Compiling Conditional Constraints

Peter J. Stuckey[1,2][0000−0003−2186−0459] and Guido Tack[1,2][0000−0003−3357−6498]

[1] Monash University, Melbourne, Australia
{peter.stuckey,guido.tack}@monash.edu
[2] Data61, CSIRO, Melbourne, Australia

Abstract. Conditionals are a core concept in all programming languages. They are also a natural and powerful mechanism for expressing complex constraints in constraint modelling languages. The behaviour of conditionals is complicated by undefinedness. In this paper we show how to most effectively translate conditional constraints for underlying solvers. We show that the simple translation into implications can be improved, at least in terms of reasoning strength, for both constraint programming and mixed integer programming solvers. Unit testing shows that the new translations are more efficient, but the benefits are not so clear on full models where the interaction with other features such as learning is more complicated.

Keywords: Constraint modelling · Conditional constraints · MiniZinc

1 Introduction

Conditional expressions are a core part of virtually any programming and modelling language. They provide a way to change behaviour of the program/model depending on some test. MiniZinc 1.6 [10] and earlier versions provided the conditional expression

if *cond* **then** *thenexp* **else** *elseexp* **endif**

restricted to the case that the expression *cond* could be evaluated at compile time to *true* or *false*. This is simple to handle, since the MiniZinc compiler can replace this expression by *thenexp* if *cond* = *true* and *elseexp* if *cond* = *false*. From MiniZinc 2.0 onwards the expression *cond* is no longer restricted to be known at compile time, it can be an expression involving decision variables whose truth will be determined during the execution of the solver. This extension is very useful, it makes the expression of many complex constraints much more natural. For example, the absolute value function can be simply expressed as

```
function var int: abs(var int: x) = if x >= 0 then x else -x endif;
```

The MiniZinc compiler must translate these conditional expressions into primitive constraints that are implemented by the solver. For example, Constraint Programming (CP) solvers may use *reification* or dedicated constraints for *logical connectives* [1,7,8] to link the truth of `x >= 0` to the result of the function being `x` or `-x`, whereas for Mixed Integer Programming (MIP) solvers we would employ techniques such as *big-M* or *indicator* constraints (see e.g. [6,9]).

The effective translation of conditional expressions is the focus of this paper. But before we determine how to translate conditional expressions we must understand their *exact meaning*, including how they interact with *undefinedness*.

Undefinedness naturally arises in constraint models through the use of partial functions like `x div y` or, much more commonly, out of bounds array lookups `a[i]` where `i` takes a value outside the index set of `a`. In the remainder of the paper we will argue for the correct semantics of conditionals with undefinedness (Sect. 3), and then illustrate how to compile conditional constraints to a form that can be executed by solvers while respecting the correct semantics. We then show in Sect. 5 how we can improve this translation for constraint programming (CP) and mixed integer programming (MIP), and how we can often improve models created by non-experts, who use conditionals in a familiar procedural programming style (Sect. 6). Sect. 7 shows how the different translations compete in terms of solving efficiency, and Sect. 8 concludes the paper.

2 Model Translation

Modern modelling languages like MiniZinc [10], Essence [5], AMPL [3] and OPL [12] provide highly expressive ways of defining a constraint problem. But the underlying solvers only solve one form of problem, typically

$$\text{minimize } o \text{ subject to } \exists V. \bigwedge_{i=1}^{n} c_i$$

where V is a set of variables and each c_i is a *primitive constraint* understood by the solver. In some solvers the primitive constraints are very limited, e.g. SAT solvers only consider clauses, and MIP solvers only consider linear constraints.

The MiniZinc compiler translates high level models to models that only contain primitive constraints suitable for a given target solver. Solver-level models are represented in a language called FlatZinc, in a process called *flattening*. FlatZinc is a much richer low level language than used by SAT and MIP, in order to support all the primitive constraints natively supported by CP solvers. For this paper we will assume the solver supports the following primitives.

```
sum(i in S)(a[i]*x[i]) <= a0;           % linear inequality
sum(i in S)(a[i]*x[i]) = a0;            % linear equality
b <-> sum(i in S)(a[i]*x[i]) <= a0;     % reified linear inequality
b <-> sum(i in S)(a[i]*x[i]) = a0;      % reified linear equality
y1 = x[y2];                             % element
b1 <-> y1 = x[y2];                      % reified element
[not] b1 \/ [not] b2 \/ ... \/ [not] bn; % clause, [not] is optional
b1 <-> b2;                              % Boolean equality
```

where `x` is an integer array, `y`n is an integer, `a` is a fixed integer array, `a0` is a fixed integer, `b`n is a Boolean, and `S` is a fixed integer set. Note that we will often write these in slightly different syntactic form, e.g. `b <-> y = 0` is a reified linear equality, and `b1 /\ not b2 -> b3` is a clause.[3] Most CP solvers directly support a wide array of *global constraints* which also appear in FlatZinc. We shall introduce new global constraints as needed during the paper.

The reader may ask why we need *reified* versions of primitive constraints. This arises because in a complex model not all constraints that appear in the model must hold.

Example 1. Consider the model `constraint y <= 0 \/ x = a[y]`.
Neither `y <= 0` nor `x = a[y]` must hold all the time (one of the two is sufficient). A correct flattening making use of reified primitive constraints is

```
constraint b1 \/ b2;
constraint b1 <-> y <= 0;
constraint b2 <-> x = a[y];
```
□

In a MiniZinc model each expression occurs in a *context*, which is the nearest enclosing Boolean expression. The *root context* is all Boolean expressions that can be syntactically determined to be *true* in any solution, that is the context of a top-level constraint or a top level conjunction. A *non-root context* is any other Boolean expression. When flattening a constraint, all total functional expressions in the constraint can be moved to the root context, while relations or partial functional expressions must be reified to maintain the meaning of the model.

3 Semantics of Conditionals

Any sufficiently complex modelling language has the ability to express undefined values, which we will represent as $\perp$, for example division by zero or array index out of bounds. The treatment of undefinedness in modelling languages is complicated. Whereas in a traditional programming language it would be handled by runtime exception or abort, in a relational language this is not correct, since the solver will be making decisions that may result in undefinedness. Frisch and Stuckey [4] considered three different semantics for the treatment of undefinedness in modelling languages: a three-valued *Kleene semantics* (agreeing with a usual logical interpretation of undefinedness, but requiring three valued logic), a two-valued *strict semantics* (essentially making any undefinedness cause the model to have no solution), and the two-valued *relational semantics* (that agrees with the relational interpretation discussed above). MiniZinc and other modelling languages such as Essence Prime [11] implement the relational semantics, since it most closely accords with a modeller's intuition and does not require introducing three truth values.

The core abstract modelling language introduced in [4] did not consider conditional expressions. There are a few plausible interpretations of the desired

[3] Note that we use a simplified FlatZinc syntax, including support for reified element constraints, to improve readability.

interaction between conditionals and undefinedness. We invite the reader to consider the simple one-line constraint models given below (each sharing the same declarations of x, y and a) and determine what they believe to be the correct set of solutions for each:

```
var 0..2: x;
var 0..2: y;
array[1..2] of int: a = [0,2];
constraint if y = 0 then x = a[3] else x = a[y] endif;      %% (1)
constraint x = if y = 0 then a[3] else a[y] endif;          %% (2)
constraint x = if y = 0 then a[y] else a[1] endif;          %% (3)
constraint y = 0 \/ x = if y = 0 then a[3] else a[y] endif; %% (4)
```

The basic rule for the relational semantics is *"an undefined value causes the nearest containing Boolean expression to be false"*. That means when we find an expression with value $\perp$ we must propagate this upward through all enclosing expressions until we reach a Boolean context, where the $\perp$ becomes *false*. For instance, assume that the index set of `a` in Example 1 is `1..2`, then `a[y]` is undefined for `y=-1`. The nearest containing Boolean context is `x = a[y]`, which therefore becomes *false*. The overall disjunction would still be true, since its other disjunct `y<=0` is true. If we consider what this means for conditional expressions, we can distinguish two cases.

– An undefined result occurs somewhere in the condition `cond` or it occurs in the `thenexp` or `elseexp` and the type of `thenexp` and `elseexp` is Boolean.
– An undefined results occurs in the `thenexp` or `elseexp` and their type is not Boolean.

In the first case, the undefinedness is captured by the subexpressions and the conditional does not directly deal with undefinedness. In the second case there are two possible solutions. The *eager* approach says that the entire conditional expression takes the value $\perp$ if either `thenexp` or `elseexp` takes the value $\perp$. The *lazy* approach says that the conditional expression takes the value $\perp$ iff the `cond = true` and `thenexp =`$\perp$ or `cond = false` and `elseexp =` $\perp$. Clearly the lazy approach reduces the effect of undefinedness. We would argue that it also accords with traditional programming intuitions, since in languages such as C, `( b ? x = y / 0 : x = y;)` does not raise an exception unless `b` is true. Hence in this paper we propose to adopt the *lazy* interpretation of the relational semantics for conditional expressions. Note that this accords with the *lazy* interpretation of array lookup which is considered in the original paper on relational semantics [4]. Hence for the examples above we find:

(1) The undefined expression `a[3]` causes the `thenexp` to be *false* requiring the condition $y = 0$ to be false, leaving solutions $x = 0 \wedge y = 1$ and $x = 2 \wedge y = 2$.
(2) The undefined expression `a[3]` causes the `thenexp` to be $\perp$ requiring the condition $y = 0$ to be false, or the whole expression evaluates to *false* leaving solutions $x = 0 \wedge y = 1$ and $x = 2 \wedge y = 2$. Note how the lazy approach gives the identical answers to the "equivalent" expression (1).

(3) The `thenexp` `a[y]` is only of interest when $y = 0$ (and hence if `a[y]` is $\perp$), so the else case must hold, leaving solutions $x = 0 \wedge y = 1$ and $x = 0 \wedge y = 2$.
(4) The then case is $\perp$ so the second disjunct evaluates to *false* when $y = 0$, leaving solutions $x = 0 \wedge y = 1, x = 2 \wedge y = 2$ and $x = 0 \wedge y = 0$, $x = 1 \wedge y = 0$, $x = 2 \wedge y = 0$.

MiniZinc supports an extended conditional expression of the form

```
if c1 then e1 elseif c2 then ... else ek endif
```

This is semantically equivalent to

```
if c1 then e1 else if c2 then ... else ek endif ... endif
```

To simplify presentation, we will assume an alternative syntax that uses two arrays of size k: `ite([c1,c2,...,true],[e1,e2,...,ek])` The `c` array are the *conditions*, the `e` array are the *results*. Note how the last condition $c_k = true$ for the else case. The (lazy) relational semantics applied to this expression requires that the conditional takes the value of expression e_i iff $c_i \wedge \bigwedge_{j=1}^{i-1} \neg c_j$ holds. Hence if e_i is $\perp$ then either the i^{th} condition cannot hold or the nearest enclosing Boolean context will be *false*. This is the same semantics as if we treated the `ite` as a nested sequence of `if then else endif` expressions.

4 Translating Conditionals

In this section we examine how to translate the conditional expression `ite(c,e)` where `c` is an array of k Boolean expressions with the last one being *true*, and `e` is an array of k expressions. We usually introduce a new variable `x` to hold the value of this expression, `x = ite(c,e)`.

Boolean Result In its most basic form, a conditional is a Boolean expression, i.e., `x` has type `par bool` or `var bool`. In this case the semantics can be defined in terms of simple expressions:[4]

```
x = forall (i in 1..k) (c[i] /\ not exists (c[1..i-1]) -> e[i])
```

If `x` is *true* (e.g. if the conditional appears directly as a constraint) this can be encoded using k simple clauses. Because Boolean values can never be $\perp$ (they simply become *false*) there is no difficulty with undefinedness here.

Non-Boolean Result If the `e[i]` are not Boolean, a straightforward translation scheme would result in the Boolean expression:

```
forall (i in 1..k) (c[i] /\ not exists (c[1..i-1]) -> x = e[i])
```

If none of the expressions `e[i]` can be $\perp$ this is correct. But if case i is the selected case and `e[i]` is $\perp$, then this translation causes the entire model to have no solution. If the conditional expression occurs at the root context this is correct, but if it occurs in a non-root context, the desired semantics is that it enforces that the context is *false*.

[4] We use array slicing notation `c[l..u]` equivalent to `[ c[j] | j in l..u ]`.

Example 2. Consider the translation of example (3). Since the expression appears in the root context we can use the simple translation:

```
constraint b1 <-> y = 0;                      % Boolean for condition y = 0
constraint b2 <-> x = a[y];                   % then expression
constraint b3 <-> x = a[1];                   % else expression
constraint b1 -> b2;                          % then case
constraint true /\ not b1 -> b3;              % else case
```

The solutions, projecting onto x and y, are $x = 0 \wedge y = 1$ and $x = 0 \wedge y = 2$ as expected. Note the undefinedness of the expression `a[y]` is captured by the reified constraint `b2 <-> x = a[y]`. □

In order to deal with non-root contexts we introduce an explicit representation of undefinedness by associating with every term `t` appearing in a non-root context the Boolean `def(t)` which records whether it is defined (*true*) or undefined, i.e. `def(t)` = *false* iff `t` = ⊥. The translation will ensure that if `def(t)` = *true* then `t` takes its appropriate defined value, and if `def(t)` = *false* then `t` is free to take some value (which does not affect the satisfiability of other constraints). We can now define the full semantics of a conditional `ite(c,e)`:

```
forall(i in 1..k)( def(x)    /\ c[i] /\ not exists(c[1..i-1]) -> x=e[i] )
forall(i in 1..k)( def(e[i]) /\ c[i] /\ not exists(c[1..i-1]) -> def(x) )
```

Let us look at the two parts of this definition in turn. The first part (line 1) states that if the overall expression is defined (`def(x)`), and alternative i is selected, then the expression takes the value of branch i. Due to the relational semantics of `x=e[i]`, this also implies that `e[i]` is defined. The second part (line 2) states that if the value of the selected branch is defined, then the value of the overall conditional is defined. Note how if `e[i]` is ⊥ and the i^{th} case is chosen, then the first part will enforce `def(x)` to be *false*, and there will be effectively no usage of the value of `e[i]`.

Example 3. Let us consider the translation of example (4). Here the conditional expression is not in a root context. The resulting translated form is

```
constraint b1 \/ b2;                          % top level disjunction
constraint b1 <-> y = 0;                      % Boolean for condition y = 0
constraint b2 <-> def(x);       % second disjunct holds iff x is defined
constraint b3 <-> x = a[3];                   % then expression
constraint b4 <-> x = a[y];                   % else expression
constraint def(x) /\ b1 -> b3;                % then case
constraint def(a[3]) /\ b1 -> def(x);
constraint def(x) /\ true /\ not b1 -> b4; % else case
constraint def(a[y]) /\ true /\ not b1 -> def(x);
constraint def(a[3]) <-> false;               % definedness of base
constraint def(a[y]) <-> y in 1..2;
```

The solutions, projecting onto x and y, are $x = 0 \wedge y = 1$, $x = 2 \wedge y = 2$, $x = 0 \wedge y = 0$, $x = 1 \wedge y = 0$, and $x = 2 \wedge y = 0$ as expected. □

5 Improving the Translation of Conditionals

The direct translation of conditionals explored in the previous section is correct, but does not necessarily propagate very well. We can create better translations if we know more about the underlying solver we will be using.

5.1 Element translation

We first restrict ourselves to the case that none of the expressions `e` can be $\perp$. In this case we can translate `x = ite(c,e)` as `x = e[ arg_max(c) ]`. Note that the `arg_max` function returns a variable constrained to be the least index which takes the maximum value. Since `c[k]` is always *true* this returns the first condition which is *true*. Thus this expression first calculates the first case j whose condition holds, and then uses an *element constraint* to equate `x` to the appropriate expression in `e`. Flattening simply yields `x = e[j] /\ maximum_arg(c, j)` where `maximum_arg` is the predicate version of the `arg_max` function. Note in the case of the simple conditional `if b then e[1] else e[2] endif` we can avoid the use of `arg_max` and simply use `x = e [2 - b]`. If `b` is *true* we obtain `x = e[1]` and otherwise `x = e[2]`.

The advantage of the element translation is that we can get stronger propagation. A solver that has native support for the element constraint can apply a form of *constructive disjunction* [13]: it can reason about all of the cases together, and propagate any common information that all cases (disjuncts) share.

Example 4. Consider the translation of the expression

```
x = if c1 then a elseif c2 then b else c endif
```

where variables $a, b, c \in \{5, 8, 11\}$. The implication form compiles to

```
constraint b1 <-> x = a;
constraint c1 -> b1;
constraint b2 <-> x = b;
constraint not c1 /\ c2 -> b2;
constraint b3 <-> x = c;
constraint not c1 /\ not c2 -> b3;
```

Initially no propagation is possible. But the element translation is

```
constraint maximum_arg([b1,b2,true],j);
constraint x = [a,b,c][j];
```

Immediately propagation determines that `x` can only take values $\{5, 8, 11\}$. If some other constraints then cause the value 5 to be removed from the domains of a, b and c, the element constraint will also remove it from x, while the implication form will not be able to perform this kind of reasoning. □

We can show that the element translation is domain consistent when using domain consistent propagators for `element` and `maximum_arg`. Hence it is a strongest possible decomposition, and therefore must propagate at least as much as the implication decomposition.

Theorem 1. *The element translation of `ite` is domain consistent assuming domain propagation of the primitives.* □

In order to take into account definedness we can extend the translation by using another element constraint to determine the definedness of the result:

```
x = e [ arg_max(c) ] /\
def(x) = [ def(e[i]) | i in index_set(e) ] [ arg_max(c) ];
```

Suppose `e[i]` is undefined, then since `def(e[i])` is *false* the resulting variable will be unconstrained, which will leave `x` unconstrained as well.

5.2 Domain consistent decomposition of `maximum_arg`

Note that the element translation relies on domain consistent propagation for the element and `maximum_arg` constraints to enforce domain consistency. Many CP solvers support domain propagation for element, but not necessarily for `maximum_arg`. We can enhance the standard decomposition of `maximum_arg` so that it enforces domain consistency.

The new decomposition is shown below. It tests for the special case that one of the elements in the array is known to be at the upper bound of all elements at compile time, i.e `exists(j in l..u)(lb([x[j]) = ubx)`. The decomposition introduces variables `d` such that `d[i]` is true iff `x[k]=ubx` for some `k<=i`.

```
predicate maximum_arg(array[int] of var int: x, var int: y) =
    let { int: l = min(index_set(x));
          int: u = max(index_set(x));
          int: ubx = ub_array(x);
        } in
    if exists(j in l..u)(lb(x[j]) = ubx)       % max is known to be ubx
    then let { array[l..u] of var bool: d; } in
         x[y] = ubx /\                        % ith case must be equal to ub
         forall(j in l..u)(x[j] = ubx -> y <= j) /\ % lower bound
         d[l] = (x[l] = ubx) /\
         forall(j in l+1..u)(d[j] <-> (d[j-1] \/ (x[j] = ubx))) /\
         forall(j in l..u)(not d[j] -> y >= j+1)    % upper bound
    else % otherwise use standard integer decomposition
         maximum_arg_int(x, y)
    endif;
```

Theorem 2. *The decomposition of `maximum_arg` above maintains domain consistency in the case where the maximum is known.* □

5.3 Domain consistent propagator for `maximum_arg`

Given the key role that `maximum_arg` plays for translating conditionals it may be worth building a specialised propagator for the case where it applies: on an array of Booleans where the maximum is known to be *true*. The following pseudocode implements a domain consistent propagator for this case. In a learning solver each propagation must be explained. The correct explanations are given in the parentheses following the *because* statements.

```
index_of_first_true($x, y$)
    $ol := lb(y)$
    **for** $(j \in 1..ol - 1)$
        propagate $\neg x[j]$ *because* $(y \geq ol \rightarrow \neg x[j])$
    $l := ub(y) + 1$
    $u := lb(y)$
    **for** $(j \in dom(y))$
        **if** $(l > ub(y) \wedge ub(x[j]) = true)$ $l := j$
        **if** $(ub(x[j]) = false)$ propagate $y \neq j$ *because* $(\neg x[j] \rightarrow y \neq j)$
        **if** $(lb(x[j]) = true)$ $u := j$ ; **break**
    propagate $y \geq l$ *because* $(y \geq ol \wedge \bigwedge_{k \in ol..l-1} y \neq k \rightarrow y \geq l)$
    propagate $y \leq u$ *because* $(x[u] \rightarrow y \leq u)$
    **if** $(lb(y) = ub(y))$ propagate $x[lb(y)]$ *because* $(y = lb(y) \rightarrow x[lb(y)])$
    **for** $(j \in ol..lb(y) - 1)$
        propagate $\neg x[j]$ *because* $(y \geq lb(y) \rightarrow \neg x[j])$
```

The propagation algorithm above first ensures that all the index positions lower than the lower bound of y cannot be true. Note that propagation does nothing if this is already the case. The propagator iterates through the remaining possible values of y. It finds the first element indexed by y that might be true and records this as l. If it finds index positions j where $x[j]$ is false, it propagates that y cannot take these values. If it finds an index position j where $x[j]$ is true, it records the position as u and breaks the loop. Once the loop is exited it sets the new lower bound as l and the new upper bound as u. If the *updated* lower bound and upper bound of y are equal we propagate that x at that position must be true. Finally we ensure that all x positions less than the new lower bound of y are false. Note that if $l = ub(y) + 1$ at the end of the loop this effectively propagates failure, similarly if $lb(y) = ub(y)$ and $x[lb(y)]$ is known to be false, or $x[j], j < lb(y)$ is known to be true.

Theorem 3. *The index_of_first_true(x, y) propagator enforces domain consistency for maximum_arg(x, y) when the max is known to be true.* □

5.4 Linear translation

When the target solver to be used for conditional constraints is a MIP solver, the conditional constraints have to be linearised. The implication form of a conditional constraint is in fact not too difficult to linearise, however it may result in a weak linear relaxation. We now discuss how to improve over the default linearisation of Boolean constraints in MiniZinc.

Example 5. Consider the linearisation of the implication form of the expression:

```
x = if c1 then 5 elseif c2 then 8 else 11 endif
```

Assuming the initial bounds on x are 0..100, the result is (see [2] for details):

```
constraint x >= 5*b1;                  % b1 -> x >= 5
constraint x <= 5 + 95*(1-b1);         % b1 -> x <= 5
constraint x >= 8*b2;                  % b2 -> x >= 8
constraint x <= 8 + 92*(1-b2);         % b2 -> x <= 8
constraint x >= 11*b3;                 % b3 -> x >= 11
constraint x <= 11 + 89*(1-b3);        % b3 -> x <= 11
constraint b1 >= c1;                   % c1 -> b1
constraint c1 + b2 >= c2;              % not c1 /\ c2 -> b2
constraint c1 + c2 + b3 >= 1;          % .. -> b3
```

This has a weak linear relaxation, x can take any value in the range 0..100. □

We can do better, at least in the case where all the expressions e are fixed, by translating `x = ite(c,e)` as follows:

```
int: k = length(c);
array[1..k] of var 0..1: b;
constraint forall(i in 1..k) (sum(c[1..i-1]) + b[i] >= c[i]);
constraint 1 = sum(b);      % redundant: exactly one case is true
constraint x = sum(i in 1..k)(b[i]*e[i]);
```

The case selector variables `b` are the same as in the implication form, but we add that exactly one of them is 1, and generate `x` as a linear combination of the cases. If all elements of the array `e` are fixed integer expressions then the last constraint is simply a linear equation. In the case of a conditional with just one condition we again have a much simpler translation: `x = b*e[1] + (1-b)*e[2];`

Example 6. The result of translating the expression of Example 5 is

```
constraint b1 >= c1;                     % c1 -> b1
constraint c1 + b2 >= c2;                % not c1 /\ c2 -> b2
constraint c1 + c2 + b3 >= 1;            % .. -> b3
constraint b1 + b2 + b3 = 1;             % exactly one case
constraint x = b1 * 5 + b2 * 8 + b3 * 11; % definition of x
```

Bounds propagation can immediately determine from the last equation that `x` takes values in the range 0..24. A linear solver can immediately determine (using the last two constraints) that `x` takes values in the range 5..11. □

The example above shows that the linear translation can sometimes be much stronger than the linearisation of the implication translation. The following theorem establishes that it is indeed never weaker.

Theorem 4. *Assuming all values in* `e` *are constant, the linear relaxation of the linear translation of a conditional expression is no weaker than the linear relaxation of the implication translation.* □

Since the linear translation is not worthwhile unless all the `e` expressions are fixed we do not need to extend it to handle undefinedness.

6 Improving "Procedural" Code

Many non-expert users of MiniZinc make heavy use of the conditional statement, since they are familiar with it from procedural programming. This section shows that reformulating the procedural style can lead to improved propagation when the element translation is used.

Example 7. Consider the constraint
`if c1 then x = 5 elseif c2 then x = 8 else x = 11 endif;`
The resulting element translation is

```
constraint b1 <-> x = 5;
constraint b2 <-> x = 8;
constraint b3 <-> x = 11;
constraint maximum_arg([c1,c2,true],j);
constraint true = [b1,b2,b3][j];
```

Initially a CP solver will be able to propagate no information. But consider the equivalent expression

```
constraint x = if c1 then 5 elseif c2 then 8 else 11 endif;
```

discussed in Example 5. Using the element translation the solver will immediately propagate that the domain of `x` is $\{5, 8, 11\}$.

The linear translation of the original form above is the same as shown in Example 5. The linear relaxation of this system does not constrain `x` more than its original bounds 0..100. The linear relaxation of the translation of the equivalent expression discussed in Example 6 enforces that `x` is in the range 5..11. □

We can define a transformation for arbitrary conditional expressions that share at least some equational constraints for the same variable. This is a very common modelling pattern for inexperienced modellers. Consider the constraint

```
if x = 0 then y = 0 /\ z = 0
elseif x <= l then y = l /\ z = x
elseif x <= u then y = x /\ z = u
else x >= y + z endif
```

In most cases the conditional (conjunctively) defines values for `y` and `z`. To compile this efficiently we can in effect duplicate the conditional structure to give separate partial definitions of `y`, `z` and any leftover constraints.

```
y = if x=0 then 0 elseif x<=l then l elseif x<=u then x else y endif /\
z = if x=0 then 0 elseif x<=l then x elseif x<=u then u else z endif /\
if x = 0 then true elseif x<=l then true elseif x<=u then true
else x >= y + z endif
```

When translating the result using the element translation we can reuse the `maximum_arg` computation for each part, arriving at

```
var 1..4: j;
constraint maximum_arg([x = 0, x <= l, x <= u, true], j);
constraint y = [0,l,x,y][j];
constraint z = [0,x,u,z][j];
constraint true = [true,true,true,x >= y+z][j];
```

We can show that lifting the equalities out of an if-then-else expression can only improve propagation. The theorem assumes that each branch has an equality for variable `x`, if this is not the case we can add the equation `x = x` to give the right syntactic form.

Theorem 5. *Using the element translation for a conditional constraint of the form* $\mathtt{ite}(c, [x = ex_1 \wedge r_1, x = ex_2 \wedge r_2, \ldots x = ex_m \wedge r_m])$ *and the equivalent constraint* $x = \mathtt{ite}(c, [ex_1, ex_2, ..., ex_m]) \wedge \mathtt{ite}(c, [r_1, r_2, \ldots, r_m]$*, then the second form propagates at least as much.* □

7 Experimental Evaluation

This section presents an evaluation of the different approaches to compiling conditionals using an artificial unit test and a more natural model of a problem that uses conditionals.

All experiments were run on an Intel Core i7 processor at 4 GHz with 16 GB of RAM. The models were compiled with current development versions of MiniZinc, Gecode and Chuffed[5], CBC version 2.9.8/1.16.10 and CPLEX version 12.8.0. The solvers were run in single-core mode with a time out of 120 seconds. All models use fixed search for CP solvers so that the differences for traditional CP solvers only arise from differences in propagation, in learning solvers differences also arise from different learning. For the MIP solvers CBC and CPLEX, presolving was switched off to improve consistency of results.

7.1 Unit Tests

Let π be a sorted sequence of $4n$ distinct integers in the range $0 \ldots 100n$, with $\pi(i)$ denoting the ith integer in the sequence. We build a (procedural) constraint system consisting of two `ite` constraints[6]

$$\mathtt{ite}([x \leq \pi(4i-3) | i \in 1..n-1]\mathtt{++}[\mathtt{true}], [y = \pi(4i-2) | i \in 1..n])$$
$$\mathtt{ite}([y \leq \pi(4i-1) | i \in 1..n-1]\mathtt{++}[\mathtt{true}], [x = \pi(4i) | i \in 1..n-1]\mathtt{++}[\mathtt{0}])$$

By definition the first constraint implies $y > x$ unless $x > \pi(4n-7)$ then $y = \pi(4n-2)$. The second constraint implies $x > y$ unless $y > \pi(4n-5)$ then $x = 0$. Together these are unsatisfiable. We also consider the equivalent functional form

$$y = \mathtt{ite}([x \leq \pi(4i-3) | i \in 1..n-1]\mathtt{++}[\mathtt{true}], [\pi(4i-2) | i \in 1..n])$$
$$x = \mathtt{ite}([y \leq \pi(4i-1) | i \in 1..n-1]\mathtt{++}[\mathtt{true}], [\pi(4i) | i \in 1..n-1]\mathtt{++}[\mathtt{0}])$$

[5] MiniZinc revision e1d9d10, Gecode revision b3fceb0, Chuffed revision b74a2d7
[6] The `++` operator stands for array concatenation.

Table 1: Unit testing for the Gecode and Chuffed CP solvers.

	Gecode					Chuffed				
n	$\rightarrow$	$[]$	$[]_P$	$x = []$	$x = []_P$	$\rightarrow$	$[]$	$[]_P$	$x = []$	$x = []_P$
25	0.07	0.06	**0.05**	0.06	0.06	0.07	0.07	0.06	**0.05**	**0.05**
50	0.06	0.07	**0.05**	0.06	**0.05**	0.08	0.08	0.08	0.06	**0.05**
100	0.21	0.15	0.22	0.24	**0.05**	0.10	0.10	0.07	0.17	**0.05**
200	0.15	0.15	0.07	0.13	**0.05**	0.17	0.17	0.08	0.14	**0.05**
400	0.38	0.33	0.09	0.30	**0.06**	0.42	0.36	0.11	0.31	**0.07**
800	1.43	1.12	0.21	1.03	**0.08**	1.43	0.98	0.17	1.03	**0.09**
1600	5.11	3.08	0.24	3.24	**0.12**	5.40	3.68	0.27	3.39	**0.15**
3200	26.23	12.92	0.44	12.78	**0.19**	26.29	13.53	0.53	13.70	**0.23**
6400	—	67.39	3.78	72.43	**0.39**	—	60.54	1.97	60.64	**0.52**

Table 2: Unit testing for the CBC and CPLEX MIP solvers.

	CBC			CPLEX		
n	$\rightarrow$	$x = []$	$\sum$	$\rightarrow$	$x = []$	$\sum$
25	0.21	1.84	**0.06**	1.41	0.19	**0.05**
50	0.27	1.68	**0.08**	0.22	0.37	**0.07**
100	0.95	13.08	**0.17**	0.71	1.16	**0.12**
200	4.69	48.92	**0.38**	3.90	4.52	**0.30**
400	40.99	—	**1.27**	38.62	17.12	**1.01**
800	—	—	**5.34**	—	71.97	**3.94**
1600	—	—	**20.51**	—	—	**18.76**
3200	—	—	**94.75**	—	—	**70.37**

Table 1 compares the Gecode and Chuffed CP solvers on five different versions: $\rightarrow$ is the implication translation (which is identical for both forms); $[]$ is the element translation of the procedural form with domain consistent `maximum_arg` decomposition; $[]_P$ is the element translation of the procedural form with domain consistent `maximum_arg` propagator; $x = []$ is the element translation of the functional form with domain consistent `maximum_arg` decomposition; and $x = []_P$ is the element translation of the functional form with domain consistent `maximum_arg` propagator; A — indicates 120s timeout reached.

The results demonstrate the clear benefits of the mapping from procedural form to functional form, and the benefit of the element translation, in particular with a global propagator for `maximum_arg` over the implication form.

Table 2 shows the results of the MIP solvers of the functional form of the model with: $\rightarrow$ linearisation of the implication translation; $x = []$ linearisation of the element translation with decomposed `maximum_arg`; and $\sum$ the direct linearisation defined in Section 5.4. Note here we cannot control the search, so that there may be more variance in results due to other factors.

The linearisation of the implication translation is generally better than the linearisation of the element translation, since the `maximum_arg` decomposition makes the whole thing much more complex. The direct linear translation is substantially better and much more scalable.

7.2 Fox-Geese-Corn

To study the effects of different translations on a more realistic model, we consider a generalisation of the classic *Fox-Geese-Corn* puzzle, where a farmer needs to transport goods from one side of a river to the other side in several trips. It contains several rules such as "When the farmer leaves some foxes and geese alone on one side of the river, and there are more foxes than geese, one fox dies in argument over geese, and no geese die; if there are no more foxes than geese, each fox eats a goose." The overall goal is to maximise the farmer's profit of goods successfully transported across.[7]

The most natural way of modelling this problem follows the structure of the specification, using conditionals to constrain state variables for each time point and type of object. We manually analysed models written by students and submitted as part of an online assignment, and we have identified a number of different modelling approaches based on conditional expressions. Almost all students use the "procedural" syntax. In general, the constraints take the form such as `if cond then geese[i+1]=geese[i]-c elseif ... endif`, where `geese[i]` is the number of geese at step `i`, `cond` specifies the condition from the rules, and `c` is a constant (e.g. `1` in the case that one goose dies) or a variable (e.g. `fox[i]` in the case that each fox eats one goose). We manually performed the conversion into expression form such as `geese[i+1] = if cond then geese[i]-c elseif ... endif` and call this Model 1.

Several students described all cases explicitly in the conditions, not noticing that the case distinction is in fact exhaustive. For example, they might have written `if cond1 then geese[i+1]=geese[i]-c elseif not cond then geese[i+1]=geese[i] else true endif`, instead of the simpler `if cond then geese[i+1]=geese[i]-c else geese[i+1]=geese[i] endif`. The resulting expression form, which we call Model 2, would contain the less compact `geese[i+1]=if cond then geese[i]-c elseif not cond then geese[i] else geese[i+1] endif`.

Table 3 shows the results of running two different models on ten instances, with the implication ($\rightarrow$) and element ($x = []_P$) encoding, and the additional linear encoding ($\sum$) for CPLEX. Model 1 is the one described above. Model 2 is a variation of Model 1,

A number without superscript represents time in seconds for finding the optimal solution and proving optimality. For cases where search timed out, a number with superscript (e.g. $\mathit{1.75}^{120}$) represents the time when the best solution was found (1.75 seconds) and the objective value of the best solution found (120). The last line in each table summarises the results for each solver: for a given solver, an encoding wins for an instance if it proves optimality faster than the other encoding(s), or if neither encoding proves optimality, if it finds a better solution, or the same objective but faster. A — represents that no solution was found within the 120 second timeout. Numbers in bold indicate the winning model (highest objective found in least amount of time) for each solver.

[7] A model for this problem can be found in the MiniZinc benchmarks repository at https://github.com/minizinc/minizinc-benchmarks.

Table 3: Fox-Geese-Corn
Model 1 (exhaustive conditional):

	Gecode		Chuffed		CPLEX		
	$\to$	$x = []_P$	$\to$	$x = []_P$	$\to$	$x = []$	$\sum$
1	**0.16**	**0.16**	0.10	**0.06**	1.86	1.74	**0.22**
2	0.53	**0.42**	**0.12**	0.13	0.92	0.76	**0.67**
3	*1.75*^{120}	***1.39***^{120}	***7.18***^{148}	*8.70*^{148}	*95.46*^{112}	***50.57***^{141}	*11.92*^{132}
4	0.79	**0.64**	0.34	**0.25**	0.45	**0.28**	0.29
5	*113.53*^{38}	***102.16***^{40}	*97.97*^{52}	***86.38***^{52}	*48.01*^{40}	***105.14***^{46}	*86.35*^{44}
6	*116.91*^{64}	***91.89***^{64}	***90.29***^{71}	*103.45*^{71}	***106.70***^{71}	*118.12*^{71}	*19.22*^{69}
7	72.36	**61.35**	**1.50**	1.55	6.63	**1.73**	2.46
8	88.34	**75.29**	2.43	**2.27**	10.61	**4.13**	6.16
9	*24.61*^{72}	***19.39***^{72}	*84.65*^{130}	***43.01***^{130}	*93.49*^{116}	**57.06**	*58.58*^{128}
10	*0.52*^{140}	***0.40***^{140}	*81.76*^{280}	***77.41***^{280}	*36.82*^{219}	***116.24***^{430}	*115.81*^{391}
Wins	0	9	4	6	1	7	2

Model 2 (additional `else true` case):

	Gecode		Chuffed		CPLEX		
	$\to$	$x = []_P$	$\to$	$x = []_P$	$\to$	$x = []$	$\sum$
1	0.16	**0.06**	0.10	**0.07**	1.76	0.41	**0.28**
2	0.13	**0.11**	**0.10**	**0.10**	0.90	**0.79**	0.96
3	***0.15***^{75}	***0.15***^{75}	***3.16***^{148}	*3.54*^{148}	*45.76*^{142}	***15.16***^{148}	*82.25*^{143}
4	0.14	**0.12**	**0.11**	0.12	0.53	0.60	**0.33**
5	*0.21*^{14}	***0.16***^{14}	***39.62***^{52}	*57.61*^{52}	*109.03*^{32}	***118.09***^{47}	*23.22*^{24}
6	*0.13*^{21}	***0.10***^{21}	***48.06***^{71}	*49.84*^{71}	*12.65*^{67}	*111.29*^{65}	***14.13***^{70}
7	3.09	**2.56**	**0.29**	0.32	**2.55**	2.77	2.69
8	1.90	**1.47**	0.24	**0.20**	**1.30**	1.37	1.33
9	*0.15*^{29}	***0.12***^{29}	*55.69*^{130}	***54.68***^{130}	*64.47*^{130}	*51.36*^{132}	***36.42***^{132}
10	*0.34*^{112}	***0.25***^{112}	***21.46***^{350}	*22.62*^{350}	—	*51.92*^{419}	***75.52***^{525}
Wins	0	9	6	3	2	3	5

In more complex models the relative performances of the different encodings are less distinct. For Gecode the $x = []_P$ version continues to be superior, for Chuffed the $\to$ version is more competitive because it can learn on intermediate literals. Surprisingly for CPLEX the $x = []$ translation is usually better than the implication translation and overall best for Model 1. For Model 2 the Σ translation is strongest even though in this case the e arguments are not fixed.

8 Conclusion

Conditional expressions are one of the most expressive constructs that appear in modelling languages, allowing natural expression of complex constraints. They are also frequently used by beginner modellers used to thinking procedurally. In this paper we show how to translate conditional expressions to a form that underlying solvers can handle efficiently. Unit tests clearly demonstrate the improvements from new translations, though this is not so clear on more complex models, where other factors interact with the translations. The concepts presented here are available in MiniZinc 2.3.0, Gecode 6.2.1, and Chuffed 0.10.5.

Acknowledgements We would like to thank the anonymous reviewers for their comments that helped improve this paper. This work was partly sponsored by the Australian Research Council grant DP180100151.

References

1. Bacchus, F., Walsh, T.: Propagating logical combinations of constraints. In: Kaelbling, L.P., Saffiotti, A. (eds.) IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005. pp. 35–40 (2005)
2. Belov, G., Stuckey, P.J., Tack, G., Wallace, M.: Improved linearization of constraint programming models. In: Rueher, M. (ed.) Proceedings of the 22st International Conference on Principles and Practice of Constraint Programming. pp. 49–65. No. 9892 in LNCS, Springer (2016)
3. Fourer, R., Kernighan, B.: AMPL: A Modeling Language for Mathematical Programming. Duxbury (2002)
4. 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-Verlag (2009)
5. 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)
6. Hooker, J.N.: Integrated Methods for Optimization. Springer, 2nd edn. (2011)
7. Jefferson, C., Moore, N.C.A., Nightingale, P., Petrie, K.E.: Implementing logical connectives in constraint programming. Artif. Intell. **174**(16-17), 1407–1429 (2010). https://doi.org/10.1016/j.artint.2010.07.001
8. Lhomme, O.: Arc-consistency filtering algorithms for logical combinations of constraints. In: Régin, J., Rueher, M. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3011, pp. 209–224. Springer (2004). https://doi.org/10.1007/978-3-540-24664-0_15
9. McKinnon, K.I.M., Williams, H.P.: Constructing integer programming models by the predicate calculus. Annals of Operations Research **21**(1), 227–245 (Dec 1989). https://doi.org/10.1007/BF02022101
10. 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-Verlag (2007)
11. Nightingale, P.: Savile Row, a constraint modelling assistant (2018), http://savilerow.cs.st-andrews.ac.uk/
12. Van Hentenrcyk, P.: The OPL Optimization Programming Language. MIT Press (1999)

13. Van Hentenryck, P., Saraswat, V., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(FD). In: Podelski, A. (ed.) Constraint Programming: Basics and Trends. pp. 293–316. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)