Data-Independent Type Reduction for Zinc

LESLIE DE KONINCK, SEBASTIAN BRAND,
JULIEN FISCHER and PETER J. STUCKEY

National ICT Australia, Victoria Research Laboratory
Dept. of Computer Science & Software Engineering, University of Melbourne
Australia
E-mail: {`lesliedk,sbrand,juliensf,pjs`}`@csse.unimelb.edu.au`

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Zinc is a high-level logical modelling language with support for decision variables of complex types. Current constraint solvers only support decision variables of a subset of the types Zinc offers. Therefore, these variables and the constraints that involve them need to be reduced to a form that is supported by the target solvers. In this paper, we show how to reduce the type of decision variables in a systematic way, independent of a model's instance data. We use the Cadmium model transformation language for a concise and modular implementation of these transformations.

1 Introduction

Zinc (Marriott et al. 2008) is a modern high-level logical modelling language. It offers decision variables of a wide variety of types as well as constraints that operate on them. As a consequence, models in Zinc can be compact and close to their natural language description, and designing and maintaining models is considerably simpler.

A Zinc model typically needs to be converted to a simpler, equivalent model before it can be solved by current solvers because they only support a subset of the variable types that Zinc offers. This means decision variables of types that are not supported need to be represented by variables of supported types, and constraints over variables of unsupported types need to be represented by supported constraints. We call this *type reduction*. Which variable types need to be eliminated depends on the solver designated to handle the corresponding constraints. For instance, a Finite Domain (FD) solver may support set variables, whereas Mixed Integer Programming (MIP) solvers do not. Therefore, we want to be able to run different type reduction transformations, independently of each other.

In the G12 implementation of Zinc (G12 Team 2010), we do type reduction by means of Zinc-to-Zinc transformations written in Cadmium, a purpose-designed, rule-based model transformation language (Duck et al. 2008). We support the following reductions:

- from an enumeration type to an integer range
- from a record or tuple to its components
- from a set of finite element type to a set of integers

- from a set of finite element type T to a Boolean array, indexed by T.

The latter two reduce the same type, but to a different target type. The first of them can be used for solvers that support integer set variables only, the second for solvers that do not support set variables regardless of the element type. After type-reducing these variable types away, the only remaining Zinc variable types are Boolean, integer, float, and potentially integer set variables.[1]

Our type reduction transformations have the following significant features:

Zinc-to-Zinc: the result of the transformations is again Zinc. This allows type reduction to be followed by or combined with other Zinc transformations.

Data-independence: the transformations are independent of parameter values. Instance data need not be available at transformation time but can later be supplied in data files.

Transparency: the original types remain available for input and output. Input values of types not supported by the solver in use are reduced to values of supported types. After solving, solution values of a reduced type are converted back, so the solution is returned using the original types.

Locality: type reduction only changes those parts of the model that contain variables of an unsupported type. It does not require flattening of the entire model and deals with user-defined functions without requiring inlining.

High-level: the type reduction transformations are implemented using the model transformation language Cadmium. This makes their development, maintenance and extension comparatively easy in our experience.

The next section introduces Zinc, Cadmium, and the process of type reduction; Section 3 shows how we can type-reduce set variables; Section 4 presents the step-wise process we use to type-reduce a model; in Section 5 we give some empirical evaluation; Section 6 discusses related work and Section 7 concludes.

2 Preliminaries

2.1 Zinc

A Zinc model consists of a set of items for variable/parameter declaration, constraints, solving objective and output. A variable or parameter declaration has the form T: x or T: $x = A$, where x is the name of the variable, T is its type-inst and expression A is its optional assignment.[2] A *type-inst* is the combination of a type (a set of values) and an instantiation pattern, which determines which components of a variable are fixed when solving starts. Examples of type-insts are `int`, `var` 0..1, `tuple(bool, var float)` and `array[int] of var set of` 1..9, representing respectively an integer parameter, a binary (decision) variable, a tuple whose first and second field are a Boolean parameter and a float variable respectively, and an

[1] Our current implementation does not yet handle non-flat enumerations, this is ongoing work.
[2] A parameter must be assigned but the assignment can be in a separate data file.

integer indexed array whose elements are set variables taking elements from 1..9. The distinction between variables and parameters via the instantiation pattern in the type is crucial for data-independent transformation and compilation of Zinc models.

A constraint item holds a constraint. A solve item states whether the aim of solving is satisfaction, or optimisation in which case it also holds the objective function. An output item states how a solution to the problem should be presented. There are other types of items; see (Marriott et al. 2008) for more details.

Example 1
Here is a simple Zinc model:

```
int: c;                      % declare an integer parameter c
var set of 1..3: s;          % declare a set variable subset of {1,2,3}
constraint card(s) = c;      % enforce that the cardinality of s is c
solve satisfy;               % search for any solution
output ["s = ", show(s)];    % output the resulting set

c = 2;
```

The last line defines the parameter `c`. It could be in a separate data file. □

2.2 Cadmium

Our Zinc transformations are implemented in Cadmium (Duck et al. 2008), a term rewriting language supporting context sensitive rewriting and associative / commutative operators. Cadmium rules have the form CC `\` H `<=>` G `|` B. Such a rule states that a term matching H rewrites into B given that its *conjunctive context* contains terms matching CC and the guard term G rewrites into `true`. The conjunctive context of a term T consists of the terms that are conjunctively conjoined with T or a superterm of T. If no reference to the conjunctive context is needed then CC and the backslash (\) are omitted. If no guard is needed then G and the `|` symbol are omitted.

Cadmium rewrites terms; therefore Zinc models need to be represented as a term. A carefully designed term representation of Zinc exists that aims to be unambiguous, compact, and easy to use at the same time. For instance, it ensures that all variable declarations that hold at a given expression are in the conjunctive context of that expression. For example, the following rule evaluates the Zinc `index_set` function on an array whose declaration can be found in the conjunctive context:

```
decl(array(IT, _), X, _, _, _) \ app("index_set", _, [id(X)]) <=>
        is_finite_type_inst(IT) | type_to_set(IT).
```

The `decl`/5 term represents a variable/parameter declaration with fields for type-inst, name, assignment, identifier kind, and annotations. The `app`/3 term represents a predicate/function call. The `is_finite_type_inst`/1 guard rewrites to `true` if its argument is a finite type-inst, and `type_to_set`/1 rewrites a finite type to the set of its values.

To improve the presentation of rules in this paper, we will use a variation of standard Zinc syntax, using slanted type to distinguish it from Cadmium terms. Here is the above rule in simplified syntax:

$$\textit{array}[\texttt{IT}]\ \textit{of}\ \_\!:\texttt{X} = \_ \setminus \textit{index\_set}(\texttt{X}) \ \Leftrightarrow$$
$$\texttt{is\_finite\_type\_inst}(\texttt{IT}) \mid \texttt{type\_to\_set}(\texttt{IT}).$$

2.3 Evaluation Stages

In our implementation of Zinc, models are compiled into executables. An executable when run first reads any instance data files, then processes the input data, solves the problem and finally generates output.

Input processing consists of computing derived parameter values, creating variables and setting up constraints. Since data files can contain parameters of arbitrary Zinc type, the complete Zinc language needs to be supported for parameter computations. This also holds for output generation, where computations on the then fixed values of variables can take place.

Constraints can be formulated using variables of any type-inst supported by Zinc. Type reduction limits the variable types that solvers need to support by reducing variables of complex types into ones of simpler types, and converting between original and reduced variables in the input and output processing stages.

Example 2
For use with a solver that does not support set variables, we can reduce the model of Example 1 to the following:

```
int: c;
array[1..3] of var bool: s;                             % changed
constraint sum(e in 1..3)(bool2int(s[e])) = c;          % changed
solve satisfy;
output ["s = ", show({e | e in 1..3 where fix(s)[e]})];   % changed
```

The set variable `s` is reduced to a Boolean array. The expression `card(s)` is converted into an equivalent expression using the reduced representation of `s`. In this case, the expression is the sum of `bool2int` (which maps `true` to 1 and `false` to 0) applied to each array element. The occurrence of `s` in the output statement is replaced by an expression that converts the reduced value of `s` to the corresponding value of its original type. In particular, it converts the Boolean array representation to a set by means of a set comprehension, taking those values from the array's index set for which array lookup evaluates to `true`. Note that array `s` is fixed at this point so we can apply the `fix` function to use `s` where a fixed value is expected by Zinc (`where` clauses in comprehensions need to be fixed). □

In the preceding example, we did not need to change the types of input parameters. The following is an example where we do.

Example 3
Consider the following model:

```
enum e;              % declare an enumeration type    (defined in data file)
e: p;                % declare a constant p of type e (defined in data file)
var e: v;            % declare a variable v of type e
constraint p = v;    % enforce p = v
solve satisfy;       % search for any solution
output [show(v)];    % output the result
```

Assume a solver that does not support enumeration variables. The enumeration type `e` and parameter `p` are declared in the model but defined in a data file.

```
enum e;              % declare an enumeration type    (defined in data file)
e: p;                % declare a constant p of type e (defined in data file)
1..card(e): p_reduced = inverse_lookup(p, e);    % type-reduced p
var 1..card(e): v;           % type-reduced v as integer
constraint p_reduced = v;    % enforce p = v
solve satisfy;               % search for any solution
output [show(e[fix(v)])];    % output result by looking up entry in enum
```

Because the data file still refers to parameter `p`, we need to use a new name for the reduced version: `p_reduced`. In Zinc, the name of an enum (e.g. `e`) can be used as an expression that stands for the set of its cases. A set can in turn be coerced into an array. The function `inverse_lookup`(v, a) returns the (first) position of v in array a. Input processing converts parameter `p` into `p_reduced` which has reduced type `1..card(e)`. The reduced type is given in a data-independent way, i.e. independent of the definition of `e`. The value of `p_reduced` is the position of `p` in the ordered set of cases of `e`; e.g. if `e` $= \{$`red`, `green`, `blue`$\}$ and `p` $=$ `green`, then `p_reduced` $= 2$. In the output statement, the value of the reduced variable `v` is converted into the corresponding enum value. The equality constraint between enums becomes an equality constraint between integers. □

3 Encoding Type-Reduced Set Variables in Zinc

This section presents some alternative encodings for set variables in Zinc. Similar encodings have been proposed before, see e.g. (Walsh 2003), but their data-independent representation in Zinc is novel. We note that because of the data independence, using these reduced representations does not fundamentally increase the model size.

3.1 Reducing the Element Type of Set Variables

Zinc supports set variables of any finite type, i.e. involving nested tuples, records, sets and arrays. Solvers for set variables typically only support them for restricted element types, most notably integer ranges. Therefore, one aspect of type reduction for set variables concerns the set element type: if it is not supported as element type, we can reduce it to an integer range. Every finite type can be represented as the set of values of its base type. E.g. `bool` $= \{$`false`, `true`$\}$ and `set of` $1..2 = \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$. A simple conversion from any finite type to an integer range is made by mapping each value to its position in the ordered list of all values.

However, in general, we may need to perform the conversion on variables of the set element type, and in such cases it may be beneficial to use a specialised conversion function that leads to more propagation. With this encoding, it is up to the solver to choose a suitable low-level representation for the resulting set variable.

3.2 Eliminating Set Variables

If set variables are not supported by the target solver, they need to be reduced to a type that is supported. The two main alternative encodings for a variable of type-inst `var set of` T (shown below), are as an array of Boolean variables, and as an array of set elements. The latter is useful if we have a (non-trivial) bound on the cardinality of the set.

$$\texttt{array}[T]\ \texttt{of var bool} \tag{1}$$

$$\begin{aligned}(\texttt{tuple}(\texttt{var}\ l..u, \texttt{array}[1..u]\ \texttt{of var}\ T) : \texttt{t where forall}(\texttt{i in}\ 1..u)\\ ((\texttt{i} < \texttt{t}.1 \rightarrow \texttt{t}.2[\texttt{i}] < \texttt{t}.2[\texttt{i}+1]) \wedge (\texttt{i} > \texttt{t}.1 \rightarrow \texttt{t}.2[\texttt{i}] = d)))\end{aligned} \tag{2}$$

For the second representation, we assume l and u are respectively a lower and an upper bound on the cardinality of the set. We define an array of size u whose first c elements are the elements in the set variable, where c is the actual cardinality of the set variable. We use a *constrained type* to furthermore enforce that the elements in the array are ordered so as to remove symmetries (first implication) and to set the unused elements of the array to a dummy value d to avoid searching on these variables (second implication).

Let x be a variable of type-inst `var set of` T, and let x_1 and x_2 be variables of the type-insts of respectively the first and second representation. We link x to x_1 and x_2 using the following constraints:

$$\texttt{forall}(\texttt{i in}\ \textit{type\_to\_set}(T))(\texttt{i in}\ x \leftrightarrow x_1[\texttt{i}])$$

$$\texttt{card}(x) = x_2.1 \wedge \texttt{forall}(\texttt{i in}\ 1..u)(\texttt{i} \leq \texttt{card}(x) \rightarrow x_2.2[i]\ \texttt{in}\ x)$$

Note that the second representation requires that decision variables of the set element type are supported; we cannot use it for sets of strings, for instance. In our current implementation, we only use the first representation, but we plan to support the second one in future work.

4 Data-Independent Type Reduction

In this section, we present the process of type reduction. We start this process by type-reducing variables and function definitions. Then, we type-reduce expressions. The whole process is deterministic as we currently only use one encoding scheme. In the future, we may add alternative encodings, which will be selected by means of annotations.

4.1 Parameters and Variables

Parameters or decision variables may need to be type-reduced to eliminate unsupported types from the constraints. A compound type-inst, such as a tuple or array, may be unsupported because of its components. For instance, a solver may support arrays of integers but not of enums.

Assigned values are reduced by applying a conversion function. Values of a compound type-inst that is not fully supported are deconstructed, have their unsupported components reduced, and are then reconstructed again. In Zinc, this can be done in a functional way without using constraints.

Example 4
Let `e` be an enumeration type in the following declarations:

```
e: p1 = x1;
tuple(e, int): p2 = x2;
set of e: p3 = x3;
array[e] of e: p4 = x4;
```

Eliminating `e`, they are reduced to

```
1..card(e): p1 = inverse_lookup(x1, e);
tuple(1..card(e), int): p2 = (inverse_lookup(x2.1, e), x2.2);
set of 1..card(e): p3 = { inverse_lookup(x, e) | x in x3 };
array[1..card(e)] of 1..card(e): p4 = [ inverse_lookup(x4[x], e) | x in e ];
```
□

When reducing a variable x of type-inst T, we reduce its assignment if present and replace its occurrences in constraints by $\mathit{unreduce}_T(V)$ :: `type_reduced_as`(V) where $\mathit{unreduce}_T$ converts a value of the reduced type-inst to its corresponding value of original type-inst T. The :: operator separates a term and its annotation. The `type_reduced_as/1` annotation contains the reduced expression (i.e. V). In practice, we also keep track of how an expression is reduced in the annotation, i.e. which components of the expression are reduced (if it has a compound type-inst such as tuple or array), and which representation is used for the reduced value. We have omitted these details from the annotation to simplify the presentation. Note that after type-reducing variables, we have a valid Zinc model with the same semantics as the original.

Example 5
Given the model fragment

```
var set of 1..3: s;
constraint card(s) > 1;
```

type reduction of set variables results in

```
array[1..3] of var bool: s;
constraint card(new2old(s) :: type_reduced_as(s)) > 1;
```

We define the following conversion functions:

```
function var set of $T: new2old_var(array[$T] of var bool: a);
function array[$T] of var bool: old2new_var(var set of $T: s);
function set of $T: new2old_par(array[$T] of bool: a) =
                                        { e | e in index_set(a) where a[e] };
function array[$T] of bool: old2new_par(set of $T: s) = [ e: true | e in s ];
```

The function `new2old_var/1` is declared but not defined. It only serves to keep the model valid during type reduction and all calls to it are removed when type reduction finishes. Function `old2new_var/1` is its inverse and is used similarly; both simplify away using `old2new_var(new2old_var(`X`))` $= X$. Function `new2old_par/1` is used to convert the values of reduced set variables back to set values for output after solving. Finally, `old2new_par/1` is used to convert fixed sets to their reduced representation in contexts where a set decision variable is expected. □

A type may be reduced in different ways depending on the context; cf. Section 3.2. The best representation may depend on the specific solvers used and on the constraints in which the variable appears. Furthermore, we could also support multiple representations and link them via channelling constraints (Cheng et al. 1999), although currently we only support a single representation for each variable type.

The above transformation is applied to global and local variables and parameters, comprehension generators and variables used in defining constrained type-insts as part of a variable declaration being reduced. It is not applied to function arguments (see further) and to variables that are introduced during the transformation.

4.2 Predicate and Function Definitions

Zinc supports user-defined functions and predicates (functions with a return type-inst of `var bool`). They may apply to expressions of an unsupported type-inst. We create reduced versions of such functions and replace calls accordingly.

Example 6
Consider the following function, which is polymorphic in the element type of the set arguments:

```
function var set of $T: set_op(var set of $T: s1, var set of $T: s2) =
    if card(ub(s1)) > card(ub(s2)) then s1 diff s2 else s2 diff s1 endif;
```

Here, `ub/1` returns the upper bound of the variable that is its argument. One could inline the above function definition in a call such as `set_op(`s_1, s_2`)`. However, inlining cannot always be used, in particular when dealing with higher-order functions (`foldl` and `foldr`), e.g. in the call `foldl(set_op,{},` a`)` where the array a is only known once the instance data is available. □

Type-reducing a function consists of reducing the formal arguments and result type-inst, followed by reducing the body expression. The reduction of formal arguments is essentially the same as the reduction of variables and parameters described earlier. We only reduce the formal arguments of the reduced versions of functions, and we keep the original function definitions to ensure the model is valid Zinc at every stage of the type reduction process. We annotate the reduced version of a function to link it with its original. The type reduction of expressions is described in Section 4.3 below.

Example 7
The result of the first step of type-reducing set_op/2 is shown below. For brevity, we have merged the conversions from reduced arguments $\mathtt{s}_1$ and $\mathtt{s}_2$ to their respective original values using local variables.

```
function array[$T] of var bool: set_op′(array[$T] of var bool: s1,
                        array[$T] of var bool: s2) :: type_reduction_of(set_op) =
    let {var set of $T: s′1 = new2old_var(s1), var set of $T: s′2 = new2old_var(s2)}
    in old2new_var(if card(ub(s′1 :: Ann1)) > card(ub(s′2 :: Ann2))
                   then s′1 :: Ann1 diff s′2 :: Ann2
                   else s′2 :: Ann2 diff s′1 :: Ann1 endif);
```

Here, Ann_i stands for type_reduced_as($\mathtt{s}_i$) for $i \in \{1, 2\}$. □

4.2.1 Function Specialisation

One difficulty with type-reducing functions is that we sometimes treat fixed and unfixed values differently, e.g. set variables are reduced but set parameters are not. This means that expressions whose type-insts match the same type-inst variable may no longer do so after type reduction.

Example 8
Consider the following function and declarations

```
function var bool: my_eq(any $T: a1, any $T: a2) = (a1 = a2);
var set of 1..3: s1;
set of 1..3: s2;
```

then my_eq($\mathtt{s}_1$, $\mathtt{s}_2$) is a valid expression, but after type-reducing $\mathtt{s}_1$ into a Boolean array, the same expression is no longer valid: while the original type-insts of $\mathtt{s}_1$ and $\mathtt{s}_2$, namely var set of 1..3 and set of 1..3, resp., both match type-inst variable any $T, the same does not hold for their new type-insts, array[1..3] of var bool and set of 1..3. □

To deal with this problem, we specialise such functions for each combination of (base) type-insts for which they are called. So for instance, if there is a call my_eq($\mathtt{s}_1$, $\mathtt{s}_2$), then we create the following specialised version of my_eq/2:

```
function var bool: my_eq(var set of $T: a1, set of $T: a2) = (a1 = a2);
```

which is subsequently reduced.

4.2.2 Empty Sets

A related problem is that in Zinc there is a distinction between explicitly and implicitly indexed arrays, based on the finiteness of the array index type. If an array is indexed by a finite type, then its index set must exactly be this type. This becomes an issue when coercing the empty set into a set variable of some element type. In case we coerce the empty set into a set variable of finite element type, then the type-reduced value must be an array indexed by all values of this finite type where each element equals false. However, if we coerce the empty set to a set variable of infinite element type then we can return an empty array. E.g. a coercion of {} into type-inst `var set of bool` becomes [`false`:`false`, `true`:`false`] whereas a coercion of {} into type-inst `var set of int` becomes [ ].

In function definitions, we may see a coercion from the empty set into a `var set of` $\$T$ with $\$T$ a type-inst variable. Since we do not know the value of $\$T$, we do not know how to convert the empty set to an array. Therefore, we also need to specialise functions here, for each call pattern of functions in which an empty set is coerced into a set variable whose element type is given by a type-inst variable.

4.3 Expressions

We apply a bottom-up approach to type-reduce constraint expressions. We first annotate expressions of an unsupported type-inst that do not contain a proper such subexpression. This is similar to how we annotate (and revert the reduction of) variables of unsupported type-insts. The annotation again contains a reduced version of the expression. It principally concerns the following kinds of expressions:

- the (implicit) coercion of an expression of supported type-inst, to an unsupported one, e.g. a set parameter used in a position where a set variable is expected, such as in a function application, or a coercion of a tuple to a record if we do not support records;
- the application of a user-defined function whose return type-inst is unsupported, but whose arguments all have supported type-insts;
- the application of a built-in that returns a varified[3] version of one of its arguments: an array lookup where the index is not fixed, or the minimum or maximum of two or more expressions, where at least one of them is not fixed; note that such built-ins cannot be defined in a polymorphic and type-correct way in Zinc as the result type-inst is varified and not all type-insts can be.

Example 9 (*Minimum*)
Consider the expression `min`(t_1, t_2) in a context with the following variable declarations:

```
var int: i;
tuple(var int, set of int): t1 = (i, {1, 3});
tuple(var int, set of int): t2 = (0, {2, 4});
```

[3] A type-inst is varified by making all its fixed components unfixed. E.g. varifying type-inst `tuple(var int, set of int)` results in `tuple(var int, var set of int)`.

Assume that $\mathtt{t}_1$ and $\mathtt{t}_2$ both have supported type-insts, but $\mathtt{min}(\mathtt{t}_1, \mathtt{t}_2)$ does not (it has type-inst `tuple(int, var set of int)`). First, we varify and reduce $\mathtt{t}_1$ and $\mathtt{t}_2$, which means that the second field of each tuple is converted into a Boolean array. We ensure that both are reduced in exactly the same way, that is, the resulting arrays have the same index sets, so that we can use the standard comparison operations to determine which of $\mathtt{t}_1$ and $\mathtt{t}_2$ is the smallest. Unfortunately, we cannot just apply `min`/2 to the reduced versions of $\mathtt{t}_1$ and $\mathtt{t}_2$, as unlike sets, arrays are not varifiable, so instead, we produce the following:

$$\begin{array}{l}
\mathtt{let}\ \{\mathtt{tuple}(\mathtt{int},\ \mathtt{array}[1..4]\ \mathtt{of\ var\ bool})\text{: } \mathtt{t}_1' = (\mathtt{t}_1.1,\ [\ \mathtt{e\ in\ t}_1.2 \mid \mathtt{e\ in}\ 1..4\ ]),\\
\quad\quad \mathtt{tuple}(\mathtt{int},\ \mathtt{array}[1..4]\ \mathtt{of\ var\ bool})\text{: } \mathtt{t}_2' = (\mathtt{t}_2.1,\ [\ \mathtt{e\ in\ t}_2.2 \mid \mathtt{e\ in}\ 1..4\ ]),\\
\quad\quad (\mathtt{tuple}(\mathtt{var\ int},\ \mathtt{array}[1..4]\ \mathtt{of\ var\ bool})\text{: } \mathtt{x\ where}\\
\quad\quad\quad\quad\quad\quad\quad\quad (\mathtt{x} = \mathtt{t}_1' \vee \mathtt{x} = \mathtt{t}_2') \wedge \mathtt{x} \le \mathtt{t}_1' \wedge \mathtt{x} \le \mathtt{t}_2')\text{: } \mathtt{m}\}\ \mathtt{in\ m}
\end{array}$$

i.e. the result is a variable of constrained type-inst that is constrained to be smaller than or equal to the reduced versions of $\mathtt{t}_1$ and $\mathtt{t}_2$, and equal to either of them. □

Example 10 (*Element Constraint*)
An array lookup with variable index (element constraint) varifies the array element type-inst. As such it is similar to the minimum example given above. Consider the expression $[\{1,2\},\{2,3\}][i]$ where i is a variable of type-inst `var` 1..2. There are various ways to reduce this expression; we give one possibility here:

$$\begin{array}{l}
\mathtt{let}\ \{\mathtt{array}[1..3,\ 1..2]\ \mathtt{of\ var\ bool}\text{: } \mathtt{a} =\\
\quad\quad [\,|\,\mathtt{true},\ \mathtt{false}\,|\,\mathtt{true},\ \mathtt{true}\,|\,\mathtt{false},\ \mathtt{true}\,|\,]\}\ \mathtt{in}\ [\ \mathtt{a}[\mathtt{j},\ \mathtt{i}] \mid \mathtt{j\ in}\ 1..3\ ]
\end{array}$$
□

After reducing the base cases, we deal with those expressions that have a proper subexpression that is reduced. Most notably, we need to deal with applications of built-in operators and functions, applications of user-defined functions, and structured term construction and access. For each built-in operator and function, we define what the result should be if it is applied to a reduced expression.

Example 11
The set membership constraint (`in`/2) is converted into an array access. The Cadmium rule for this is:

$$\mathtt{E}\ \mathit{in}\ (\_ :: \mathit{type\_reduced\_as}(\mathtt{New})) \ \Leftrightarrow\ \mathtt{New}[\mathtt{E}].$$

The encoding of the element constraint in Example 10 is especially advantageous if we test for set membership on the resulting expression, e.g. j `in` $[\{1,2\},\{2,3\}][i]$ for some integer variable j, because we can further reduce it into

$$[\,|\,\mathtt{true},\ \mathtt{false}\,|\,\mathtt{true},\ \mathtt{true}\,|\,\mathtt{false},\ \mathtt{true}\,|\,][j,\ i]$$

requiring just a single (two-dimensional) element constraint.

The length of an array whose elements are reduced is just the length of the reduced array:

$$\mathit{length}(\_ :: \mathit{type\_reduced\_as}(\mathtt{New})) \ \Leftrightarrow\ \mathit{length}(\mathtt{New}).$$

The union of two reduced set variables is translated into a call to a type-reduced version of the union operator, called `reduced_union`/2, by the following rule:

$$\textit{function}\ \_\colon \mathtt{F}'(\_,\,\_) :: \textit{type\_reduction\_of}(\mathtt{F}) = \_ \setminus$$
$$\mathtt{F}(\mathtt{O}_1 :: \textit{type\_reduced\_as}(\mathtt{N}_1),\ \mathtt{O}_2 :: \textit{type\_reduced\_as}(\mathtt{N}_2)) \Leftrightarrow$$
$$\mathtt{F}(\mathtt{O}_1,\,\mathtt{O}_2) :: \textit{type\_reduced\_as}(\mathtt{F}'(\mathtt{N}_1,\,\mathtt{N}_2)).$$

The rule applies to a call to binary function F for which a type-reduced version F' is defined, and for which the actual arguments are both type-reduced. The above rule is only a simplified version: the actual rule deals with any arity of function, any combination of arguments being type-reduced, and ensures that the reduced function is the one to be used here (with respect to overloading). The function `reduced_union`/2 is defined as follows:

```
function array[$T] of var bool:
      reduced_union(array[$T] of var bool: a1, array[$T] of var bool: a2) ::
            type_reduction_of(union) =
   [ if x in index_set(a1) intersect index_set(a2) then a1[x] ∨ a2[x]
     else if x in index_set(a1) then a1[x] else a2[x] endif endif
   | x in index_set(a1) union index_set(a2) ];
```

and so (s_1 :: `type_reduced_as`(a_1)) `union` (s_2 :: `type_reduced_as`(a_2)) becomes (s_1 `union` s_2) :: `type_reduced_as`(`reduced_union`(a_1, a_2)). □

4.4 Contexts Expecting a Type-Reduced Value

Expressions occur in a context that expects them to have a certain type-inst. For example, an expression that is the argument of a constraint item should have a type-inst of (`var`) `bool` and an expression that forms the assignment of a variable of type-inst T should be of that type-inst. Type reduction can change the expected type-inst of an expression to a reduced type-inst, and so we add an explicit conversion from the original type-inst to the reduced one. We have seen examples of this in Section 4.1 (assignment to variables) and Section 4.2 (body of a function definition). After type-reduction, we have no more calls to functions that require solver support for the type-insts we are reducing away (e.g. the functions `old2new_var`/1 and `new2old_var`/1 used for converting between set variables and Boolean arrays in Example 7).

Example 12
Consider again the function `set_op`/2 and its (partial) reduction into function `set_op'`/2 from Example 7. After type-reducing expressions, we get

```
function array[$T] of var bool: set_op'(array[$T] of var bool: s1,
            array[$T] of var bool: s2) :: type_reduction_of(set_op) =
   let {array[$T] of var bool: result =
         if reduced_card(ub(s1)) > reduced_card(ub(s2))
         then reduced_diff(s1, s2) else reduced_diff(s2, s1) endif}
   in old2new_var(new2old_var(result) :: Ann);
```

which further simplifies to

```
function array[$T] of var bool: set_op′(array[$T] of var bool: s1,
          array[$T] of var bool: s2) :: type_reduction_of(set_op) =
  if reduced_card(ub(s1)) > reduced_card(ub(s2))
  then reduced_diff(s1, s2) else reduced_diff(s2, s1) endif;
```

with `reduced_card`/1 and `reduced_diff`/2 reduced versions of respectively `card`/1 and `diff`/2. The upper bound (`ub`) of a reduced set is equal to the reduction of the upper bound of the original set. □

5 Practical Evaluation

Type reduction is now an integral part of the Zinc compiler, and it is run on every input model. On a suite of 49 Zinc examples, the system identified 15 for which type reduction was required (given a solver that supports Boolean, integer, float and integer set variables). Eight of them contained enum variables, six contained set variables with non-integer element types (enum, record and tuple), and two contained tuple variables. Type reduction applied to the Zinc regression suite of 380 tests, detects 48 that require type reduction. We expect as modellers become more aware of the high level modelling facilities of Zinc, the proportion of models requiring type reduction will grow.

While in most cases type reduction is necessary to make a model acceptable to solvers in the first place, we can also apply type reduction to types that *are* supported by the target solver, such as integer set variables in the G12 FD solver. In this solver, a set variable is represented by its lower bound (i.e. all elements that must be in it), upper bound (all elements that can be in it) and cardinality. Our reduced representation of sets as Boolean arrays does not represent the cardinality explicitly, which may cause it to perform worse on models involving cardinality constraints. We compared the run times of models with integer set variables with and without type reduction of these set variables, using the G12 FD solver. For a meaningful comparison, we ensured that the search strategy remains unchanged. The default search strategy in the FD solver prefers searching Booleans over integers and integers over sets. We postpone the search of Booleans representing set variables in the original model, and ensure that the variable and value selection of unreduced and reduced set variables is the same. Table 1 shows the run times of the original and reduced models (split into posting and search). The results show that the type reduction of set variables to Boolean arrays has little effect for the FD solver. Results for different solvers may vary of course.

6 Related Work

The $\mathcal{F}$ language (Hnich 2003) is a constraint modelling language supporting function variables, i.e. variables that take a function as their value. An $\mathcal{F}$ model is translated to a set of alternative models in the lower level language $\mathcal{L}$ (that does not support function variables) using the Fiona modelling tool. The generation of

Model	Instance	Original			Type-Reduced		
		Cd	Post	Search	Cd	Post	Search
steiner triples	n = 7	5.38s	0.01s	1.23s	7.37s	0.01s	1.33s
	n = 8		0.01s	141.89s		0.01s	145.78s
social golfers	players: 15, size: 3, weeks: 5	6.32s	0.12s	8.51s	9.69s	1.76s	5.43s
	players: 24, size: 4, weeks: 5		0.24s	58.53s		7.06s	58.34s
trucking	periods: 8, trucks: 4	4.19s	0.00s	3.93s	5.75s	0.01s	2.51s
	periods: 7, trucks: 5		0.00s	196.35s		0.01s	139.85s
round robin	10 teams	5.52s	0.29s	14.38s	10.08s	3.82s	19.53s

Table 1. *Run times of models with/without type reduction of set variables: Cd times the Zinc-to-Zinc Cadmium transformation (data-independent); Post is the time to post the constraints and do the initial propagation; Search is the time spent on searching.*

$\mathcal{L}$ models is done using $\mathcal{F}$-to-$\mathcal{L}$ rewrite rules. One important limitation is that arbitrarily nested expressions are not supported in $\mathcal{F}$, whereas they are in Zinc. Zinc does not support function variables, and so the actual rewrite rules for $\mathcal{F}$ are not relevant to it. $\mathcal{F}$ was followed up by ESRA (Flener et al. 2004) which extends it with more variable types.

The s-COMMA platform (Soto 2009) consists of an object-oriented solver-independent constraint modelling language whose models can be translated to different solvers. During the transformation process, the model is combined with instance data and flattened. Enum variables are reduced to integers as in our work.

Essence (Frisch et al. 2008) is a specification language for CSPs and shares many features with Zinc. It offers the following structured types: sets, multisets, matrices (arrays), relations, functions, and partitions. Similarly to Zinc, Essence models are transformed into a lower-level language (Essence′). This transformation is done by means of rules written in Conjure (Frisch et al. 2005), part of which are for eliminating variables of complex types that are not supported by the solvers. We note that Essence has no publicly available implementation, whereas Zinc is available through the G12 academic release (G12 Team 2010). Some important technical differences with our work follow.

In Essence, the connection between original problem variables and reduced ones is not maintained. Features needed for this are an output statement and comprehensions (or some other way to traverse and construct sets, arrays and such). Essence only supports a limited form of structure iteration in the form of the $\forall$, $\exists$ and $\sum$ constructs. This means that e.g. we can only convert a set to a Boolean array by means of equality constraints in a $\forall$ quantification.

In Conjure, the result of refining an expression, is a new expression, potentially tagged with a set of constraints. These constraints need to be lifted to the top, and so each rule that refines subexpressions during its execution, needs to state explicitly what to do with the constraints that might result from these refinements. Furthermore, the control flow in Conjure is top-down, whereas refinement is to take place from the bottom up (i.e. we first need to know how an expression's components are refined, before we can refine the expression itself), so each rule in CONJURE refining an expression E, calls explicitly for the refinement of all subexpressions of E.

Our approach does not exhibit the same drawbacks because of two reasons. Firstly, the Zinc language offers two features that enable us to encapsulate constraints, namely local variables and constrained type-insts. By encapsulating constraints, we can deal with them separately from the problem of type reduction. In particular, we have transformations for local variable lifting and for constrained type elimination. Secondly, Cadmium, being a term-rewriting system, works from the bottom up, and so we do not need to attempt refining expressions that do not contain subexpressions of an unsupported type-inst. Furthermore, we do not need to explicitly program the desired control flow: it is already implicitly there.

Finally, our rules do not require flattening of the model. Flattening is needed in ESSENCE because it does not have the features to model value conversion as an expression. In Zinc, value conversions can be implemented using comprehensions, local variables and constrained type-insts.

7 Conclusions

High-level modelling allows constraint programmers to build models closer to the problem specification by using complex types for variables and parameters. Unfortunately, current solvers do not implement complex variable types. Therefore, we need type reduction as an important part of transforming Zinc models to what solvers understand. Without type reduction, we would not have support for tuple and record variables, enum variables, or variable sets of arbitrary element type. Type reduction also forms the starting point for transformations such as linearisation (Brand et al. 2008), which creates models that can be solved by MIP solvers, or Booleanisation, which creates models for SAT solvers.

Type reduction has proven to be more difficult than we originally anticipated. One reason is the expressiveness of the Zinc language, which has many features that make modelling easier, but increase the burden on model transformation tools. Another is the data independence, which means that model transformations cannot rely on flattening, function inlining, etc. During the development of our transformations, we encountered some difficulties that stem from limitations of Zinc as a target language. One of them is that Zinc does not allow arbitrary set expressions to be used as a type, even though it is only a syntactic restriction as sets can always be given a name. Another difficulty is in the difference between implicitly and explicitly indexed arrays. In particular, if an array is declared with a finite index type, then its index set must be exactly that type and cannot be a subset of it.

Our type reduction transformations are part of the G12 academic release (G12 Team 2010). In future work, we plan to add support for annotations on how to reduce variables and constraints involving them. In related work, such as on ESRA (Flener et al. 2004) and ESSENCE (Frisch et al. 2008), it has been proposed to generate all possible models and then select a good one using some criteria. In the context of Zinc, we plan first to allow the modeller to steer the reduction process using annotations.

References

Brand, S., Duck, G. J., Puchinger, J., and Stuckey, P. J. 2008. Flexible, rule-based constraint model linearisation. In *PADL*. LNCS 4902. 68–83.

Cheng, B., Choi, K., Lee, J., and Wu, J. 1999. Increasing constraint propagation by redundant modeling: an experience report. *Constraints 4,* 2, 167–192.

Duck, G. J., De Koninck, L., and Stuckey, P. J. 2008. Cadmium: An implementation of ACD Term Rewriting. In *ICLP*. LNCS 5366. 531–545.

Flener, P., Pearson, J., and Ågren, M. 2004. Introducing esra, a relational language for modelling combinatorial problems. In *LOPSTR*. LNCS 3018. 214–232.

Frisch, A. M., Harvey, W., Jefferson, C., Martínez Hernández, B., and Miguel, I. 2008. Essence: A constraint language for specifying combinatorial problems. *Constraints 13,* 3, 268–306.

Frisch, A. M., Jefferson, C., Martínez Hernández, B., and Miguel, I. 2005. The rules of constraint modelling. In *IJCAI*. Professional Book Center, 109–116.

G12 Team. 2010. The G12 constraint programming platform website. `http://g12.research.nicta.com.au/`.

Hnich, B. 2003. Function variables for constraint programming. Ph.D. thesis, Uppsala University.

Marriott, K. et al. 2008. The design of the Zinc modelling language. *Constraints 13,* 3, 229–267.

Soto, R. 2009. Languages and model tranformation in constraint programming. Ph.D. thesis, CNRS, LINA, Université de Nantes, France.

Walsh, T. 2003. Consistency and propagation with multiset constraints: a formal viewpoint. In *CP*. 724–738.