MiniZinc with Functions

Peter J. Stuckey[1] and Guido Tack[2]

[1] National ICT Australia (NICTA) and University of Melbourne, Victoria, Australia
pstuckey@unimelb.edu.au
[2] National ICT Australia (NICTA) and Faculty of IT, Monash University, Australia
guido.tack@monash.edu

Abstract. Functional relations are ubiquitous in combinatorial problems – the Global Constraint Catalog lists 120 functional constraints. This paper argues that the ability to express functional constraints with functional syntax leads to more elegant and readable models, and that it enables a better translation of the models to different underlying solving technologies such as CP, MIP, or SAT.
Yet, most modelling languages only support built-in functions, such as arithmetic, Boolean, or array access operations. Custom, user-defined functions are either not catered for at all, or they have an ad-hoc implementation without a useful semantics in Boolean contexts and not exploiting potential optimisations.
This paper develops a translation from MiniZinc with user-defined functions to FlatZinc. The translation respects the relational semantics of MiniZinc, correctly dealing with partial functions in arbitrary Boolean contexts. At the same time, it takes advantage of the full potential of common subexpression elimination.

1 Introduction

Functions are ubiquitous in models of combinatorial problems. They appear in arithmetic expressions, array accesses, expressions over sets of variables, or more complicated relations such as sorting or channeling. While it is always possible to express a functional dependency as a relational constraint, standard functional notation makes models more elegant, concise, and self-documenting. Furthermore, functional constraints are the main source of *common subexpressions*, which can be detected and eliminated from the model to improve solving performance.

Yet, most constraint modelling languages only provide a restricted set of built-in functions, e.g. for arithmetic expressions. It is either impossible for users to define their own functions, or one must resort to functions as present in the host language that the modelling language is embedded in. The latter, however, means that the semantics of functions is dictated by the host language, and in the case of partial functions, this almost certainly clashes with the logical semantics one would expect from the constraint model. Using host language functions also prevents the detection of common subexpressions and the corresponding optimisations.

Solver independent modelling languages need powerful abstraction facilities in order to encode how models are mapped to the form required by an underlying solver. The MiniZinc [11] compiler translates a model together with the instance parameters to

FlatZinc, a lower-level language understood by many solvers. To specialise the generated FlatZinc for a particular solver, the compiler uses a library of solver-specific constraint decompositions. The addition of functions, with common subexpression elimination, makes the encoding of solver-specific translations much more powerful.

This paper introduces an extension to the MiniZinc modelling language that adds support for user-defined functions, respecting the relational semantics of the language, and taking full advantage of potential optimisations. The contributions of this paper are:

– An extension to MiniZinc to support user-defined functions and constraints in `let` expressions.
– The first algorithmic description of the translation from MiniZinc (with functions) to FlatZinc. Keeping FlatZinc unchanged means that any solver that can interpret FlatZinc can take advantage of the improved modelling features.
– The introduction of totality annotations and a schema for defining partial functions that makes user-defined functions more flexible and efficient.

1.1 Partial Functions and the Relational Semantics

Partial functions are undefined on some inputs. The *relational semantics* [5] of MiniZinc regards a partial function as special notation for a relation, which when applied to a value outside its domain makes its surrounding *Boolean context* false.

Let us have a look at the partial function `div`, which expresses integer division and requires the divisor to be non-zero. Consider the following simple constraint:

```
constraint y != 0 -> (x div y) + z = 0;
```

The relational semantics demands that $x = 0, y = 0, z = 0$ is a solution of the problem, because the partiality of `div` is confined to its Boolean context, the right-hand side of the implication. This means that we cannot simply decompose constraints by introducing auxiliary variables for intermediate results:

```
var int: tmp;
constraint tmp = x div y;
constraint y != 0 -> tmp + z = 0;
```

This formulation lifts the partiality of `div` to the top level, so that $y = 0$ is ruled out by propagation of the `div` constraint, and $x = 0, y = 0, z = 0$ is no longer a solution of the problem. This simple decomposition gives the *strict semantics* [5], as implemented e.g. in SICStus Prolog and OPL. But this is not usually what modellers require – indeed the example was written specifically to guard against the case that $y = 0$.

Clearly, user-defined functions should respect the same relational semantics as the built-ins. Any function that constrains its arguments must be considered partial.

1.2 Common Subexpression Elimination

A widely used optimisation for programming languages in general and constraint modelling in particular is common subexpression elimination (CSE). A compiler detects when the same expression (modulo some equivalences) appears several times in a model, and automatically keeps only one of these expressions, replacing all others by a reference to it. For example, the division in the following code occurs twice:

```
constraint (x div y) + a = 0;
constraint b * (x div y) = c;
```

So an equivalent model would hoist the division to the top level:

```
var int: tmp = x div y;
constraint tmp + a = 0;
constraint b * tmp = c;
```

User-defined functions increase the potential for automatic CSE because they introduce **syntactic equalities**. One could introduce and detect the same equalities without functional notation, by e.g. annotating “functional predicates”. However, this would not simplify the translation presented in section 5 that performs CSE while maintaining the relational semantics, but it would make the syntax much less convenient.

CSE is particularly effective if the function definition is complex, it introduces additional variables, or its constraints are expensive to propagate. CSE also makes the abstraction facilities in a language more useful, as common subexpressions will be detected across different functions.

2 Adding Functions and Local Constraints to MiniZinc

We extend MiniZinc by adding functions, using the Zinc [10] syntax for functions. The other important extension to MiniZinc, not currently supported by Zinc, is to allow constraints to occur inside `let` constructs.

The most basic form of user-defined functions would be a simple macro mechanism that can be used to define abbreviations for functional compositions. For instance, we could define a function for the Manhattan distance between two points as

```
function var int: manhattan(var int:x1, var int:y1,
                            var int:x2, var int:y2) =
    abs(x2 - x1) + abs(y2 - y1);
```

Such a macro mechanism would be useful by itself and straightforward to implement; both CSE and the relational semantics would be ensured by the translation of primitive functions. However, as soon as we permit `let` expressions that introduce new variables and constraints in the function body, we have to define how these are translated in reified contexts. This is the main contribution of this paper.

Suppose we wished to add a `sqr` function to MiniZinc that squares its argument.[3] While we could do this without using local variables

```
function var int: sqr(var int:x) = x * x;
```

we can make the model propagate stronger using the following definition

[3] This does not preclude solvers from using a more efficient built-in version of `sqr`, as they can simply override the standard definition in their solver-specific MiniZinc library.

```
function var int: sqr(var int: x)  =
  let { var int: y = x * x; constraint y >= 0 } in y;
```

which explicitly adds that the result is non-negative.

Note that we assume that functions defined in MiniZinc are indeed pure (always give the same answer for each input). It is certainly possible to write impure functions:

```
function var int: f(var int:x) = let { var 0..1: b } in b*x;
```

We consider this a modelling error – non-functional relations like this must be expressed using predicates in MiniZinc. As purity analysis is difficult, our implementation does not enforce purity but results in undefined behaviour of impure functions. We plan to investigate a simple but incomplete purity analysis in future work.

3 Using Functions

In this section we illustrate the modelling possibilities that arise from the introduction of user-defined functions.

Big Data. The bigger the data sets involved, the more crucial it is to get common subexpression elimination. The motivating example that made us add functions to MiniZinc arose from modelling data mining problems [7].

An item set mining problem consists of a large data base of *transactions* `TDB`, which is a list of sets of *items* (e.g., items that were bought together in one transaction from a supermarket). Each transaction has an integer identifier.

An important concept in item set mining is the *cover* of an item set, the set of labels of transactions in which the item set occurs. It can be defined naturally as a function:

```
function var set of int: cover(var set of int: Items,
        array[int] of set of int: TDB) =
    let {  var set of index_set(TDB): Trans;
           constraint forall (t in index_set(TDB)) (
                          t in Trans <-> Items subset TDB[t])
    } in Trans;
```

Constraints involving the cover could restrict its size to at least *k* (*frequent* item set mining), or require item sets to be *closed* (i.e., maximal).

With `TDB` being a huge data base, it would have a catastrophic impact on translation and solver performance if we did not get CSE for different calls to `cover`. Without support for functions, we can only lift out the definition of `cover`, performing CSE by hand. The result is a much less readable model, and indeed a loss in compositionality, as predicates that use `cover` can no longer be defined in a self-contained way.

Functional Global Constraints. The global constraint catalog [3] lists 120 constraints (almost a third) as functional in nature, such as `cycle`, `change`, `common`, `distance`, `global_cardinality`, `graph_crossing`, `indexed_sum`, `path`.

We can define functional versions that map to the global constraints, which yields natural models and gives the MiniZinc compiler important hints for performing common subexpression elimination.

In a model for the Warehouse Location Problem (CSPLib 034) we need to constrain the number of stores supplied by each warehouse. We can use the global constraint `count`(x,i,c) which constrains c to be the number of times that the value i appears in the array x. Assuming an array `supplier` mapping each store to its supplying warehouse, the function `inUse`(i) returns the number of stores supplied by warehouse i.

```
function var int: inUse(int: i) =
  let { var int: use;
        constraint count(supplier,i,use) } in use;
constraint forall (i in index_set(supplier))
    ( inUse(i) <= capacity[i] );
```

Using MiniZinc functions we do not have to introduce auxiliary variables or perform CSE by hand in our model.

In recent work on reifying global constraints [2], functional global constraints play a key role, as they permit decomposition of globals into a functionally defined part that need not be reified, and a part that is easy to reify. User-defined functions make these techniques immediately accessible in MiniZinc.

Functional Tables. A convenient way to model an ad-hoc partial function is using a `table` constraint, but doing so hides the fact that the constraint is functional.

The following model fragment defines a partial function `partner` using a list of pairs `pairs`. This is used in a complex constraint (with common subexpressions).

```
array[1..10] of var 1..10: p;
array[1..10] of var bool: notused;
array[1..2][1..6] of int: pairs =
      [ 1, 4 | 2, 1 | 5, 1 | 7, 10 | 8, 3 | 10, 2  ];
function var int: partner(var int:x) =
  let { var int: r;
        constraint table([x,r], pairs) } in r;
constraint forall(i in 1..10)( notused[i] \/
  partner(p[i]) > i \/ partner(partner(p[i])) > i );
```

The function definition allows the MiniZinc compiler to detect the common subexpressions, and the user can simply use the partial function as they intended. Note that `partner` creates a new variable and a `table` constraint in a reified context. Without functions, CSE as well as guarding for partiality has to be performed manually:

```
constraint forall(i in 1..10)(
   let { var int: p_pi;
         var int: p_p_pi;
         var bool: b1 = table([p[i],p_pi],pairs);
         var bool: b2 = table([p_pi,p_p_pi], pairs);
   } in notused[i] \/ (b1 /\ p_pi > i) \/
        (b1 /\ b2 /\ p_p_pi > i));
```

This manual approach is not *compositional*. MiniZinc encourages the use of predicate definitions, which is incompatible with manual, whole-program CSE.

Solver specific translation / channelling. Solver independent models in MiniZinc are translated to solver-specific versions using custom predicate definitions. In many cases these definitions need to introduce new variables that are functionally defined in terms of the original variables. We can use functions to make this straightforward and automatically achieve CSE if the translation is required more than once.

Consider the mapping of not equals for an integer programming solver. We can define a function `int2array01` such that `int2array01(x)[i]=1` iff `x=i`. Using this function, the not equals constraint is straightforward to define[4]:

```
predicate int_neq(var int: x, var int:y) = let {
    array[int] of var 0..1: bx = int2array01(x);
    array[int] of var 0..1: by = int2array01(y);
} in forall(i in max(lb(x),lb(y))..min(ub(x),ub(y)))
            (bx[i] + by[i] <= 1);
```

Crucially, CSE guarantees that for any integer variable, at most one array of 01 variables is created, so that we get an efficient translation of constraints that share variables such as `int_neq(x,y) /\ int_neq(x,z)`. Without functions, an efficient abstraction like `int_neq` would not be possible, and the user would be required to carefully avoid introducing common subexpressions while modelling.

The above encoding generalises easily to the *alldifferent* constraint on an array of variables. We can use similar mechanisms for translating integer and set models for SAT solvers, and for channelling constraints between two different viewpoints of a model, such as a set-based model and a Boolean model where for each set variable x we introduce an array of Boolean variables b such that $i \in x \Leftrightarrow b[i]$.

Simplifying MiniZinc Translation. Once we have generic methods for handling functions, many of the built-in functions in MiniZinc can be simply treated as library functions. We can then use the generic methods for flattening. This simplifies the flattening process and makes it more transparent and adaptable.

Consider the built-in in `abs` function. We can remove it from MiniZinc and add the following library definition.

```
function var int: abs(var int: x) =
  let { var int: y; constraint int_abs(x,y) } in y;
```

The handling of this function will automatically create the FlatZinc constraint `int_abs` for any absolute value expressions.[5] Built-in functions that can be relegated in this way include: the trigonometric functions, their inverses and hyperbolic versions, exponentiation and logarithms, amongst others.

4 Partial and Total Functions and Negation

Negative Boolean contexts require particular care in constraint programming systems with local variables. Consider

[4] The `lb` and `ub` functions return a guaranteed lower and upper bound of a variable (usually the bound declared in the model).

[5] To be used in all contexts it needs to be annotated as `total` as explained in the next section.

```
function var int: evendiv2(var int: x) =
  let { var int: y; constraint x = 2*y } in y;
constraint not evendiv2(x)=1;
```

A logical interpretation would be $\neg\exists y. x = 2 \times y \wedge y = 1$ or $\forall y. x \neq 2 \times y \vee y \neq 1$. However, we cannot easily express universal quantification, and a naive translation of the above (simply ignoring the universal quantifier) would produce the following:

```
var int: y;
constraint not (x = 2*y /\ y = 1);
```

This does not have the desired semantics, as it permits for example the solution $x = 2, y = 3$, although $x = 2$ clearly should not be a solution. MiniZinc (and Zinc) therefore consider models illegal that contain `let` expressions in a negative context which introduce non-functionally defined variables (like y in the above example).

The upshot of this is that user-defined functions become almost useless in negative contexts, since interesting functions can usually only be defined by introducing new variables (as in most of our examples). The following technique avoids this problem.

Totality Annotations. Functions with local variables can be used safely in negative contexts if they are *total*, that is they are defined on all their inputs. Consider the following example:

```
function var int: g(var int: x) :: total =
  let { var int: y;
        constraint (x > 0)  -> y = x;
        constraint (x <= 0) -> y = 10-x
  } in y;
var -10..10: u;
constraint not g(u) = 5;
```

Even though syntactically y is not functionally defined by the function, it is semantically. It is safe to translate the last line as

```
var int: y = g(u);       % evaluate g in root context
constraint not y = 5;    % only negate the equality
```

To allow the user to declare total functions, we introduce the annotation `total` which can be added to function definitions as seen above. The annotation `total` promises that the function is total for all uses, and that it does not constrain its arguments. The function will be translated *in the root context*, which means that free variable definitions are allowed. If a user annotates a function as total that is in fact partial, this effectively results in the function being translated according to the strict semantics. We do not, at this point, attempt to detect incorrect totality annotations automatically.

Recipe for partial functions. Totality annotations enable us to define some partial functions that introduce variables, by following a simple **recipe**. Assume that we want to implement a partial function $f(x)$, and that we can express its domain of definition by a constraint $c(x)$ that does not introduce any free variables. Then

1. create a *total extension* f' such that $f'(x) = f(x)$ if $c(x)$, and $f'(x) = 0$ otherwise. We annotate f' as `::total`, so it can introduce arbitrary free variables.
2. create a *partial guard function* $g(x) =$ `let {` $c(x)$ `} in` $f'(x)$ `;`

Consider for example `evendiv2` as defined above. It can be rewritten as

```
function var int: evendiv2(var int: x) =
  let { constraint x mod 2 = 0 } in safe_ed2(x);
function var int: safe_ed2(var int: x) :: total =
  let { var int: y;
        constraint x mod 2 = 0 -> x = 2*y;
        constraint not (x mod 2 = 0) -> y = 0} in y;
```

Now `evendiv2` is a partial function, but does not introduce variables which are not functionally defined, and `safe_ed2` is a total function, which is the same as the original `evendiv2` when $x \mod 2 = 0$. The constraint `not evendiv2(x)=1` now translates as

```
var int: y;
constraint safe_ed2(x)=y;
constraint not (x mod 2 = 0 /\ y = 1);
```

Now $x = 2$ enforces $y = 1$, so the negation fails and only correct solutions are returned.

We use the same mechanism for the translation of built-in partial functions `div` and array access, assuming built-in total functions *safediv* and *safeelement*.

If we can express the implicit constraint $c(x)$ of a partial function on its input arguments without introducing new variables, we can freely use the partial function in any context. Of course, for some functions, $c(x)$ may be difficult to express, so this mechanism is no general solution to the problem. This is not surprising, as otherwise we could translate models with arbitrary quantifiers efficiently to MiniZinc.

5 Translation to FlatZinc

Constraint problems formulated in MiniZinc are solved by translating them to a simpler, solver-specific subset of MiniZinc, called FlatZinc. This section shows how to translate our extended MiniZinc to FlatZinc.

The complexities in the translation arise from the need to simultaneously (a) unroll array comprehensions (and other loops), (b) replace predicate and function applications by their body, and (c) flatten expressions.

Once we take into account CSE, we cannot perform these separately. In order to have names for common subexpressions we need to flatten expressions. And in order to take advantage of functions for CSE we cannot replace predicate and function applications without flattening to generate these names. And without replacing predicate and function application by their body we are unable to see all the loops to unroll.

The translation algorithm presented below generates a flat model equivalent to the original model as a global set of constraints S. We ignore the collection of variable declarations, which is also clearly important, but quite straightforward. The translation uses full reification to create the model. It can be extended to use half reification [4], but we omit this for space reasons. Common subexpression elimination is implemented using a technique similar to *hash-consing* in Lisp [1]. For simplicity we only show syntactic CSE, which eliminates expressions that are identical after parameter replacement. The extension to semantic CSE, using commutativity and other equivalences, is well understood (see [12] for a detailed discussion) but makes the pseudo-code much longer.

MiniZinc Syntax. Below is a grammar for a subset of MiniZinc as it currently stands, with enough complexity to illustrate all the main challenges in extending it to include functions. The cons nonterminal defines constraints (Boolean terms), term defines integer terms, barray defines Boolean arrays, and iarray defines integer arrays:

```
  cons ⟶ true | false | bvar | term relop term
       ⟶ not cons | cons /\ cons | cons \/ cons | cons -> cons | cons <-> cons
       ⟶ forall barray | exists barray | barray[term]
       ⟶ pred(term, ..., term ) | if cons then cons else cons endif
       ⟶ let { decls } in cons
  term ⟶ int | ivar | term arithop term | iarray[term] | sum iarray
       ⟶ if cons then term else term endif
       ⟶ let { decls } in term
barray ⟶ [ cons, ..., cons ] | [ cons | ivar in term .. term ]
iarray ⟶ [ term, ..., term ] | [ term | ivar in term .. term ]
```

The grammar uses the symbols bvar for Boolean variables, relop for relational operators { ==, <=, <, !=, >=, > }, pred for names of predicates, int for integer constants, ivar for integer variables, and arithop for arithmetic operators { +, -, *, div }.

In the `let` constructs we make use of the nonterminal decls for declarations. We define this below using idecl for integer variable declarations, bdecl for Boolean variable declarations. We also define args as a list of integer variable declarations, an item item as either a predicate declaration or a constraint, items as a list of items, and a model model as some declarations followed by items. Note that ε represents the empty string.

```
idecl ⟶ int : ivar | var int : ivar | var term .. term : ivar
bdecl ⟶ bool : bvar | var bool : bvar
decls ⟶ ε | idecl ; decls | idecl = term ; decls | bdecl ; decls | bdecl = cons ; decls
 args ⟶ var int : ivar | var int : ivar , args | int : ivar | int : ivar , args
 item ⟶ predicate pred( args) = cons ; | constraint cons ;
items ⟶ ε | item items
model ⟶ decls items
```

To simplify presentation, we assume all predicate arguments are integers, but the translation can be extended to arbitrary arguments in a straightforward way.

To introduce functions in MiniZinc, the grammar above is modified as follows. We add new item types for integer variable and integer parameter functions (again for simplicity all arguments are assumed to be integers). We change the form of the `let` construct for both constraints and terms to allow optional constraints (ocons).

```
 item ⟶ function var int : func ( args ) = term ; |
      ⟶ function int : func ( args ) = term ;
 cons ⟶ let { decls ocons } in cons
 term ⟶ let { decls ocons } in term
ocons ⟶ ε | ; constraint cons ocons
```

Further notation. Given an expression e that contains the subexpression x, we denote by $e[\![x/y]\!]$ the expression that results from replacing all occurrences of subexpression x by expression y. We will also use the notation for multiple simultaneous replacements $[\![\bar{x}/\bar{y}]\!]$ where each $x_i \in \bar{x}$ is replaced by the corresponding $y_i \in \bar{y}$.

Given a cons term defining the constraints of the model we can split its cons subterms as occurring in different kinds of places: root contexts, positive contexts, negative contexts, and mixed contexts. A Boolean subterm t of constraint c is in a *root context* iff there is no solution of $c[\![t/\mathit{false}]\!]$, that is c with subterm t replaced by *false*.[6] Similarly, a subterm t of constraint c is in a *positive context* iff for any solution θ of c then θ is also a solution of $c[\![t/\mathit{true}]\!]$; and a *negative context* iff for any solution θ of c then θ is also a solution of $c[\![t/\mathit{false}]\!]$. The remaining Boolean subterms of c are in *mixed* contexts. While determining contexts according to these definitions is hard, there are simple syntactic rules which can determine the correct context for most terms, and the rest can be treated as mixed. Consider the constraint expression

```
constraint x > 0 /\ (i <= 4 -> x + bool2int(b) = 5);
```

then $x > 0$ is in the root context, $i \leq 4$ is in a negative context, $x + \texttt{bool2int}(b) = 5$ is in a positive context, and b is in a mixed context. If the last equality were $x + \texttt{bool2int}(b) \geq 5$ then b would be in a positive context.

Flattening Constraints. Flattening a constraint c in context *ctxt*, $\mathsf{flatc}(c, \mathit{ctxt})$, returns a Boolean literal b representing the constraint and as a side effect adds a set of constraints (the flattening) S to the store such that such that $S \models b \Leftrightarrow c$.

It uses the context (*ctxt*) to decide how to translate, where possible contexts are: *root*, at the top level conjunction; *pos*, positive context, *neg*, negative context, and *mix*, mixed context. We use the context operations $+$ and $-$ defined as:

$$\begin{array}{llll} +\mathit{root} = \mathit{pos} & +\mathit{neg} = \mathit{neg}, & -\mathit{root} = \mathit{neg} & -\mathit{neg} = \mathit{pos}, \\ +\mathit{pos} = \mathit{pos} & +\mathit{mix} = \mathit{mix} & -\mathit{pos} = \mathit{neg} & -\mathit{mix} = \mathit{mix} \end{array}$$

Note that flattening in the root context always returns a Boolean b made equivalent to *true* by the constraints in S. For succinctness we use the notation **new** b (**new** v) to introduce a fresh Boolean (resp. integer) variable and return the name of the variable.

The Boolean result of flattening c is stored in a hash table, and reused if an identical constraint expression is ever flattened again. If the context for an expression e is root, the result of a successful hash should be *true*. If we have a common subexpression e in another context, then since it is true at the root it is true there. If we first meet the expression e in a non-root context and later meet expression e at the root, we simply need to set the Boolean created for the first met version to *true*. This is the role of the

[6] For the definitions of context we assume that the subterm t is uniquely defined by its position in c, so the replacement is of exactly one subterm.

addition to S on the first line of flatc. Note that in this simplified presentation, if an expression introduces a fresh variable and it appears first in a negative context and only later in the root context, the translation aborts. This can be fixed in a preprocessing step that sorts expressions according to their context.

The flattening proceeds by evaluating fixed Boolean expressions and returning the value. We assume fixed checks if an expression is fixed (determined during MiniZinc's type analysis), and eval evaluates a fixed expression. For simplicity of presentation, we assume that fixed expressions are never undefined.

For non-fixed expressions we treat each form in turn. Boolean literals and variables are simply returned. Basic relational operations flatten their terms using the function flatt, which for a term t returns a tuple $\langle v, b\rangle$ of an integer variable/value v and a Boolean literal b such that $b \Leftrightarrow (v = t)$ (described in detail below). The relational operations then return a reified form of the relation. The logical operators recursively flatten their arguments, passing in the correct context. The logical array operators evaluate their array argument, then create an equivalent term using foldl and either `/\` or `\/` which is then flattened. A Boolean array lookup flattens its arguments, and creates a *safeelement* constraint (which does not constrain the index variable) and Boolean b'' to capture whether the array lookup was safe. Built-in predicates abort if not in the root context. They flatten their arguments and add an appropriate built-in constraint. User defined predicate applications flatten their arguments and then flatten a renamed copy of the body. *if-then-else* evaluates the condition (which must be fixed) and flattens the *then* or *else* branch appropriately. The handling of `let` is the most complicated. The expression is renamed with new copies of the let variables. We extract the constraints from the `let` expression using function flatlet which returns the extracted constraint and a rewritten term (not used in this case, but used in flatt). The constraints returned by function flatlet are then flattened. Finally if we are in the root context, we ensure that the Boolean b returned must be *true* by adding b to S.

```
flatc($c$,$ctxt$)
  $h$ := hash[$c$]; **if** ($h \neq \bot$) $S \cup:= \{(ctxt = root) \Rightarrow h\}$; **return** $h$
  **if** (fixed($c$)) $b$ := eval($c$)
  **else**
    **switch** $c$
    **case** $b'$ (bvar): $b := b'$;
    **case** $t_1\ r\ t_2$ (relop): $\langle v_1, b_1\rangle$ := flatt($t_1, ctxt$); $\langle v_2, b_2\rangle$ := flatt($t_2, ctxt$);
      $S \cup:= \{$**new** $b \Leftrightarrow (b_1 \land b_2 \land$ **new** $b'), b' \Leftrightarrow v_1\ r\ v_2\}$
    **case** not $c_1$: $b := \neg$flatc($c_1, -ctxt$)
    **case** $c_1$ /\ $c_2$: $S \cup:= \{$**new** $b \Leftrightarrow$ (flatc($c_1, ctxt$) $\land$ flatc($c_2, ctxt$))$\}$
    **case** $c_1$ \/ $c_2$: $S \cup:= \{$**new** $b \Leftrightarrow$ (flatc($c_1, +ctxt$) $\lor$ flatc($c_2, +ctxt$))$\}$
    **case** $c_1$ -> $c_2$: $S \cup:= \{$**new** $b \Leftrightarrow$ (flatc($c_1, -ctxt$) $\Rightarrow$ flatc($c_2, +ctxt$))$\}$
    **case** $c_1$ <-> $c_2$: $S \cup:= \{$**new** $b \Leftrightarrow$ (flatc($c_1, mix$) $\Leftrightarrow$ flatc($c_2, mix$))$\}$
    **case** forall $ba$: $b$ := flatc( foldl(evala($ba$), true, /\), $ctxt$)
    **case** exists $ba$: $b$ := flatc( foldl(evala($ba$), false, \/), $ctxt$)
    **case** $[c_1, \ldots, c_n]$ [ $t$ ] : **foreach**($j \in 1..n$) $b_j$ := flatc($c_j, +ctxt$); $\langle v, b_{n+1}\rangle$ := flatt($t, ctxt$);
      $S := \{$**new** $b \Leftrightarrow (b_{n+1} \land$ **new** $b' \land$ **new** $b''), safeelement(v, [b_1, \ldots, b_n], b'), b'' \Leftrightarrow v \in \{1, .., n\}\}$
    **case** $p$ $(t_1, \ldots, t_n)$ (pred) built-in predicate:
      **if** ($ctxt \neq root$) **abort**
      **foreach**($j \in 1..n$) $\langle v_j, \_\rangle$ := flatt($t_j, ctxt$);
      $b$ := $true$; $S \cup:= \{p(v_1,\ldots,v_n)\}$
    **case** $p$ $(t_1,\ldots,t_n)$ (pred): user-defined predicate
      **let** $p(x_1,\ldots,x_n) = c_0$ be defn of $p$
      **foreach**($j \in 1..n$) $\langle v_j, b_j\rangle$ := flatt($t_j, ctxt$);
      **new** $b'$ := flatc($c_0[\![x_1/v_1,\ldots,x_n/v_n]\!], ctxt$)
      $S \cup:= \{$**new** $b \Leftrightarrow b' \wedge \bigwedge_{j=1}^{n} b_j\}$
    **case** `if` $c_0$ `then` $c_1$ `else` $c_2$ `endif`: **if** (eval($c_0$)) $b$ := flatc($c_1, ctxt$) **else** $b$ := flatc($c_2, ctxt$)
    **case** `let { ` $d$ ` } in` $c_1$:
      **let** $\bar{v}'$ be a renaming of variables $\bar{v}$ defined in $d$
      $\langle c', \_\rangle$ := flatlet($d[\![\bar{v}/\bar{v}']\!], c_1[\![\bar{v}/\bar{v}']\!], 0, ctxt$)
      $b$ := flatc($c', ctxt$);
  **if** ($ctxt = root$) $S \cup:= \{b\}$;
  hash[$c$] := $b$
  **return** $b$
```

The function evala replaces an array comprehension by the resulting array. Note that terms *lt* and *ut* must be fixed in a correct MiniZinc model.

```
evala($t$)
  **switch** $t$
  **case** `[` $t_1,\ldots,t_n$ `]`: **return** `[` $t_1,\ldots,t_n$ `]`
  **case** `[` $e$ `|` $v$ `in` $lt$ `..` $ut$ `]`: **let** $l$ = eval($lt$), $u$ = eval($ut$)
    **return** `[` $e[\![v/l]\!], e[\![v/l+1]\!], \ldots e[\![v/u]\!]$ `]`
```

Handling `let` expressions. Much of the handling of a `let` is implemented by flatlet$(d,c,t,ctxt)$ which takes the declarations d inside the `let`, the constraint c or term t of the scope of the `let` expression, as well as the context type. First flatlet replaces parameters defined in d by their fixed values. Then it collects in c all the constraints that need to stay within the Boolean context of the `let`: the constraints arising from the variable and constraint items, as well as the Boolean variable definitions. Integer variables that are defined have their right hand side flattened, and a constraint equating them to the right hand side t' added to the global set of constraints S. If variables are not defined and the context is negative or mixed the translation aborts.

```
flatlet($d,c,t,ctxt$)
  **foreach** ($item \in d$)
    **switch** $item$
    **case** `int :` $v = t'$ :        $d$ := $d[\![v/\text{eval}(t')]\!]$;  $c$ := $c[\![v/\text{eval}(t')]\!]$;  $t$ := $t[\![v/\text{eval}(t')]\!]$
    **case** `bool :` $b = c'$ :       $d$ := $d[\![b/\text{eval}(c')]\!]$; $c$ := $c[\![b/\text{eval}(c')]\!]$; $t$ := $t[\![b/\text{eval}(c')]\!]$
  **foreach** ($item \in d$)
    **switch** $item$
    **case** `var int :` $v$ :         **if** ($ctxt \in \{neg, mix\}$) **abort**
    **case** `var int :` $v = t'$ :    $\langle v', b'\rangle$ := flatt($t', ctxt$); $S \cup:= \{v = v'\}$; $c$ := $c$ `/\` $b'$
    **case** `var` $l$ `..` $u$ `:` $v$ :       **if** ($ctxt \in \{neg, mix\}$) **abort else** $c$ := $c$ `/\` $l$ `<=` $v$ `/\` $v$ `<=` $u$
    **case** `var` $l$ `..` $u$ `:` $v = t'$ :  $\langle v', b'\rangle$ := flatt($t', ctxt$); $S \cup:= \{v = v'\}$;
                                  $c$ := $c$ `/\` $b'$ `/\` $l$ `<=` $v$ `/\` $v$ `<=` $u$;
    **case** `var bool :` $b$ :        **if** ($ctxt \in \{neg, mix\}$) **abort**
    **case** `var bool :` $b = c'$ :   $c$ := $c$ `/\` ($b$ `<->` $c'$)
    **case** `constraint` $c'$ :        $c$ := $c$ `/\` $c'$
  **return** $\langle c, t\rangle$
```

Flattening integer expressions. flatt$(t, ctxt)$ flattens an integer term t in context *ctxt*. It returns a tuple $\langle v, b\rangle$ of an integer variable/value v and a Boolean literal b, and as a side effect adds constraints to S so that $S \models b \Leftrightarrow (v = t)$. Note that again flattening in the root context always returns a Boolean b made equivalent to *true* by the constraints in S.

Flattening first checks if the same expression has been flattened previously and if so returns the stored result. flatt evaluates fixed integer expressions and returns the result. For non-fixed integer expressions t each form is treated in turn. Simple integer expressions are simply returned. Operators have their arguments flattened and the new value calculated on the results. Safe division (which does not constrain its second argument to be non-zero) is used for division with the constraint being captured by the new Boolean b'. Array lookup flattens all the integer expressions involved and creates a *safeelement* constraint as in the Boolean case. `sum` expressions evaluate the array argument, and then replace the sum by repeated addition using foldl and flatten that. *if-then-else* simply evaluates the *if* condition (which must be fixed) and flattens the *then* or *else* branch appropriately. Functions are simply handled by flattening each of the arguments, the function body is then renamed to use the variables representing the arguments, and the body is then flattened. Importantly, if the function is declared total it is flattened *in the root context*. Let constructs are handled analogously to flatc. We rename the scoped term t_1 to t' and collect the constraints in the definitions in c'. The result is the flattening of t', with b capturing whether anything inside the let leads to failure.

```
flatt($t$,$ctxt$)
  $\langle h,b'\rangle$ := hash[$t$]; if ($h \neq \bot$) $S$ ∪:= $\{(ctxt = root) \Rightarrow b'\}$; return $\langle h,b'\rangle$
  if (fixed($t$)) $v$ := eval($t$); $b$ := $true$
  else switch $t$
    case $v'$ (ivar): $v$ := $v'$; $b$ := $true$
    case $t_1$ $a$ $t_2$ (arithop): $\langle v_1,b_1\rangle$ := flatt($t_1$,$ctxt$); $\langle v_2,b_2\rangle$ := flatt($t_2$,$ctxt$);
      if ($a$ is not div) $S$ ∪:= $\{\textbf{new}\ b \Leftrightarrow (b_1 \wedge b_2), a(v_1,v_2,\textbf{new}\ v)\}$
      else $S$ ∪:= $\{\textbf{new}\ b \Leftrightarrow (b_1 \wedge b_2 \wedge \textbf{new}\ b'), safediv(v_1,v_2,\textbf{new}\ v), b' \Leftrightarrow v_2 \neq 0\}$
    case $[t_1,\ldots,t_n]$ [ $t_0$ ] : foreach($j \in 0..n$) $\langle v_j,b_j\rangle$ := flatt($t_j$,$ctxt$);
      $S$ := $\{\textbf{new}\ b \Leftrightarrow (\textbf{new}\ b' \wedge \bigwedge_{j=0}^{n} b_j), safeelement(v_0,[v_1,\ldots,v_n],\textbf{new}\ v), b' \Leftrightarrow v_0 \in \{1,...,n\}\}$
    case sum $ia$: $\langle v,b\rangle$ := flatt( foldl(evala($ia$), 0, +), $ctxt$)
    case if $c_0$ then $t_1$ else $t_2$ endif: if ($eval(c_0)$) $\langle v,b\rangle$ := flatt($t_1$,$ctxt$) else $\langle v,b\rangle$ := flatt($t_2$,$ctxt$)
    case $f$ $(t_1,\ldots,t_n)$ (func): function
      foreach($j \in 1..n$) $\langle v_j,b_j\rangle$ := flatt($t_j$,$ctxt$);
      let $f(x_1,\ldots,x_n) = t_0$ be defn of $f$
      if ($f$ is declared total) $ctxt' = root$ else $ctxt' = ctxt$
      $\langle v,b'\rangle$ := flatt($t_0[\![x_1/v_1,\ldots,x_n/v_n]\!]$,$ctxt'$)
      $S$ ∪:= $\{\textbf{new}\ b \Leftrightarrow b' \wedge \bigwedge_{j=1}^{n} b_j\}$
    case let { $d$ } in $t_1$:
      let $\bar{v}'$ be a renaming of variables $\bar{v}$ defined in $d$
      $\langle c',t'\rangle$ := flatlet($d[\![\bar{v}/\bar{v}']\!]$,true,$t_1[\![\bar{v}/\bar{v}']\!]$,$ctxt$)
      $\langle v,b_1\rangle$ := flatt($t'$,$ctxt$); $b_2$ := flatc($c'$,$ctxt$); $S$ ∪:= $\{\textbf{new}\ b \Leftrightarrow (b_1 \wedge b_2)\}$
  if ($ctxt = root$) $S$ ∪:= $\{b\}$;
  hash[$t$] := $\langle v,b\rangle$
  return $\langle v,b\rangle$
```

Consider the translation of the code

```
function var int: h(var int: a) =
  let { var int: d = 12 div a; constraint d < 3 } in d;
constraint not (h(c) = 1);
```

Flattening of `not (h(c) = 1)` requires flattening the term `h(c)` in a *neg* context. This requires flattening `let { var int: d = 12 div c; constraint d < 3 } in d;` in a negative context. This requires flattening `12 div c` in a negative context which creates the constraints $\texttt{safediv}(12,c,v), b_1 \Leftrightarrow c \neq 0$ and returns $\langle v, b_1 \rangle$. Flattening the let adds $d = v$ and collects b_1 `/\ d < 3` in the constraints c which are flattened to give $b_3 \Leftrightarrow (d < 3), b_4 \Leftrightarrow (b_3 \wedge b_1)$ and returns tuple $\langle d, b_4 \rangle$. The treatment of the equality adds $b_5 \Leftrightarrow d = 1, b_6 \Leftrightarrow (b_5 \wedge b_4)$ and returns b_6. The negation returns $\neg b_6$ and asserts $\neg b_6$.

Implementation. We have implemented the above rules in a compiler that turns MiniZinc with functions into FlatZinc. This prototype can handle the complete grammar as presented here, extended with support for arbitrary function arguments (arrays, Booleans, sets). Compared to the existing `mzn2fzn` translator, most built-in functions (such as `abs`, `bool2int`, `card`) are now realised as user-defined functions in the MiniZinc standard library rather than hard coding them into the translation.

The following table shows some experimental results obtained with the prototype. We compiled five different instances of a standard model for a 16x16 Sudoku problem. Each instance has 48 *alldifferent* constraints, and we used the linearisation presented in section 3 to show the effect of common subexpression elimination. Crucially, the *alldifferent* constraints in Sudoku puzzles *overlap*, many pairs of constraints share either one or four variables. The standard decomposition without functions cannot take advantage of this sharing across constraints, as the results below will show.

Benchmark	*Translation (s)*		*# Cons*		*Solving (s)*		
	fn	*nofn*	*fn*	*nofn*	*fn*	*nofn*	*speedup*
Sudoku 1 (16x16)	0.2	0.3	1280	2304	0.24	0.78	3.33
Sudoku 2 (16x16)	0.2	0.3	1280	2304	0.29	3.45	11.96
Sudoku 3 (16x16)	0.2	0.3	1280	2304	0.32	15.32	47.78
Sudoku 4 (16x16)	0.2	0.3	1280	2304	0.24	0.43	1.80
Sudoku 5 (16x16)	0.2	0.3	1280	2304	0.34	6.07	17.72

The columns labelled *fn* present the results for the new prototype translator using a decomposition of *alldifferent* based on functions. The *nofn* columns use the existing `mzn2fzn` tool with the standard MiniZinc linearisation library (available with the command line option `-G linear`).

Translation time does not suffer from the additional CSE, as the results in column *Translation* show. In fact, the new translator is slightly faster. CSE clearly reduces the number of constraints by almost half (column *# Cons*). The solving time (column *Solving*) is the average of 20 runs of the Gurobi MIP solver [8] on a 3.4 GHz Intel Core i7, running on a single core under Windows. The additional, redundant constraints in *nofn* cause a noticeable overhead.

We will extend our prototype into a full replacement for the current `mzn2fzn`, working towards version 2.0 of the MiniZinc language and toolchain. The prototype and the benchmark problems are available from the authors upon request.

6 Related Work and Conclusion

The only modelling language that supports functions other than MiniZinc that we are aware of is Zinc [10]. Zinc functions share the same restrictions as MiniZinc functions on not allowing new variable definitions in negative or mixed contexts. Constraints in let expression and `total` annotations are (currently) not part of Zinc. This means the functions in MiniZinc are much more expressive than those in Zinc. Almost all examples in this paper make use of constraints in let expressions to define functions. Of course we intend to extend Zinc as we have extended MiniZinc. But a further difficulty arises as Zinc models are compiled without instance data. This means that the treatment of partiality is much more complex in Zinc, since we must lift local variable definitions to the root context, without lifting any failure that they may cause. This also precludes most CSE, since the Zinc compiler can only detect common subexpressions that occur at the model level, rather than after instantiation with the data.

Constraint-based local search languages such as Comet [15] support user-defined *objective functions*. These cannot be used as arguments in other constraints, and therefore these systems do not deal with partiality, Boolean contexts, or CSE.

Other modelling languages such as OPL [14] and Essence [6] do not include user-defined functions or local variables (let expressions), and hence the issues that we consider here do not arise. Essence supports constrained *function variables*, which are used to model problems whose *solution* is a function. Our user-defined functions, in contrast, serve a different purpose, they express the *constraints* of a problem. The approach to flattening and CSE for Essence is described in [12], but Essence includes neither `let` expressions, predicates or functions which are the main complicating features herein. This earlier work [12] showed the importance of CSE for modelling languages, and this is only magnified by the introduction of user-defined functions.

Modelling languages incorporated in a procedural OO language, such as IBM ILOG Concert [9] expressions, Gecode's [13] MiniModel expressions or Comet's [15] modelling constructs, do allow the use of functions in the *host language*. The problem is that such functions do not extend the modelling language, and if treated in this manner define the *strict* semantics.

In conclusion the addition of user-defined functions, local constraints in let constructs, and totality annotations gives a powerful modelling capability to MiniZinc. Using our schema for separating a partial function into a total extension with local variable introduction and a partial function with no local variables, user-defined functions are usable in all parts of a model. We believe functional modelling will become more and more commonplace, particularly given the prevalence of functional constraints in the global constraint catalog, given the importance of abstraction for defining solver-specific MiniZinc libraries, and given that functions can be implemented efficiently as shown in this paper.

Acknowledgements. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

References

1. Allen, J.: Anatomy of LISP. McGraw-Hill, Inc., New York, NY, USA (1978)
2. Beldiceanu, N., Carlsson, M., Flener, P., Pearson, J.: On the reification of global constraints. Constraints (2012), DOI: 10.1007/s10601-012-9132-0
3. Beldiceanu, N., Carlsson, M., Rampon, J.X.: Global constraint catalog, working version as of December 2012. `http://www.emn.fr/z-info/sdemasse/gccat/`
4. Feydy, T., Somogyi, Z., Stuckey, P.: Half-reification and flattening. In: Lee, J. (ed.) Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 6876, pp. 286–301. Springer (2011)
5. Frisch, A., Stuckey, P.: The proper treatment of undefinedness in constraint languages. In: Gent, I. (ed.) Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 5732, pp. 367–382. Springer (2009)
6. Frisch, A.M., Harvey, W., Jefferson, C., Hernández, B.M., Miguel, I.: Essence : A constraint language for specifying combinatorial problems. Constraints 13(3), 268–306 (2008)
7. Guns, T.: Declarative Pattern Mining using Constraint Programming. Ph.D. thesis, Department of Computer Science, K.U.Leuven (2012)
8. Gurobi Optimization, Inc.: Gurobi Optimizer Reference Manual (2012), `http://www.gurobi.com`
9. IBM: ILOG Concert, part of IBM ILOG CPLEX Optimization Studio (2012), `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/interfaces/`
10. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P., Garcia de la Banda, M., Wallace, M.: The design of the Zinc modelling language. Constraints 13(3), 229–267 (2008)
11. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessiere, C. (ed.) Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 4741, pp. 529–543. Springer (2007)
12. Rendl, A.: Effective Compilation of Constraint Models. Ph.D. thesis, School of Computer Science, University of St Andrews (2010)
13. Schulte, C., et al.: Gecode, the generic constraint development environment (2009), `http://www.gecode.org/`
14. Van Hentenryck, P.: The OPL Optimization Programming Language. MIT Press (1999)
15. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. MIT Press (2005)