local variables have the nice property that they can, with appropriate renaming, be treated as if they are existentially quantified global variables even if they occur within a negative context. More exactly, $\neg(\exists x \,.\, x = f(\bar{y}) \wedge F(x, \bar{y}))$ is logically equivalent to $(\exists x \,.\, x = f(\bar{y}) \wedge \neg F(x, \bar{y}))$ if f is a partial function. Thus, Zinc allows a local variable in a negative context so long as it is functionally-dependent.

The astute reader will have noticed that functional-dependency is not strictly required for non-continuous local variables since range-restricted universal quantification is solvable. In fact, Zinc allows range restricted local variables inside negative contexts but they must be expressed using an iteration expression rather than a let expression. Thus, for example, the constraint:

```
var 0..n: x ; var 0..n: y;
not ( let { var int: z } in z > 1 /\ x mod z == 0 /\ y mod z == 0 );
```

is illegal. Instead the user can write:

```
not exists (z in 2..n) (x mod z == 0 /\ y mod z == 0);
```

where `exists` is a library predicate with the following definition

```
predicate exists(list of bool: L) = foldl('\/', false, L);
```

3.3. Items

This section describes Zinc items, with the exception of enumerated type items which were described in Section 3.2.7, and function and predicate items which will be described in Section 4.

3.3.1. Include Items Include items allow libraries to be used in models, and also allow models to be split across multiple files. For example, the item:

```
include "globals.zinc";
```

imports the file called `globals.zinc`, which is a standard Zinc library. Include items effectively cause the text of the include file to be inserted at the inclusion point. Since Zinc has a single global namespace, all global identifiers (e.g. global variables, predicates and functions, type-inst synonyms) must be distinct.

> ***Design Decision.*** Items can appear in any order within a model. Therefore, identifiers do not need to be declared before they are used, and the position of an include item is not important (as opposed to, for example, OPL or C). However, cyclic dependencies cause an error.

> ***Design Decision.*** For simplicity, Zinc lacks a module system. We have so far not found the need for one, because Zinc models tend to be fairly small. If bigger models become common and the single global namespace becomes a problem, we may add a module system.

3.3.2. Type-inst Synonym Items Type-inst synonym items declare new names for type-insts. For example:

```
type Point = tuple(float, float);
```

They can make models more concise and readable, as we saw with the `Square` synonym in Figure 1.

3.3.3. Variable Declaration and Assignment Items Variable declaration items introduce variables and possibly initialise them. Assignments can also be separated from variable declarations. This is required to allow parameters to be defined in data files, but decision variables can be assigned to as well. Examples include:

```
int: a;
int: b = 3;
var int: c = max(x, y);
int: d;
d = 4;        % could be in a data file
```

3.3.4. Constraint Items Constraint items form the heart of a model. For example:

```
constraint x > 0;
```

The expression in a constraint item must be Boolean (and is usually *var*).

3.3.5. Solve Items Every model must have exactly one solve item, which 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. For example:

```
solve satisfy;
solve minimize cost;
solve maximize 3*x + 2*y;
```

The expression in a minimize/maximize solve item must be an integer or float expression.

3.3.6. Output Items Each model can have at most one output item, which is run when a solution is found. It specifies an array of strings, which are concatenated and then printed. It can use `show`, which converts any value, even one that is not fixed, to a *par* string. For example:

```
output ["The solution is " ++ show(x) ++ "\n"];
```

If a solution is found but no output item is specified, the default output routine prints the name and value of each decision variable. If no solution is found, the string "No solution found" is printed.

Although printing output nicely might require tests on variables, if-then-else conditions must be *par*. Since decision variables should be fixed by the time the output item runs, Zinc has a built-in function `fix` which returns a value if it is fixed, and aborts otherwise. For example:

```
output ["The solution is " ++
    (if fix(x) > 0 then "positive" else "non-positive" endif) ++ "\n"];
```

4. Extensible Modelling

As mentioned in the introduction, one of Zinc's main design aims was to allow the modeller to extend Zinc to different application domains. User-defined predicates and functions and constrained types are the main mechanisms for doing this, since they allow the modeller to create new libraries for different applications.

One of the most powerful features of Zinc is the ability for users to define their own predicates and functions, such as `nonOverlap` in Figure 1.

All predicates, including user-defined predicates, are regarded as Boolean functions (i.e., their return type-inst is `var bool`) and can be combined using the standard Boolean operators. Zinc also support "tests", which are similar to predicates but return a `par bool`. They are introduced via the `test` keyword instead of the `predicate` keyword.

Zinc supports parametric polymorphism in function and predicate arguments. Type-inst variables are written with a `$` prefix, e.g., `$T`. For example, consider the polymorphic function `between`:

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

This returns a *par* value that will be true if the value of `y` is between the values of `x` and `z`. Thanks to the use of type-inst variable `$T`, the function applies to *par* values of any type! If we want to match a non-*par* values, we must use precede the `$T` with the keyword `any`:

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

This version works for values of any type-inst. However, it returns a `var bool` instead of a `par bool`. (Note that the first version of `between` could have been declared as a `predicate`, and the second version could have been declared as a `test`.)

Note that although `$T` (and also `par $T`) only *matches* fixed type-insts, it does not mean that the type-inst variable `$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
```

Zinc supports context-free overloading. For example, the library function `sum` is overloaded to take either integers or floats:

```
function var int:   sum(array[int] of var int:   xs) = foldl('+', 0, xs);
function var float: sum(array[int] of var float: xs) = foldl('+', 0, xs);
```

Design Decision. Currently, when a function or predicate is overloaded, its body must be duplicated for each version. This is awkward when two versions have the same body, but it has not caused problems so far. If it does become a bigger problem, we will introduce syntax that allows multiple versions of an overloaded function or predicate to share the same body.

Although the average modeller might not find the need to use overloading and polymorphism, it allows standard modelling functions, such as `all_different`, to be defined in Zinc itself. (See Section 6.4 for the definition.)

Much of the power of user-defined functions and constraints comes from the ability to use the two higher-order built-in functions `foldl` and `foldr`. Zinc is an entirely first-order language, except for these two functions. In a call to `foldl(fun, init, array)`, the binary function `fun` is applied to each element in `array` (working left-to-right) with an initial accumulator `init`; `foldr` is the same but works right-to-left. These two functions, plus set and array comprehensions, are the only iteration constructs available in Zinc. They are standard in functional programming languages, and they allow various standard modelling operations to be defined in Zinc itself. For example:

```
predicate forall(array[int] of var bool: xs) = foldl('/\', true, xs);
```

We also saw their use in the definition of `sum` above, `multi_union` in Section 3.1, and `exists` in Section 3.2.10.

Design Decision. We believe providing `foldl` and `foldr` is preferable to hard-wiring operations like `forall` and `sum` into the language, since in language design providing general building blocks is usually preferable to canned solutions.

Design Decision. One potential difficulty with user-defined predicates and functions is that if recursive definitions are allowed, the satisfiability of models can become undecidable. For this reason, Zinc does not allow functions or predicates to be recursive.

4.2. Reflection functions

When defining functions and predicates that use arrays, often the index set for the array is also needed. Thus, the index set is implicitly passed as part of the array, and the built-in function `index_set` can be used to return the index set for any array. This is less error-prone and also less cumbersome than requiring modellers to explicitly pass around index sets. For example:

```
function array[$V, $U] of $T: transpose(array[$U, $V] of $T: a) =
     [ (i.2, i.1): a[i.1, i.2] | i in index_set(a) ];
```

defines a polymorphic function for transposing a matrix (recall that a multidimensional array is encoded as a one-dimensional array with a tuple index.)

Similarly, when *var* sets and *var* numbers are passed as arguments to a predicate, it is often required to declare local variables whose domain is dependent on the domains of those arguments. Zinc provides built-in functions `lb`, `ub` and `dom` so

that variable domains need not be passed explicitly. For *var* numbers, `lb` and `ub` return the declared lower and upper bounds, respectively. For *var* integers, `dom` returns the declared domain (as a set). For *par* numbers, `lb` and `ub` both return the actual number, and `dom` returns the actual number in a singleton set. Finally, `ub` is overloaded to return the declared domain of *var* sets.

4.3. Constrained types

One of the novel features of Zinc is that a programmer can associate a unary constraint with a type. This generalises more ad hoc mechanisms in earlier modelling languages in which some built-in types had associated constraints, such as OPL which provides positive floats and positive integers as primitive types. It is also related to constrained objects [16]. Such *constrained types* allow concise, natural modelling. Two examples are:

```
type PosInt   = (int: i where i > 0);
type Interval = (record(var int: start, var int: end):
                   r where r.end >= r.start);
```

As a convenient shorthand, for certain kinds of sets (such as ranges and set literals) a set S of elements with type t can be used as a constrained type. This works like a traditional finite domain, but over a specific type t. For example, the first of the following declarations is a shorthand for the second:

```
1..n: x;
(int: i where i in 1..n): x;
```

A constrained type can be used anywhere a standard type can be used. In particular, they can be used in a global variable declaration, local variable declaration or a function/predicate parameter declaration. However, it is not immediately obvious what their meaning should be in these different contexts.

In the case of a global variable declaration the meaning does seems obvious: if the variable is a parameter then its value should satisfy the constraint, if it is a decision variable, then the constraint from the type places a constraint on the allowed values for the variable. In this, the *constraint view* of constrained types, the constrained type can be seen as a shorthand for specifying structural constraints that are common to a set of variables. For example, the Zinc item:

```
var PosInt y;
```

is simply syntactic sugar for

```
var int y;
constraint y > 0;
```

Another possible meaning for constrained types is provided by type theory. In this view a constrained type defines a subtype of the base unconstrained type, i.e. the meaning of the constrained type is the set of base type elements satisfying the constraint. However, the standard view of types (and subtypes)—as sanity checks whose removal will not affect the operation of a (type) correct program—is at odds

with the expected behaviour of constrained types for global variable declarations since the modeller wants the constraint in the constrained type of a decision variable to actively constrain the solutions.

The constraint view of constrained types naturally extends to local variables and provides a reasonable semantics. Note that in the case of local variables in a negative context the constraint must remain inside the negative context. For example,

```
constraint not (let {var PosInt: y = 2*x} in p(y));
```

is simply syntactic sugar for

```
constraint not (let {var int: y = 2*x} in y > 0 /\ p(y));
```

One complicating issue is that local variables can occur in non-Boolean expressions. In this case the constraint "percolates" out to the closest surrounding Boolean context and so the expression fails if its arguments fail the constraint.

While this might seem counter-intuitive, in other modelling languages it is the de facto semantics for expressions such as an array access `a[x]` which in the case `x` is a decision variable has the effective semantics of constraining `x` to be in the index set of the array `a`.

However, the constraint view is somewhat counter-intuitive when constrained types are used in declarations of the formal arguments of predicates and functions. Consider the following definitions:

```
function var PosInt:bmax(var PosInt:x, var PosInt:y) = max(x, y);
predicate gt(var PosInt:x, var PosInt:y) =  x >= y;
```

and this model:

```
var int: x = 5;  var int: y = -6;  var int: z;
constraint z = bmax(x,y) \/ z = 1;
```

From the constraint viewpoint, the model will succeed since the constraint is equivalent to:

```
constraint (x > 0 /\ y > 0 /\ z = max(x,y)) \/ z = 1;
```

and so functions are now effectively partial functions.

However, it is likely that this is not the intended behaviour and that calling `bmax` with a negative integer indicates a modelling error. Things are even more problematic in negative contexts. Consider:

```
var int: x = 5;  var int: y = -6;  var int: z;
constraint ( gt(x,y) /\ z = 1) \/ (not gt(x,y) /\ z = 2);
```

In the constraint viewpoint this is equivalent to:

```
var int: x = 5;  var int: y = -6;  var int: z;
constraint ( (x > 0 /\ y > 0 /\ x >= y) /\ z=1 )
        \/ ( not (x > 0 /\ y > 0 /\ x >= y) /\ z=2 );
```

so it is satisfiable with $x = 5 \wedge y = -6 \wedge z = 2$ which is probably not what the modeller expects since $x \geq y$.

The un-intuitiveness of this behaviour is even more apparent in the case of parameters. A reasonable principle underlying the design of Zinc is that wherever reasonable, parameters should have the same behaviour as logically equivalent decision variables. But this means that in the constraint viewpoint, this:

```
int: x = 5; int: y = -6;
z = if gt(x,y) then 1 else 2;
```

will succeed with $x = 5 \wedge y = -6 \wedge z = 2$ and no indication of a potential error, which seems unlikely to be the intended behaviour.

In this case the type-theoretic viewpoint seems closer to the intuitively expected behaviour since our belief (corroborated by our experience with existing Zinc models) is that in correct models the actual arguments of user-defined predicates and functions satisfy the formal constrained types, and that if they do not then this most likely indicates an error. Thus, constrained types in the formal arguments of functions and predicates should be treated as subtypes rather than as additional constraints on the actual arguments.

Design Decision. This suggests a hybrid approach, which we use, to handle constrained types: their semantics is given by the constraint view but the modeller is warned if the constraint view differs from the type-theoretic view for arguments of functions and predicates. If no warning is generated this means that the logical meaning of the program would be unchanged if the constraints on the formal arguments were removed.

5. Software Engineering

As mentioned in the introduction, Zinc is designed to support good software engineering practices, particularly by facilitating error-checking. Zinc model instance evaluation has three distinct steps. The first is *model checking* which performs an initial data-independent checking of the model. The second is *instance checking*, which is performed when the data is combined with the model to create the problem instance. The third step is *instance evaluation*, which is performed when the instance is solved. We have designed Zinc so that checking of models and data can occur as early as possible in this pipeline, so that errors can be caught early in the design process. In particular, where possible, error checking occurs at model checking time since this provides the assurance that the model is correct for *all* instances.

5.1. Type-inst checking

The main software engineering feature is Zinc's strong type and inst checking. This catches a lot of potential errors, and also improves program readability by requiring the modeller to provide type and instantiation information about most variables.

We initially tried to separate type checking from inst checking, but types and insts are so closely intertwined in Zinc that we found that doing them together was much more natural.

The algorithm for type-inst checking is basically a straightforward Hindley-Milner style algorithm, but the presence of automatic coercions (which make the syntax more mathematical) complicates things significantly. In this section we provide a sketch of the algorithm and some examples to illustrate the main ideas.

5.1.1. Type-inst lattice The Zinc type-inst lattice is shown in Figure 3. It figuratively demonstrates the type ordering and the least upper bound (*lub*) operation. From these, the following possible coercions are clear.

- Integers can be automatically coerced to floats.
- Sets can be automatically coerced to arrays, if the elements have appropriate type-insts. The resulting array contains the set elements in order, and has an index set $0..n-1$, where n is the size of the set.
- Tuples can be automatically coerced to records, if the field type-insts are appropriate and the sizes match.
- *Par* values can be automatically coerced to *var* values, where applicable.
- The anonymous variable ('`_`') has type `var` $\perp$ and can be automatically coerced to any *var* type.

These coercions can be combined, for example, a `par int` can be coerced to a `var float`.

The coercion for arrays is slightly more restrictive than it needs to be since we could allow `array[`S`] of` $T \preceq$ `array[`V`] of` U iff $T \preceq U$ and $V \preceq S$. For simplicity we currently do not, and in practice this restriction has caused no problems so far. Again for simplicity of implementation, we currently do not perform automatic coercion for functions using the relation $f(S_1, ..., S_n) \rightarrow T \preceq f(U_1, ..., U_n) \rightarrow V$ iff $T \preceq V$ and $S_i \preceq U_i, 1 \leq i \leq n$. The main reason is that since Zinc is almost purely first-order, such coercion is only useful in very rare circumstances.

If there is a coercion from type-inst T to type-inst T', Zinc effectively treats T as a subtype-inst of T' during type-inst checking. We write this as $T \preceq T'$. The subtype-inst relationship lifts to non-primitive types in the obvious way.

5.1.2. The basic algorithm Before type-inst checking occurs, the code is topologically sorted so that all symbols are declared before they are used. This allows type-inst checking to be performed in a single pass.

The information flow during type-inst checking goes in two directions. First, there is bottom-up information flow: the type-inst of any expression can be determined just by looking at the type-insts of its sub-expressions, independent of context.

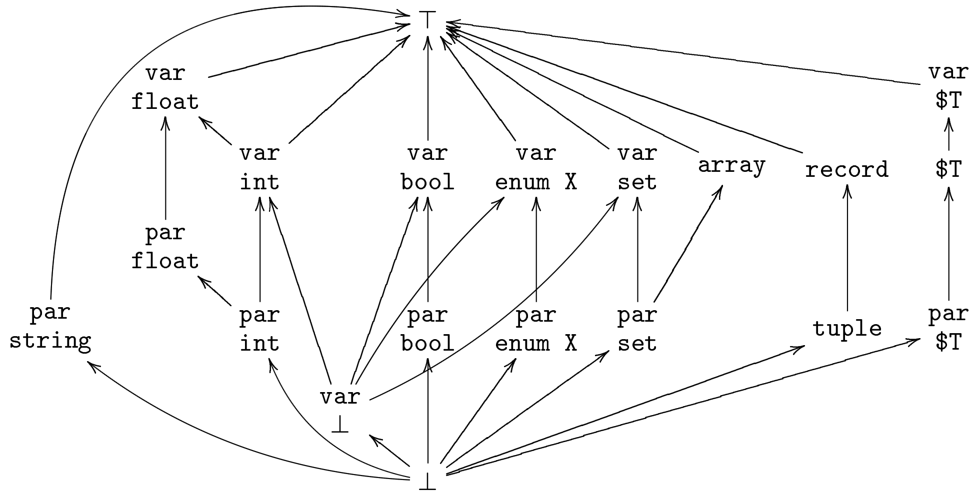

Figure 3. Zinc type-inst lattice. An arc $T \rightarrow T'$ indicates that $T \preceq T'$ and thus T is coercible to T'. `var` $\bot$ is the type-inst of an anonymous variable '`_`'. The diagram omits the following details:
`par set of` $T \preceq$ `par set of` U iff $T \preceq U$;   `par set of` $T \preceq$ `var set of` U iff $T \preceq U$;
`par set of` $T \preceq$ `array[int] of` U iff $T \preceq U$;   `var set of` $T \preceq$ `var set of` U iff $T \preceq U$;
`array[`S`] of` $T \preceq$ `array[`S`] of` U iff $T \preceq U$;
`tuple(`$T_1, \ldots, T_n$`)` $\preceq$ `tuple(`$U_1, \ldots, U_n$`)` iff $T_i \preceq U_i, 1 \leq i \leq n$;
`tuple(`$T_1, \ldots, T_n$`)` $\preceq$ `record(`$U_1 : x_1, \ldots, U_n : x_n$`)` iff $T_i \preceq U_i, 1 \leq i \leq n$;
`record(`$T_1 : x_1, \ldots, T_n : x_n$`)` $\preceq$ `record(`$U_1 : x_1, \ldots, U_n : x_n$`)` iff $T_i \preceq U_i, 1 \leq i \leq n$.

> ***Design Decision.*** All variables must be declared with a type-inst except for local variables occurring in an array or set comprehension. The reason for requiring explicit typing is that automatic coercion and separate data files precludes complete type inference.

For expressions that involve collections of expressions (e.g., sets and arrays), the overall type-inst of the elements is the *lub* of the individual elements' type-insts.

Second, there is top-down information flow: once an expression has had its type-inst determined, we may have to check that it has an appropriate type-inst for its context. This may require adding a coercion.

An example makes this clearer. Consider this item:

```
array[int] of var float: x = [1,2,_];
```

Sub-expressions `1` and `2` are `par int`, while '`_`' is `var` $\bot$. The overall type-inst context for the array elements is the *lub* of these, which is `var int`. Therefore, all three elements need to be coerced, resulting in this array literal:

```
[ coerce(par int, var int, 1),
  coerce(par int, var int, 2),
  coerce(var bottom, var int, _) ];
```

where `coerce(T1, T2, E)` is the coercion of expression `E` from type-inst `T1` to `T2`, and `var bottom` is a synonym for `var` $\perp$. While such explicit coercions are not legal Zinc expressions, they show how we add coercions internally in the compiler.

The array literal thus has a type-inst of `array[int] of var int`. We then consider the assignment context, which is `array[int] of var float`. The array literal's type-inst is less than this, but not equal, so it needs to be coerced. One might expect this result:

```
array[int] of var float: x =
    coerce(array[int] of var int, array[int] of var float,
          [ coerce(par int, var int, 1),
            coerce(par int, var int, 2),
            coerce(var bottom, var int, _) ];
```

and in general, Zinc can generate any required coercions for compound types, e.g., `coerce(array[int] of var int, array[int] of var float, x)`. However, when adding coercions we "push them down" in order to apply them as early as possible, so in this case the final result becomes:

```
array[int] of var float: x =
    [ coerce(par int, var float, 1),
      coerce(par int, var float, 2),
      coerce(var bottom, var float, _) ];
```

We also perform compile-time coercions of constants where possible. For example, if we have the following item:

```
record(float: x, float: y): point = (3, 4);
```

The expression `(3, 4)` has type-inst `tuple(int, int)`, which can be coerced to `record(float:x, float:y)`. These can be done at compile-time, giving:

```
record(float: x, float: y): point = (x: 3.0, y: 4.0);
```

Overall, this combination of behaviours—the coercion of both scalar and compound types, the "pushing down" of coercions to make them occur as early as possible, and the compile-time coercions—makes automatic coercion more powerful in Zinc than in most programming languages.

5.1.3. Overloading As mentioned before, Zinc allows functions and predicates (both built-in and user-defined) to be overloaded so as to allow natural syntax. Unfortunately, combining overloading with automatic coercion causes potential ambiguities (and it is therefore avoided in most programming languages) because different placement of coercions in an expression may allow different choices for the overloaded function. For instance, if the 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.

Let two overloaded versions of a predicate/function have input type-inst $(S_1, ..., S_n)$ and $(T_1, ..., T_n)$. We say the overloaded versions *overlap* if the greatest lower bound (*glb*) of every S_i and T_i pair is not $\perp$.

Design Decision. To ensure that different choices do not lead to different results, we require overlapping overloaded versions of each predicate/function to be semantically equivalent with respect to coercion.

Currently, this requirement is not checked and the modeller must satisfy it manually. We may introduce a way of sharing bodies among different versions of overloaded functions or predicates, which would provide automatic satisfaction of this requirement.

The other potential problem caused by combining coercion and overloading is the need for the compiler to explore different choices of coercions and overloading. For instance, 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.

Design Decision. We avoid the need for such context-dependent, possibly combinatorial search during type-inst checking by requiring that overloaded predicates and functions satisfy two properties. First, they must be closed under intersection of their input type-insts, that is, if overlapping overloaded versions have input type-inst $(S_1, ..., S_n)$ and $(T_1, ..., T_n)$ then there is another overloaded version with input type-inst $(R_1, ..., R_n)$ where each R_i is the *glb* of S_i and T_i. Second, they 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 $S \preceq T$.

In the above example, 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);
```

The three overloadings satisfy the monotonicity requirement since 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.

5.1.4. Array accesses and varification Array accesses are simple when the array index is *par*—the overall type-inst is the same as that of the array element. For example, if we have:

```
array[int] of par int: a = [1,2,3];
```

then the access `a[1]` has type-inst `par int`. However, if we have an integer decision variable `i` then the access `a[i]` (which is equivalent to an `element` constraint) has type-inst `var int`.

More generally, for any array access with a non-*par* index, the type-inst of the result is the *varified* type-inst of the array elements. All the type-insts that can be decision variables can be varified: integers, floats, Booleans, sets, and flat enumerated types (in the *var* cases, the varification does not change the type-inst, e.g., `var int` is unchanged by varification). Tuples and records can also be varified, by varifying their constituent elements. Arrays cannot be varified, nor can type-inst variables.

5.2. Domain Checking

Zinc also requires some other checks on the model. Set solvers require set variables to have a known finite domain. Zinc requires model time checking (i.e., checking before the instance data is provided) based on the type of the set to confirm that this is true. We check that the type used in the type declaration of any uninitialised *var* set is a *finite type*: a Boolean, a flat enumerated type, a constrained type defined via a range expression or set expression (e.g., `1..10`), or a tuple, record, set or non-flat enumerated type containing only finite types. Thus, for example:

```
var set of {1.1, 1.2, 1.3}: s;
```

is legal while the following is not:

```
var set of float: s;
```

Similar checking is performed for *var* integers, although in this case if the type is not finite, only a warning is generated rather than an error, and typically a finite domain solver will use a large default domain for the variable.

5.3. Data files

As mentioned before, Zinc (like many modelling languages) allows the modeller to separate the generic model from instance data by allowing the modeller to specify data files for the model at execution time. Data files consist of variable assignment items and flat enumerated type definitions.

> ***Design Decision.*** Unlike most other modelling languages, Zinc allows the right-hand side of a variable assignment in a data file to be arbitrary Zinc expressions, rather than just constants. This improves readability of data files.

Identifiers can be used before they are declared, as Section 3.3.1 explained. This means there are no ordering issues relating to how data files are combined with model files.

5.4. Assertions

Assertions are used to check data integrity and catch modelling errors. There are two built-in assert functions, with the following signatures:

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

The first one checks that the condition `c` is true. If so, it returns the value `val`. If not, it aborts and prints `s`. It is particularly useful in functions and predicates, for example:

```
$U: function array_head(array[$T] of $U: a) =
    assert(length(a) > 0, "array_head: array is empty",
           a[min(index_set(a))]);
```

The second one is equivalent to the first, but with `true` as the third argument. It is useful for global assertions, via constraint items. For example, if `S1` and `S2` are sets, the following constraint ensures they have the same cardinality:

```
constraint assert(card(S1) == card(S2),
                  "S1 and S2 must have the same cardinality");
```

> ***Design Decision.*** Originally, Zinc had assertion items which could only assert global properties, as in the second example above. We decided to use assertion expressions because they are more flexible and do not require an extra language construct.

6. Practical Solver-independence

One of Zinc's main design aims was to allow the same conceptual model to be mapped to different solving techniques and solvers. In this section we describe the particular language features that support this.

6.1. Implementability

As we have seen, Zinc is a very high-level, expressive modeling language. While this makes it ideal for developing conceptual models, it also introduces a considerable gap between the conceptual Zinc model and an associated design model targeted to a specific solver and search technique. Indeed, it may not be obvious that this gap is bridgeable.

In order to demonstrate that Zinc can in fact be implemented using current constraint solving technology, we briefly describe how a model instance expressed in Zinc can be translated to a model in an intermediate language, called the Solver-Independent Flattened Zinc Model (SI-FZM), that is much closer to existing solvers and so is clearly solvable. SI-FZM is described more fully in [21], as is the flattening process.

The first step to generate the SI-FZM instance from a Zinc model and its associated data file(s) is to insert all assignment items from the data file(s) into the model. From then on, one or more of the following steps are performed to every item in the problem instance:

- Evaluate all parameters and check the associated assertions hold.
- Determine an initial domain or range for all decision variables.
- Simplify record types by: (a) replacing all records by tuples, (b) flattening tuples of tuples into a single tuple, and (c) appropriately replacing field accesses by the contents of the field addressed.
- Replace flat enumerated types by integer range types, and any constraints involving flat enumerated types by the appropriate integer constraints.
- Unfold calls to `foldl` and `foldr`, unfold set and array comprehensions, and unfold (i.e., inline) calls to user-defined predicates and functions. Note that this may introduce new variables due to the formal parameters and to the existence of local variables in the definitions.
- Insert constraints arising from constrained types. If these involve only parameters, check that they hold.
- Replace local variables by new global variables and appropriately rename them.
- Simplify arrays by rewriting them to be one-dimensional arrays with an integer index set starting from 0, and appropriately updating the computation of the array index in constraints.
- Translate variable sets of structured types into variable sets over integers, and add a constraint mapping the structured type elements to integers. This is also used to flatten sets of sets into linked sets of integers. For instance:

  ```
  var set of {{2,5},{1,3,6},{1,2}}: S1;
  var set of {{2,5},{1,2},{3,4}}: S2;
  constraint {1,2} in S1 \/ {3,4} in S2;
  constraint card(S1 intersect S2) == 1;
  ```

  is translated to (assuming the encoding starts from 1):

  ```
  var set of {1,2,3}: S1;
  var set of {1,3,4}: S2;
  var set of {1,2,3,4}: S12;
  constraint 1 in S1 \/ 4 in S2;
  constraint S1 intersect S2 == S12;
  constraint card(S12) == 1;
  ```

  with appropriate insertion of `element` constraints to link the structured type elements with their integer encoding.

- Use reification to separate logical combinations of constraints from the constraints themselves, i.e., replace each constraint c with reify(c, B) which constrains Boolean variable B to be true iff c holds. For example, the constraint:

  ```
  constraint (x < y \/ x < z) /\ (x > w)
  ```

  is replaced with:

  ```
  constraint reify(x < y, B1);
  constraint reify(x < z, B2);
  constraint reify(x > w, B3);
  constraint B4 == B1 \/ B2;
  constraint B4 /\ B3;
  ```

  Note that reification is performed after unfolding predicates and functions, leaving only constraints defined by the underlying solvers.

As mentioned before, while Zinc provides many of the standard types and operations found in modern programming languages, it is not Turing-complete. In particular, it only allows iteration (via `foldl`, `foldr`, and comprehensions) over finite sets and arrays whose length is known, and does not allow the user to define recursive predicates or functions. These restrictions are key to the viability of SI-FZM since they guarantee that the above process terminates and only finitely many new variables are introduced.

Note that, as we have discussed, all local variables occurring in a negative context are required to be functionally-dependent on global variables. Therefore, it is safe to push them out of the negative context and treat them as implicitly existentially quantified global variables.

The translation process can thus be seen as a series of unfolding and simplifying steps, which results in a (large) conjunction of (possibly reified) simple Boolean, integer, float and set constraints. All variables are global and implicitly existentially quantified, integer variables are range-restricted, and set variables have a fixed (finite) domain of integers. The translated model also contains `element` constraints, which arise from array access using variable indices, and lexicographic ordering constraints, which arise from the translation of ordering on complex types like tuples. This range-restricted, existentially quantified, conjunction of constraints can be understood as the operational semantics of the Zinc model instance.

6.2. Dependence on the Underlying Solver(s)

Although Zinc has been designed as a solver-independent language, in practice it must be implemented on top of existing solvers. This poses a challenge, as different solvers have different capabilities. For example, most FD solvers only support integer variables over a range over values (although some are able to represent ∞ and $-\infty$ distinctly), and the behaviour of operations that overflow these ranges varies between solvers. Similarly, solvers that support floats provide varying ranges

and levels of precision. More strikingly, some commercial linear solvers occasionally return incorrect results!

One possible solution to this is to mandate a “minimum” interface that a solver must support in order for it to be used with Zinc. However, designing this interface is difficult because there are so many different and varied solvers.

With Zinc we take a more pragmatic approach. The base Zinc language is specified in as much detail as possible, but there are some inevitable “holes” where its semantics depend on the underlying solver. Therefore, a particular Zinc implementation should provide a clear specification of how it implements these “holes”, and any ways in which it may fall short of the base Zinc specification. We intend to provide such a specification with our G12 implementation of Zinc (see Section 6.5).

Although this situation is not ideal, it is unavoidable at present, due to the inconsistencies in solver interfaces and capabilities. In practice, we anticipate that in many cases it will not be a problem. However, it is likely that advanced Zinc users will have to understand the limits of the underlying solver(s) in order to use Zinc to its fullest. This situation is no different to that faced by existing users of solvers.

6.3. Annotations

Zinc allows constraints and variables to be marked with annotations which can contain attributes. These annotations do not change the semantics of the model but can be used to guide the generation of a decision model for a particular solver or solving technique.

Annotations are declared in a manner similar to predicates. For example, the following declares a `solver` annotation which takes one of four possible solver kinds:

```
enum SolverKind = { Lp, Ip, Fd, Sat };
annotation solver(SolverKind);
```

Declared annotations are used by writing ‘`::`’ after the annotated element. For example:

```
constraint (x != y) :: solver(Fd);
```

might indicate to the solver that the disequation should be handled by the `Fd` sub-solver.

Variable constraints might indicate how a variable should be represented. For example, this annotation:

```
annotation bounds;
var int: x :: bounds;
```

might indicate to the solver that the decision variable `x` need only be represented in a form that maintains its bounds.

6.4. Decomposable global constraints

To the modeller, global constraints are simply Zinc predicates defined in standard libraries. This supports translation to different implementations since it allows solver-specific definitions of the library predicates to be used in the translation process: if a solver does not support that particular global constraint, then it can be unfolded using a solver-specific definition in terms of constraints supported by the solver. For example, it is simple to specify `all_different` in terms of disequalities:

```
predicate all_different(array[$U] of $V: a) =
   forall(i, j in index_set(a) where i < j) (a[i] != a[j]);
```

Note that this definition is polymorphic in the array index set and the array elements.

The ability to provide a low-level logical definition of the library global constraints also supports automatic reasoning about Zinc models, and easy experimentation with different decompositions, since the modeller can provide alternate implementations.

Example: Consider the sequence constraint $sequence(l, u, k, [X_1, \ldots, X_n])$ which holds iff $l \leq \sum_{j=i}^{i+k-1} X_i \leq u$ for $1 \leq i \leq n-k+1$. This can be straightforwardly expressed in Zinc as shown by the predicate `sequence_among` defined in Figure 4. This decomposition has the advantage that the local variable is functionally-dependent, and so the predicate can be used in a negative context.

There are other decompositions with different propagation behaviour (see [4] for more details). For example, $S_j = \sum_{i=1}^{j} X_i, 1 \leq j \leq n$ and $l \leq S_{i+k-1} - S_i \leq u, 1 \leq i \leq n-k+1$ which is shown as `sequence_cumul` in Figure 4, gives stronger propagation than `sequence_among` for some inputs. However, it does require non-functionally-dependent local variables and so cannot be used in a negated context. We can combine these two decompositions, to gain the propagation behaviours of both as in `sequence_joint`.

Finally, a domain consistency propagator `sequence_partial_sum` can also be defined, again using local variables. This provides the strongest propagation possible. By appropriately defining `sequence` and any of these different predicates we can experiment with different decompositions. □

6.5. Implementation

We have two implementations of Zinc: a prototype implementation, and a more robust second implementation.

The prototype compiler implements all the language constructs. It is written in Mercury [23] with a Yacc generated parser and flex generated lexical analyser. It is about 12,000 lines of Mercury code, and 5,000 lines of C.

After doing syntax and semantic checking, the prototype compiler reads the data files and generates the SI-FZM instance. Currently, we have developed mappings

```
predicate sequence_among(int: l, int: u, int: k, array[int] of var 0..1: x) =
    let { int: n = max(index_set(x)) } in
    assert(min(index_set(x)) == 1 /\ card(index_set(x)) == n,
        "array x must be indexed 1..n",
    forall(i in 1..n-k+1)(
        among(l, u, [x[j] | j in i..i+k-1])));

predicate among(int: l, int: u, array[int] of var 0..1: x) =
    let { var int: s = sum(x) } in
        l <= s /\ s <= u;

predicate sequence_cumul(int: l, int: u, int: k, array[int] of var 0..1: x) =
    let { int: n = max(index_set(x)),
          array[1..n] of var 0..n: s   } in
    assert(min(index_set(x)) == 1 /\ card(index_set(x)) == n,
        "array x must be indexed 1..n",
    s[1] == x[1] /\
    forall(i in 2..n)(s[i] == x[i] + s[i-1]) /\
    forall(i in 1..n-k+1)(l <= s[i+k-1] - s[i] /\ s[i+k-1] - s[i] <= u));

predicate sequence_joint(int: l, int: u, int: k, array[int] of var 0..1: x) =
    sequence_among(l, u, k, x) /\ sequence_cumul(l, u, k, x);

predicate sequence_partial_sum(int: l, int: u, int: k,
        array[int] of var 0..1: x) =
    let { int: n = max(index_set(x)),
          array[1..n, 1..k] of var 0..k: p } in
    assert(min(index_set(x)) == 1 /\ card(index_set(x)) == n,
        "array x must be indexed 1..n",
    forall(i in 1..n)(p[i, 1] == x[i]) /\
    forall(j in 2..k, i in 1..n-j, m in 1..j-1)(
        p[i, j] == p[i, m] + p[i + m, j - m]) /\
    forall(i in 1..n-k)(l <= p[i, k] /\ p[i, k] <= u));
```

Figure 4. Various decompositions of the sequence constraint.

from SI-FZM to ECLiPSe [28] for three different solving techniques. The first design model uses a complete tree search with propagation based finite domain and set solvers; the second uses a form of local search, in which the local move automatically maintains hard constraints; and the third one uses mathematical techniques to solve a linear version of the model and, for models involving integer variables, it uses standard branch and bound search. Our first experimental results show that there is not a large overhead on the mappings. More details about these mappings can be found in [21].

The second Zinc implementation, currently underway, is intended to be an "industrial strength" implementation that forms part of the G12 constraint solving

system. The G12 implementation of Zinc is written entirely in Mercury and it is currently about 25,000 lines of code. After syntax and semantic checking, it is optionally transformed by an ACD term-rewriting engine that is controlled by a rewriting language called Cadmium. The current back-end is an interpreter which solves the model by using the G12 solvers, although the plan is to eventually convert Zinc to compiled Mercury. In this implementation, the user will have some control over this transformation via Cadmium, which will allow easy experimentation with different solving techniques.

7. Extended Example

We have claimed that Zinc is extensible, i.e., that we can build Zinc modules that effectively define new application-specific constraint languages. In this section we provide an extended example of this, by defining a module mimicking some of the scheduling features defined in OPL.

7.1. Unary Resource Scheduling

Imagine a bricklayer building a house whose work is divided into various activities: building the chimney, each of the north, east, south and west walls, and the garage. The bricklayer can only perform one activity at a time, and so is a unary resource. To model this in Zinc we create an array of activities and constrain them not to overlap in time. The definition of `Activity`, and its associated unary resource scheduling definitions, is provided in Figure 5.

All activities are assumed to occur within a time window from `scheduleOrigin` to `scheduleHorizon` and, hence, we define a type `Time` which defines this set of times. Activities are made up of a start time, an end time and a duration, where the duration is the difference between start and end times. The `precedes` predicate ensures one activity occurs before another.

Resources are not specifically defined since they will result in a single resource constraint. A unary resource (which cannot be shared) is represented as a constraint that takes a list of activities. The given definition of `unaryResource` ensures, in a simple manner, that each activity does not overlap with any other. Certainly, efficient implementations will not use this decomposition, but it provides an executable declarative definition of the constraint.

We can define a periodic break from time $start$, $start + periodicity$, $start + 2 \times periodicity$ until end each of given $duration$ for a unary resource simply as an array of activities using the function `periodicBreak`. We can use it as follows: assuming the bricklayer wants a break of one hour after each three hours of work, this is modelled as:

```
constraint unaryResource(a ++ periodicBreak(3,1,4,scheduleHorizon));
```

where `++` concatenates two arrays.

The specific data for the bricklayer could be as follows:

```
int: scheduleOrigin;
int: scheduleHorizon;

type Time = scheduleOrigin..scheduleHorizon;

type Activity = (
    record(var Time: start,
           var Time: end,
           var 0..(scheduleHorizon - scheduleOrigin): duration
    ): a where a.end == a.start + a.duration);

predicate precedes(Activity: a1, Activity: a2) =
    a1.end <= a2.start;

predicate unaryResource(array[int] of Activity: a) =
    forall(i,j in index_set(a) where i < j)
        (precedes(a[i], a[j]) \/ precedes(a[j], a[i]));

function array[int] of Activity: periodicBreak(int: b_start, int: b_duration,
        int: b_periodicity, int: b_end) =
    [ (start: t, end: t + b_duration, duration: b_duration) |
      t in b_start..b_end
      where t mod b_periodicity == b_start mod b_periodicity ];
```

Figure 5. Types and constraints for scheduling unary resources.

```
  Activity: chimney = (_,_,4);
  Activity: north   = (_,_,2);  Activity: south = (_,_,2);
  Activity: east    = (_,_,2);  Activity: west  = (_,_,2);
  Activity: garage  = (_,_,3);
  array[int] of Activity: a = [chimney, north, east, south, west, garage];
  constraint unaryResource(a);
```

7.2. Reservoir Resource Scheduling

For more complicated resources like *reservoirs* (see Figure 6) we introduce the concept of a *usage*, which we will model with the record type `Usage`. There are four kinds of usage by an activity: an activity *requires* an amount of some resource if it needs the amount during its entire duration; it *consumes* an amount of resource if it uses it permanently from its start time; it *provides* an amount of resource if this is available within its duration; and *produces* an amount of resource if this is available permanently from its end time. In order to further control resource usage we add the `UsageMinMax` type which models a minimum or maximum usage over some time period.

The `reservoirResource` predicate defines the resource constraints for a reservoir: it has initial amount `init`, a maximum capacity `capacity`, and its usage is

determined by the `u` and `umm` arrays. Again the definition provides an executable specification of the meaning of the reservoir constraint. It ensures that at each time point in the horizon of the reservoir that the capacity limits are satisfied, and the usage is less than capacity.

7.3. State Resource Scheduling

For state resources we can separate out the individual demands for a resource since they are all independent constraints on the resource variables. Code for state resource constraints is shown in Figure 7. An activity can require a particular state during its lifetime, or require that the state is within a set of states, or excludes the use of some state or some set of states during its lifetime. We can make use of parametric polymorphism to use enumerated types for the state type.

An example of the usage of state resource constraints is shown in the following code, which specifies an oven state resource (which can be hot, warm or cold) and three activities: a, b and c, which respectively require a warm oven, require a not cold oven and exclude a not cold oven (equivalently demand a cold oven) over their lifetime:

```
enum OvenStates = { hot, warm, cold };
set of OvenStates: notCold = { hot, warm };
array[scheduleOrigin..scheduleHorizon] of var OvenStates: oven;
Activity: a; Activity: b; Activity: c;
constraint requiresState(a, warm, oven);
constraint requiresAnyState(b, notCold, oven);
constraint excludesAnyState(c, notCold, oven);
```

8. Conclusion

We have presented a new modelling language Zinc, and discussed the main design decisions and innovations that allow Zinc to meet its four aims: (1) support for natural modelling, so that conceptual models can be specified in a high-level, mathematical fashion; (2) support for extensible modelling, so that new application domains can be modelled easily; (3) support for good software engineering, so that model correctness and maintainability is made easier; and (4) practical solver-independence, so that conceptual models can be transformed into design models utilizing various solving techniques including local search, mathematical modelling, constraint programming, or a combination of the above.

Zinc continues the trend in modelling languages of increasing expressiveness and moving closer to a generic programming language and/or specification language. Most of the interesting design decisions result from trying to obtain the flexibility of a modern programming language with data and procedural abstraction, providing model and instance time checking, while still keeping the natural, high-level syntax of a modelling language, and the ability to solve the models using practical solving techniques. We hope that Zinc will become a widely used modelling language. However, regardless of its success, we believe the detailed discussion of various

```
enum UsageKind = { requires, consumes, provides, produces };

type Usage = record(UsageKind: kind, Activity: a, var int: amount);

enum UsageMinMax = { min_used(int: start, int: end, int: limit),
                     max_used(int: start, int: end, int: limit) };

predicate reservoirResource(int: capacity, int: init, array[int] of Usage: u,
                            array[int] of UsageMinMax: umm) =
    let { array[Time] of var 0..capacity: cap,
          array[Time] of var 0..capacity: used } in
    cap[scheduleOrigin] == init + sum(i in index_set(u))(
        case u[i].kind {
            produces --> bool2int(u[i].a.end == scheduleOrigin) * u[i].amount,
            provides --> bool2int(u[i].a.start <= scheduleOrigin /\
                                  u[i].a.end > scheduleOrigin) * u[i].amount,
            requires --> 0,
            consumes --> 0 }
    ) /\
    forall(t in scheduleOrigin+1 .. scheduleHorizon)(
        cap[t] == cap[t-1] - used[t-1] + sum(i in index_set(u))(
            case u[i].kind {
                 produces --> bool2int(u[i].a.end ==  t) * u[i].amount,
                 provides --> bool2int(u[i].a.start <= t /\
                                       u[i].a.end > t) * u[i].amount,
                 requires --> 0,
                 consumes --> 0 })
    ) /\
    forall(t in scheduleOrigin .. scheduleHorizon-1)(
        used[t] = sum(i in index_set(u))(
            case u[i].kind {
                 produces --> 0,
                 provides --> 0,
                 requires --> bool2int(u[i].a.start <= t /\
                                       u[i].a.end > t) * u[i].amount,
                 consumes --> bool2int(u[i].a.start == t) * u[i].amount }) /\
        used[t] <= cap[t]
    ) /\
    forall(j in index_set(umm))(
        case umm[j] { min_used --> forall(t in umm[j].start .. umm[j].end)(
                                       used[t] >= umm[j].limit),
                      max_used --> forall(t in umm[j].start .. umm[j].end)(
                                       cap[t] <= umm[j].limit) }
    );
```

Figure 6. Types and constraints for scheduling reservoir resources.

```
predicate requiresState(Activity: a, $T: state, array[int] of any $T: r) =
    forall(t in index_set(r)) (
        (a.start <= t /\ t < a.end) -> r[t] == state
    );

predicate excludesState(Activity: a, $T: state, array[int] of any $T: r) =
    forall(t in index_set(r)) (
        (a.start <= t /\ t < a.end) -> r[t] != state
    );

predicate requiresAnyState(Activity: a, set of $T: states,
        array[int] of any $T: r) =
    forall(t in index_set(r)) (
        (a.start <= t /\ t < a.end) -> r[t] in states
    );

predicate excludesAnyState(Activity: a, set of $T: states,
        array[int] of any $T: r) =
    forall(t in index_set(r)) (
        (a.start <= t /\ t < a.end) -> not (r[t] in states)
    );
```

Figure 7. Types and constraints for scheduling state resources.

design alternatives in this paper will be of interest to designers of future modelling languages and to programming language theorists.

Our two main goals for the future are to finish the industrial strength G12 implementation of Zinc and to investigate how to allow the modeller to augment a model with solver-specific search routines. Rather than specifying search directly in Zinc, we plan to provide a fixed set of parametric search routines, each of which captures a common search pattern. When using a particular search pattern, a modeller will specify Zinc functions for the search routine parameters. For example, in a backtracking search pattern, the routines will specify which variables and values to branch on. Our initial research into this for backtracking search and local search is extremely promising.

Acknowledgements

We thank the members of the G12 team at National ICT Australia for helpful discussions about Zinc, in particular Peter Baumgartner, Ralph Becket, Sebastian Brand, Michael Maher, John Slaney, and Toby Walsh, and the anonymous reviewers for their helpful feedback.

References

1. Solvers that work with AMPL. `http://www.ampl.com/solvers.html`.
2. K. Apt and M. Wallace. *Constraint Logic programming Using ECLiPSe*. Cambridge University Press, 2007.
3. J. P. Bowen. *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press, 1996.
4. S. Brand, N. Naroditskaya, C.-G. Quimper, P. J. Stuckey, and T. Walsh. Encodings of the sequence constaint. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP2007)*, Providence, Rhode Island, USA, September 2007.
5. B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. An overview of HAL. In J. Jaffar, editor, *Proceedings of the Fourth International Conference on Principles and Practices of Constraint Programming*, LNCS, pages 174–188. Springer-Verlag, October 1999.
6. DIMACS. Satisfiability suggested format. `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc`, 1993.
7. P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *LOPSTR*, pages 214–232, 2003.
8. R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 2002.
9. A. Frisch, M. Grum, C. Jefferson, B. Martinez Hernandez, and I. Miguel. The design of ESSENCE: A constraint language for specifying combinatorial problems. In *Proc. of the 20th International Joint Conference on Artificial Intelligence IJCAI*, 2007.
10. A.M. Frisch, C. Jefferson, B. Martinez-Hernandez, and I. Miguel. The rules of constraint modelling. In *Proc 19th IJCAI*, pages 109–116, 2005.
11. A.M. Frisch, C. Jefferson, B. Martinez-Hernandez, and I. Miguel. The design of Essence: A constraint language for specifying combinatorial problems. *Constraints*, This volume, 2008.
12. C. Gervet. *Large scale combinatorial optimization: A methodological viewpoint*, volume 57 of *Discrete Mathematics and Theoretical Computer Science*, pages 151–174. DIMACS, 2001.
13. A. Hakin. The intractability of resolution. *Theoretical Computer Science*, 39:297 – 308, 1985.
14. B. Hnich, I. Miguel, I. P. Gent, and T. Walsh. CSPLib: a problem library for constraints. `http://www.csplib.org/`.
15. Yuen B. J. and Richardson K. V. Establishing the optimality of sequencing heuristics for cutting stock problems. *European Journal of Operational Research*, 84:590–598, 1995.
16. Bharat Jayaraman and Pallavi Tambay. Modeling engineering structures with constrained objects. In *PADL*, pages 28–46, 2002.
17. J. Kallrath. *Modeling Languages in Mathematical Optimisation*. Kluwer Academic, 2004.
18. J-L. Lauriere. ALICE: A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29 – 127, 1978.
19. K. Marriott, P.J. Stuckey, and M. Wallace. *Handbook of Constraint Programming*, chapter Constraint Logic Programming, pages 409–452. Elsevier, 2006.
20. B. A. Murtagh. *Advanced Linear Programming*. McGraw-Hill, 1981. See pages 163-170.
21. R. Rafeh, M.J. Garcia de la Banda, K. Marriott, and M. Wallace. *From Zinc to Design Model*, pages 215–229. Number 4354 in LNCS. Springer, 2007. Proc. PADL 2007.
22. S. Schneider. *The B-Method: An Introduction*. Cornerstones of Computing. Palgrave, 2001.
23. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.
24. M. Stefik. Planning with constraints (MOLGEN: Part 1). *Artificial Intelligence*, 16:111–139, 1981.
25. P. J. Stuckey, M. J. García de la Banda, M. J. Maher, K. Marriott, J. K. Slaney, Z. Somogyi, M. Wallace, and T. Walsh. The G12 project: Mapping solver independent models to efficient solutions. In *CP*, pages 13–16, 2005.
26. P. Van Hentenryck, I. Lustig, L.A. Michel, and J.-F. Puget. *The OPL Optimization Programming Language*. MIT Press, 1999.

27. P. Van Hentenryck and L. Michel. *Constraint-Based Local Search.* MIT Press, 2005.
28. M. Wallace, S. Novello, and J. Schimpf. ECLiPSe - a platform for constraint programming. *ICL Systems Journal*, 12(1):159–200, 1997.
29. E. W. Weisstein. Perfect square dissection. From MathWorld—A Wolfram Web Resource. `http://mathworld.wolfram.com/PerfectSquareDissection.html`, 1999.
30. J. Yeomans. Solving Einstein's riddle using spreadsheet optimisation. *INFORMS Transactions on Education*, 3(2), 2003.