Constraints in Non-Boolean Contexts

Leslie De Koninck, Sebastian Brand, and Peter J. Stuckey

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

Abstract

In high-level constraint modelling languages, constraints can occur in non-Boolean contexts: implicitly, in the form of partial functions, or more explicitly, in the form of constraints on local variables in non-Boolean expressions. Specifications using these facilities are often more succinct. However, these specifications are typically executed on solvers that only support questions of the form of existentially quantified conjunctions of constraints.

We show how we can translate expressions with constraints appearing in non-Boolean contexts into conjunctions of ordinary constraints. The translation is clearly structured into constrained type elimination, local variable lifting and partial function elimination. We explain our approach in the context of the modelling language Zinc. An implementation of it is an integral part of our Zinc compiler.

1998 ACM Subject Classification F.4.1 Mathematical Logic: Logic and constraint programming; F.3.3 Semantics of Programming Languages; I.2.2 Automatic Programming: Program transformation

Keywords and phrases Constraint modelling languages, model transformation, partial functions

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.117

1 Introduction

In high-level constraint modelling languages such as ESSENCE [3], OPL [13] and Zinc [8] constraints can occur in non-Boolean expressions: i.e. expressions whose value is not a Boolean, but for instance an integer. Most commonly, such constraints appear in the form of partial functions. For example "$y + 1$ `div` $y = 2 \quad \vee \quad y \leqslant 0$" is a constraint involving an application of the partial function `div` (integer division), which is undefined if the divisor equals zero. The subexpression 1 `div` y is an integer expression that implicitly introduces the constraint that y must be non-zero.

Several proposals for dealing with such constraints resulting from partiality are studied in [5]. In an imperative language, we may simply abort if y takes the value 0, but in a declarative language, this is unacceptable. The most popular approach for (constraint) modelling languages is the so-called *relational* semantics, which treats functions as "shorthand" for relations, since this is easy to support by solvers. In this semantics, the implicit constraint $y \neq 0$ is active at the nearest enclosing Boolean context. The above example thus means "$(y + 1$ `div` $y = 2 \;\wedge\; y \neq 0) \quad \vee \quad y \leqslant 0$".

In Zinc, constraints in non-Boolean contexts can also arise in the form of a local variable declaration in a non-Boolean context if the type-inst[1] is constrained. Local variables are created using expressions of the form "`let` $\{T\colon\ x = A\}$ `in` E" where E is a (Boolean or

[1] A *type-inst* in Zinc is the combination of a type, such as `int`, and an instantiation pattern. See Section 2.

non-Boolean) expression in which we introduce a new local variable called x whose type-inst is T and whose assignment (optional) is A. A formal semantics of these `let` expressions is given in Section 2.

Now consider the following constraint, imposing an ordering between two tuples

$(\mathtt{u}, \mathtt{v}) < (\mathtt{y}, \mathtt{let}\ \{\mathtt{var}\ 0..4\colon\ \mathtt{x} = 2 \cdot \mathtt{y} + 1\}\ \mathtt{in}\ \mathtt{x} \cdot \mathtt{x})$

where a local variable `x` is introduced to factor out a computation. It also implicitly introduces the constraint $0 \leqslant \mathtt{x} \leqslant 4$ in the context of the non-Boolean expression $\mathtt{x} \cdot \mathtt{x}$. In the relational semantics this constraint is active at the level of the nearest enclosing Boolean expression, that is, at the level of the tuple comparison.

In the above example, we introduced a range type-inst (`var` 0..4), which is one of the forms of constrained type-ints in Zinc. The most general form is the arbitrarily constrained type-inst: (T: x `where` $C(x)$) with T a type-inst, x a name to refer to any value of T, and $C(x)$ a constraint that all values of the resulting constrained type-inst have to satisfy.

Constrained type-insts are useful to improve locality during modelling, but in particular they also frequently arise in the process of (automatic) type reduction: the mapping of complex structured types to basic types that can be handled by solvers. Consider the expression $\mathtt{min}((x, 1), (y, 2))$ where x and y are integer variables, which represents the minimum of the tuples $(x, 1)$ and $(y, 2)$ (in lexicographic order). It may be mapped to

$\mathtt{let}\ \{(\mathtt{tuple}(\mathtt{var\ int}, \mathtt{var\ int})\colon \mathtt{t}$
$\quad\quad \mathtt{where}\ (x \leqslant y \rightarrow \mathtt{t} = (x, 1)) \wedge (x > y \rightarrow \mathtt{t} = (y, 2)))\colon \mathtt{m}\}\ \mathtt{in}\ \mathtt{m}$

so that it can be dealt with by solvers that do not support `min`/2 over tuples.

These examples illustrate how constraint models can involve (in some cases quite complex) constraints attached to non-Boolean expressions. To match the relational semantics of constraint modelling languages, and to eventually reach a constraint model that can be directly supported by underlying solvers, these constraints must be lifted to the nearest enclosing Boolean context. This is a non-trivial task since the enclosing non-Boolean operations can be deeply nested.

While Zinc has a direct way of expressing constraints in non-Boolean contexts using local variables with constrained type-insts, other constraint modelling languages such as $\mathcal{F}$ [7] and ESSENCE [3] have to deal with such constraints as well during model transformations. In these languages, the model transformation rules need to keep track of constraints attached to non-Boolean expressions, and each rule needs to state how to deal with them appropriately.

In this paper, we propose a series of transformations to obtain the relational semantics for non-Boolean expressions that have constraints attached to them. Partial functions are a special case of this, and in that respect, this work generalises [5] by dealing with any sort of constraint in a non-Boolean context, and by decomposing the problem into partial function elimination, local variable lifting and constrained type elimination. While we focus on the Zinc language, this work is relevant to all constraint modelling languages and logic programming-based languages that allow partial function applications, and in particular also to functional-logic programming languages such as Curry [6]. Our transformations have the following significant features:

Zinc-to-Zinc: the result of the transformations is Zinc – no separate information outside a model needs to be maintained. This means these transformations can 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.

Locality: the transformations are only concerned with those parts of the model that contain non-Boolean constraints. Flattening of the model is not required.

2 Preliminaries

Zinc. A Zinc model consists of a set of items for decision variable and parameter declarations, 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 variable, a tuple whose first and second field are a Boolean parameter and a float variable respectively, and an 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 [8] 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 s, 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. ◀

In the following, we highlight some features of Zinc that are important for our discussion.

Zinc supports type-insts beyond the base ones (`int`, `float`, etc.), called *constrained* type-insts. One particularly expressive case of these are *arbitrarily constrained* type-insts. An example is (`int`: `i where i` > 0), denoting the positive integers.

Variables with a local scope can be introduced by `let` expressions. The scope, and thus the type-inst of the `let` expression itself, can be Boolean or non-Boolean. One can have multiple comma-separated variable declarations inside a single `let` expression.

Zinc allows the user to define their own functions and predicates (functions returning `var bool`). A function definition has the form "`function` T: $f(T_1\colon x_1, \ldots, T_n\colon x_n) = E$", where T is the return type-inst, f is the function name, T_i and x_i are respectively the type-insts and names of its arguments, and E is its body, which is an expression of type-inst T.

Semantics of Zinc. In this paper we are mainly concerned with constraints on non-Boolean expressions. In the introduction, we already gave an example of the intended meaning of an expression involving division. In the relational semantics, we consider Boolean expressions only, and in particular we decompose nested non-Boolean expressions into a conjunction of equality constraints. We denote the meaning of a Boolean expression E by $\mathcal{I}(E)$. Here is a (partial) definition of $\mathcal{I}$, given in priority order:
$\mathcal{I}(x = y) \;:=\; x = y$ where x and y are variables or constants;

[2] A parameter must be assigned, but the assignment can be in a separate data file.

$\mathcal{I}(x = B) \;:=\; x \leftrightarrow \mathcal{I}(B)$ with x a variable or constant and B a Boolean expression;

$\mathcal{I}(x = f(y_1,\ldots,y_n)) \;:=\; \exists y'_1,\ldots,y'_n : \bigwedge_{i\in 1..n} \mathcal{I}(y'_i = y_i) \wedge x = f(y'_1,\ldots,y'_n)$ for any *total* non-Boolean function f/n where x is a variable or constant;

$\mathcal{I}(x = f(y_1,\ldots,y_n)) \;:=\; \exists y'_1,\ldots,y'_n : \bigwedge_{i\in 1..n} \mathcal{I}(y'_i = y_i) \wedge p(y'_1,\ldots,y'_n,x)$ for any *partial* non-Boolean function f/n where x is a variable or constant and $p(z_1,\ldots,z_n,z_{n+1}) \leftrightarrow (f(z_1,\ldots,z_n) = z_{n+1})$;

$\mathcal{I}(x = \texttt{let}\ \{T : y = A\}\ \texttt{in}\ E) \;:=\; \exists y : \mathcal{I}_{ti}(T,y) \wedge \mathcal{I}(y = A) \wedge \mathcal{I}(x = E)$ with x a variable or constant and $\mathcal{I}_{ti}(T,y)$ the interpretation of y being of type-inst T;

$\mathcal{I}(p(x_1,\ldots,x_n)) \;:=\; \exists x'_1,\ldots,x'_n : \bigwedge_{i\in 1..n} \mathcal{I}(x'_i = x_i) \wedge p(x'_1,\ldots,x'_n)$ for any predicate p/n, including operators such as $\wedge/2$, $\neg/1$, $=/2$ and $\leqslant/2$.

The interpretation for partial functions can also be used for total functions; however, the latter is more compact.

▶ **Example 2.** For the example from the introduction we find

$$\mathcal{I}(y + 1\ \texttt{div}\ y = 2 \;\vee\; y \leqslant 0) = \\ (\exists i_1 : (\exists i_2 : \texttt{div}(1,y,i_2) \wedge i_1 = y + i_2) \wedge i_1 = 2) \;\vee\; y \leqslant 0$$

(after some simplification). For $y = 0$ it evaluates to `true`. ◀

From Zinc to solver. Our compiler for Zinc proceeds in two main stages. First, the high-level Zinc model is reduced to an equivalent model in a subset of Zinc called CoreZinc. For fixed expressions (expressions over parameters), CoreZinc is equivalent to Zinc. For expressions involving decision variables, CoreZinc is similar to MiniZinc [9]. The transformation from Zinc to CoreZinc is done independently of instance data by rewrite rules in the model transformation language Cadmium [1].

The resulting CoreZinc model is compiled into procedural code for the entire solution process, that is, to create variables and post constraints in the solvers, maintain communication between solvers, handle search, and generate output.

To handle constraints in non-Boolean context there are three interacting translations: elimination of constrained type-insts, lifting local variables, and elimination of unsafe partial function applications. They are described in the following three sections.

3 Elimination of Constrained Type-Insts

In Zinc, type-insts can be basic (e.g. `bool`, `int`, `float`) or constrained. Constrained type-insts have one of the following forms:

- ranges of the form "$l..u$" with l and u either (fixed) integer or float expressions;
- enumerated sets, e.g. $\{1,3,4\}$, or set parameters;
- arbitrarily constrained type-insts of the form (T: x `where` $C(x)$) with T a type-inst, x a name, and $C(x)$ a constraint;
- structured type-insts with constrained type-insts as components.

Solvers typically support range domains for their variables. The more complex type constraints, however, need to be converted into regular constraints.

A constraint in the type-inst of a global variable can immediately be extracted and posted as a regular constraint at the model root level. The interesting case is that of a local variable with constrained type-inst. In that case, the type constraints need to be lifted to the nearest enclosing Boolean context. This is done by first lifting the variable declaration to this nearest Boolean context and then extracting the parts of the type constraint that can cause failure. The lifting to a Boolean context is described in Section 4. We show here how to extract the type constraint and add it to the Boolean context of the declaration.

The type-inst in the declaration of a local variable can cause failure in three ways:

(a) the type domain of the variable is empty;

(b) its assignment is not within its type-inst;

(c) its assignment fails.

We deal with case (a) by redeclaring the local variable with a guaranteed non-empty type-inst and adding a constraint stating that the original type-inst must be non-empty. For range type-insts, we do so by transforming "`let` $\{$`var` $l..u$: $x\}$ `in` B" into "`let` $\{$`var` $l..$`max`(l,u): $x\}$ `in` $B \ \wedge \ l \leqslant u$". Here and in what follows, B is Boolean because we assume the `let` expression has already been lifted to the nearest Boolean context.

Local declarations with non-range set type-insts are dealt with as follows:

`let` $\{$`var` s: $x\}$ `in` B

which states that x can take any value in s, is transformed into

`let` $\{$`var` (`if card`$(s) > 0$ `then` s `else` $\{d\}$ `endif`): $x\}$ `in` $B \wedge$ `card`$(s) > 0$

where d is an arbitrary value of the appropriate type, e.g. 0 for the integers.

For tuples and record types, we ensure each field has a non-empty type-inst. Set types are always non-empty, as the empty set is always a possible value. Array types are non-empty if their element type is non-empty. The index type can be empty, as in such case the empty array is a possible value.

For case (b), an assignment that might not be within the type-inst, we relax the type-inst to its base type-inst, which always includes the assigned value. The original type-inst is made explicit as an arbitrarily constrained type-inst and then extracted.

For range type-insts, "`let` $\{$`var` $l..u$: $x = A\}$ `in` B" is transformed into "`let` $\{$`var int`: $x = A\}$ `in` $B \ \wedge \ l \leqslant A \ \wedge \ A \leqslant u$".

For set type-insts in general, "`let` $\{$`var` s: $x = A\}$ `in` B" is transformed into "`let` $\{$`var` T: $x = A\}$ `in` $B \ \wedge \ A$ `in` s" where T is the (base) element type of s.

The final case (c), a (non-Boolean) assignment that fails, can only occur because of partial functions. We treat it in Section 5.

4 Lifting Local Variables

Constraints that appear in non-Boolean contexts, either implicitly for partial functions or explicitly, must be lifted to the nearest enclosing Boolean context. Similarly, in order to allow solving by a constraint solver that can only handle existentially quantified conjunctions of constraints, all local variables must be lifted to the top level.

Local variable lifting is the process of lifting local variable declarations through enclosing expressions, modifying the declarations and occurrences of the declared variables as needed. It is the key step in dealing with these kinds of requirements.

In this section we show how we can lift local variable definitions through the different Zinc constructs.

4.1 Base Case

The base case for lifting is simple. For non-Boolean expression E_i, introducing local variable x and appearing as an argument to function f: $f(E_1, \ldots,$ `let` $\{T\colon x\}$ `in` $E_i, \ldots, E_n)$ transforms into `let` $\{T\colon x\}$ `in` $f(E_1, \ldots, E_i, \ldots, E_n)$. We assume x does not occur in $E_1, \ldots, E_{i-1}, E_{i+1}, \ldots, E_n$, renaming it to a unique new name if necessary.

4.2 Boolean Contexts

For a local variable declaration in a Boolean context, we apply the transformations of Section 3 to extract potentially failing type constraints. We can then safely lift the local variable above the Boolean context in which it appears using the base case.

One complication is that Zinc does not allow unassigned local variables in negative Boolean contexts, such as the argument of `not` or the first argument of implication `->`. This is because existentially quantified variables become universally quantified by lifting through negation, and our target solvers do not implement universal quantification.

▶ **Example 3.** The expression "`not` (`let` {`var float`: x} `in` B)" means that there is no float value x for which B holds. In other words, for all float values x, expression B is false. ◀

Original Zinc models with unassigned local variables in negative contexts will be rejected, hence all originally occurring `let` constructs can be lifted out. The failure extraction transformations of Section 3 never remove an assignment.

However, sometimes an unassigned local variable may in fact be constrained to a single value by a type constraint. For example, "`let` $\{(T\colon x$ `where` $x = A)\colon y\}$ `in` $E(y)$" is of course equivalent to "`let` $\{T\colon x = A\}$ `in` $E(x)$". Another example, that cannot be written equivalently as an assignment, was given in the introduction where we translated the minimum of two tuples into an appropriately constrained tuple variable. We allow such implicitly assigned variables to be lifted over negative contexts.

4.3 Structured Types

Constraints can occur within a component of a structured type, e.g. within a field of a tuple or within an element of an array. Following the relational semantics, this constraint holds for the whole structure, and so we have to be careful with evaluating structure access as shown in the following example.

▶ **Example 4.** Consider the constraint "(5, `let` {`var` 1..10: `i` = 0} `in` `i`).1 = 5". Evaluating the tuple access would result in "5 = 5", which is trivially true. Alternatively, we could lift the declaration, giving us "`let` {`var` 1..10: `i` = 0} `in` ((5, `i`).1 = 5)" which further evaluates to `false`. The correct answer is found by looking at the semantics (Section 2) and treating tuple construction and access as non-Boolean functions. We obtain $t.1 = 5 \wedge t = (5, t_2) \wedge (\exists i : i = 0 \wedge i \in 1..10 \wedge t_2 = i)$. The failure holds for the entire structure. ◀

One important consequence of this is that some seemingly reasonable simplifications are incorrect. For example, the tuple access $(a_1, \ldots, a_n).i$ cannot always be replaced by a_i.

4.4 Comprehensions

An array comprehension $[H(i) \mid i$ `in` G `where` $W(i)]$ generates an array of instances of H for each value in the array G satisfying condition W. Local variables can appear in comprehensions in three places: in the comprehension head H, in a generator expression G, and in the condition W. We now look at each of these in detail.

Comprehension head. The way a local variable declaration is lifted outside of a comprehension head depends on which of the following properties the declaration has:
(1) the type-inst is independent of the generator;
(2) the assignment (if present) is independent of the generator;
(3) the assignment (if present) is free of local unassigned variables.
Before describing the general approach, we give two special cases for which we can do better.

Special case 1: an assignment is present and all properties (1), (2) and (3) hold. We can lift the variable declaration outside as is, as the different instantiations of the declaration, one for each value of the generator, have the same value and type-inst. More formally, [`let` $\{T$: `x` $= A\}$ `in` $E(x) \mid g$ `in` G] is rewritten to `let` $\{T$: `x` $= A\}$ `in` [$E(x) \mid g$ `in` G].

▶ **Example 5.** To see that property (3) is indeed needed, consider

[`let {var int: x = y + let {var 0..8: z} in z} in a[x] > 0 | i in 1..9`]

which evaluates to an array of 9 Boolean variables. This would become

`let {var int: x = y + let {var 0..8: z} in z} in [ a[x] > 0 | i in 1..9 ]`.

However, this is not equivalent to the original version: it would force all 9 instantiations of `z` (and `x` by transition) to take the same value. ◀

Special case 2: property (1) holds, and if an assignment is present, it does not have property (2) or (3). In this case the instantiations of the local variable within the comprehension are different in general. We need one copy of it for each generator value. Therefore, we create an array of variables outside the comprehension and replace each occurrence of the original local variable by a lookup into this array. More concretely,
[`let` $\{T$: $x = A(g)\}$ `in` $E(x) \mid g$ `in` G] is rewritten to
`let` $\{\text{array}[G]$ of T: $y =$ [$A(g) \mid g$ `in` G]$\}$ `in` [$E(y[g]) \mid g$ `in` G]. The case without an assignment to x is similar: we simply omit the assignment to y.

Here, we use the generator as an index into the array. This is not always possible, i.e. when the generator values are either not fixed, or potentially contain duplicates. If necessary, we first transform the comprehension so that its generator ranges are sets. For example, [$E(x) \mid g$ `in` G] is rewritten to [$E(G[g']) \mid g'$ `in` `index_set`(G)].

General case: property (1) does not necessarily hold. Again, we create an array for separate declarations outside of the comprehension, but this time we need to generalise the type-inst of its element type to make it independent of the generator. This is done in two steps. We will illustrate them using the comprehension

[`let` $\{T(g)$: $x = A(g)\}$ `in` $E(x) \mid g$ `in` G].

First, we make those type constraints that depend on the generator explicit by using an arbitrarily constrained type-inst:

[`let` $\{(T'$: y `where` $C_T(y, g))$: $x = A(g)\}$ `in` $E(x) \mid g$ `in` G]

where T' is a supertype of T that does not depend on g. In the second step, we create a constrained type-inst for the resulting array in which each element has the appropriate generator-dependent type constraint:

`let` $\{(\text{array}[G]$ of T': w `where forall`$(g$ `in` $G)(\ C_T(w[g], g)\ ))$: $z =$
  [$A(g) \mid g$ `in` G]$\}$ `in` [$E(z[g]) \mid g$ `in` G]

▶ **Example 6.**

[`let {tuple(var 1..j, var i..j): x = f(i)} in g(x) | i in 1..5`] is a more concrete example. Only the second component of the type-inst of `x` depends on a generator. It is first transformed to

```
[ let ({tuple(var 1..j, var int): y where i ⩽ y.2 ∧ y.2 ⩽ j): x = f(i)}
  in g(x) | i in 1..5 ].
```

Then the constrained type-inst is lifted out of the comprehension to give

```
let {(array[1..5] of tuple(var 1..j, var int): y where
      forall(i in 1..5)(i ⩽ y[i].2 ∧ y[i].2 ⩽ j)): x = [ f(i) | i in 1..5 ]}
in [ g(x[i]) | i in 1..5 ].
```

◀

Generator expression. A declaration in the generator expression can directly be lifted out of the comprehension since it is independent of the generator.

`where` condition. The `where` condition forms a Boolean context, and so the rules of Section 3 apply. A non-failing declaration in a `where` condition can be lifted outside the comprehension similar to declarations in the comprehension head.

Multiple generators. Zinc allows multiple generators for the same comprehension. The extension to handle these is straightforward: essentially the declaration is lifted to a multi-dimensional array declaration. The most complicated case is declarations in generator expressions. In general

$[\ E \mid g_1 \texttt{ in } G_1,\ \ldots,\ g_i \texttt{ in let } \{T\colon\ x = A\} \texttt{ in } G_i(x),\ \ldots,\ g_n \texttt{ in } G_n \texttt{ where } C\ ]$

can be rewritten into

$\texttt{let } \{\texttt{array}[T_1, \ldots, T_{i-1}] \texttt{ of } T\colon\ x =$
$\qquad\qquad [\ (g_1, \ldots, g_{i-1})\colon\ A \mid g_1 \texttt{ in } G_1, \ldots, g_{i-1} \texttt{ in } G_{i-1}\ ]\}$
$\texttt{in } [\ E \mid g_1 \texttt{ in } G_1, \ldots, g_i \texttt{ in } G_i(x[g_1, \ldots, g_{i-1}]), \ldots, g_n \texttt{ in } G_n \texttt{ where } C\ ]$

where $T_1 \ldots T_{i-1}$ are the element type-insts of the arrays $G_1, \ldots, G_{i-1}$.

5 Elimination of Unsafe Partial Function Applications

Zinc includes various built-in partial functions, such as division and modulo (not defined if the divisor equals zero), array lookup (not defined for index values outside of the index set of the array), or the minimum of a set (not defined if the set is empty). Furthermore, there are a number of partial real functions, such as trigonometric ones.

The treatment of a partial function application depends on whether it operates on a fixed value. If so, the application is left as is, and it is simply evaluated when required. Not translating fixed applications avoids the associated increase in expression size.

For non-fixed values, the partial function acts as a constraint that restricts the values to be within the domain of the function. These applications are transformed to make the constraints explicit. In the process, the partial function application is made *safe*: its argument is guaranteed to be within the function's domain.

Solvers generally do not support reified versions of constraints such as the `element` constraint, which models an array lookup with a variable as the index, or integer division. Moreover, such reified constraints are absent in low-level solver input languages, such as FlatZinc [9] or XCSP [10]. If we simply lift such constraints to the top level conjunction, we end up with the strict semantics rather than the relational one [5]. We must therefore eliminate partiality from models.

The basic idea behind the transformation is that we encode the potential failure of a partial function application by a local variable declaration with a constrained type-inst.

How to move it to the nearest Boolean context is discussed earlier in the paper. Furthermore, we ensure that the partial function application in constraint form never fails because it is applied using input values outside of its domain.

Integer division. As an example of a partial arithmetic function, we show how integer division is made safe. A propagator for division as a constraint excludes zero from the domain of the divisor. Let x `div` y be a *potentially unsafe division.* That is, it is not a priori clear that y can never assume the value 0. A first attempt to transform this division is

`let` $\{$(`var int`: w' `where` $w' = y$): $w\}$ `in` x `div` w.

Constrained-type elimination will extract the type constraint $w' = y$ and add it to the nearest enclosing Boolean context. An issue here is that w is an unassigned local variable. If the original expression x `div` y appears in a negative Boolean context, then we now have an unassigned local variable in a negative context, a situation we disallow. Therefore, we find the above formulation undesirable. We prefer

`let` $\{$(`var int`: w' `where` $w' = y$): $w = y +$ `bool2int`$(y = 0)\}$ `in` x `div` w.

Again, constrained-type elimination will extract the type constraint and add it to the appropriate context. Moreover, w has an assignment which is guaranteed to be inside the domain of the partial function.

Looking at the interpretation of both expressions (see Section 2), we have that $\mathcal{I}(z = x \texttt{ div } y)$, which reduces to $\texttt{div}(x, y, z)$, is equivalent to

$$\mathcal{I}(z = \texttt{let } \{(\texttt{var int} : w' \texttt{ where } w' = y) : w = y + \texttt{bool2int}(y = 0)\} \texttt{ in } x \texttt{ div } w).$$

Assuming a *total* division operation, this reduces to

$$\exists w : w = y \land w = y + \texttt{bool2int}(y = 0) \land z = x \texttt{ div } w$$

which for $y = 0$ simplifies to `false` and for $y \neq 0$ to $z = x$ `div` y. Moreover, we can safely remove zero from the domain of w without affecting the possibility of y being zero.

Alternative encodings are possible, for example

`let` $\{$(`var int`: w' `where` $w' = y$): $w = [y,\ 1][$`bool2int`$(y = 0) + 1]\}$ `in` x `div` w.

Which encoding is more suitable depends on which solver is being used.

General approach. In general, let $f/1$ be a unary partial function, $c/1$ a constraint that succeeds when its input is in the domain of $f/1$ and fails otherwise, T be the type-inst of the domain of $f/1$ and d a value within the domain of $f/1$. We can transform a call $f(x)$ into a safe partial function application as follows:

`let` $\{$(`var` T: w' `where` $w' = x$): $w = [d, x][$`bool2int`$(c(x)) + 1]\}$ `in` $f(w)$.

We can consider n-ary functions as unary functions operating on n-ary tuples.

6 Conclusions

Local variables and constrained types are crucial for both high-level modelling and effective model transformation (e.g. type reduction or solver-specific constraint transformations). To solve Zinc models, we must have a way to transform away these constructs to obtain existentially quantified conjunctions of constraints. The transformations that do this in a data-independent way are challenging and form a core part of Zinc. Any declarative language that wishes to treat partial functions correctly and in a data-independent manner must address these issues.

We have presented three sets of transformation rules to deal with respectively constrained type-insts, local variables and partial functions in Zinc. They transform valid Zinc models to semantically equivalent ones that are free of arbitrarily constrained type-insts, local variable declarations and potentially failing partial function applications. Our transformations are data-independent and can be run concurrently. Previous work on Zinc [8] required flattening the model and was not data independent.

In practice, we run partial function elimination first. Local variable lifting and constrained type elimination take place concurrently. They are run multiple times during the model transformation process, because other transformations introduce local variables and constrained type-insts but may assume a model without them.

Related Work

Zinc belongs to the family of constraint modelling languages that also includes $\mathcal{F}$ [7] and its successor ESRA [2], s-COMMA [12], ESSENCE [3] and OPL [13].

$\mathcal{F}$ provides function variables. Both partial and total functions can be represented. It supports a function membership operation $\langle i, j\rangle \in F$ which is equivalent to $i \in dom(F) \wedge F(i) = j$. However, function application is only allowed for values within the domain of the function. $\mathcal{F}$ models are translated into a lower-level language. For some expressions, this translation requires the introduction of new variables and constraints. Since local variables and a way to encapsulate constraints in non-Boolean contexts are not part of the language, these new variables and constraints need to be lifted. It is not described how this is done.

The s-COMMA language is an object-oriented constraint modelling language. It appears to support division and variable index array lookups, but it is unclear how it deals with undefinedness in these operations.

ESSENCE is a specification language for CSPs and shares many features with Zinc. ESSENCE models are transformed into the lower-level language ESSENCE′ using transformation rules written in CONJURE [4]. In CONJURE, the result of rewriting an expression is a new expression that may be tagged with a set of constraints. Since ESSENCE lacks local variables and an encapsulation mechanism for constraints in a non-Boolean context, rules need to state explicitly what to do with constraints that result from refining subexpressions.

Frisch and Stuckey [5] study undefinedness in the constraint (logic) programming languages ECL^{i}PS^e, SWI-Prolog, SICStus Prolog, OPL and MiniZinc. They discuss three different formal semantics and show how to transform models to ensure they behave as desired. The transformations consist of first identifying the nearest Boolean context of every unsafe function application and creating a local variable alias for it, which is immediately lifted to the top level. Next, each of these Boolean contexts is made safe by replacing unsafe function applications by safe versions and adding the necessary constraints. The approach of [5] is not always applicable to Zinc, as it can be the case that the unsafe partial function application uses variables that are not defined at the level of the nearest Boolean context.

In the functional-logic programming language Curry [6], a program is a set of functions. A function in Curry can be nondeterministic and in particular it can be partial. Whenever a partial function is applied to a value outside of its domain, the function application fails, which is different from returning false. As a result, we have that for instance `not x` evaluates to `True` if `x` evaluates to `False` and vice versa, but fails if `x` fails.

Mercury [11] allows functions to have a solution or to fail. Unlike Curry, it allows reasoning about success and failure using conjunction, disjunction, negation, etc. Failure is automatically lifted to the nearest Boolean context. However, Mercury is only concerned with evaluating fixed expressions.

References

1 Gregory J. Duck, Leslie De Koninck, and Peter J. Stuckey. Cadmium: An implementation of ACD Term Rewriting. In María García de la Banda and Enrico Pontelli, editors, *24th International Conference on Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 531–545. Springer, 2008.

2 Pierre Flener, Justin Pearson, and Magnus Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In Sandro Etalle, editor, *14th International Symposium on Logic Based Program Synthesis and Transformation*, volume 3018 of *Lecture Notes in Computer Science*, pages 214–232. Springer, 2004.

3 Alan M. Frisch, Warwick Harvey, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

4 Alan M. Frisch, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The rules of constraint modelling. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *19th International Joint Conference on Artificial Intelligence*, pages 109–116. Professional Book Center, 2005.

5 Alan M. Frisch and Peter J. Stuckey. The proper treatment of undefinedness in constraint languages. In Ian Gent, editor, *15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 367–382. Springer, 2009.

6 Michael Hanus. Curry: a multi-paradigm declarative language. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *12th Workshop on Logic Programming*, 1997.

7 Brahim Hnich. *Function variables for constraint programming.* PhD thesis, Uppsala University, 2003.

8 Kim Marriott et al. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.

9 Nicholas Nethercote et al. MiniZinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

10 Olivier Roussel and Christophe Lecoutre. XML representation of constraint networks: Format XCSP 2.1. *Computing Research Repository (CoRR)*, abs/0902.2362, 2009.

11 Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

12 Ricardo Soto. *Languages and Model Tranformation in Constraint Programming.* PhD thesis, CNRS, LINA, Université de Nantes, France, 2009.

13 Pascal Van Hentenryck, Irvin Lustig, Laurent Michel, and Jean-François Puget. *The OPL Optimization Programming Language.* MIT Press, 1999.