Modeling with Metaconstraints and Semantic Typing of Variables

Andre A. Cire  
Department of Management, University of Toronto Scarborough, acire@utsc.utoronto.ca

John N. Hooker  
Tepper School of Business, Carnegie Mellon University, jh38@andrew.cmu.edu

Tallys Yunes  
School of Business Administration, University of Miami, tallys@miami.edu

Recent research in hybrid optimization shows that a combination of technologies that exploits their complementary strengths can significantly speed up computation. The use of high-level metaconstraints in the problem formulation can achieve a substantial share of these computational gains by better communicating problem structure to the solver. During the solution process, however, metaconstraints give rise to reformulations or relaxations that introduce auxiliary variables, and some of the variables in one metaconstraint’s reformulation may be functionally the same as or related to variables in another metaconstraint’s reformulation. These relationships must be recognized to obtain a tight overall relaxation. We propose a modeling scheme based on semantic typing that systematically addresses this problem while providing simpler, self-documenting models. It organizes the model around predicates and declares variables by associating each with a predicate through a keyword that is analogous to a database query. We present a series of examples to illustrate this idea over a wide variety of applications.

Key words: modeling; hybrid methods; metaconstraints; semantics

1. Introduction

Recent research in the area of hybrid optimization shows that the right combination of different technologies can simplify modeling and speed up computation substantially, over a wide range of problem classes (surveyed in Hooker 2012). These gains come from the complementary strengths of the techniques being combined, such as mathematical programming, constraint programming, local search, and propositional satisfiability. Search, inference, and relaxation lie at the heart of these techniques, and can be adjusted to exploit the structure of a given problem. Exploiting structure, as a matter of fact, is a key ingredient for successfully solving challenging optimization problems. The more structure the user can communicate to the solver, the more it can take advantage of specialized inference and relaxation techniques. A richer modeling environment, with an extended set
of constraint types, not only enables the expression of complex structures, but also results in simpler models and that require less development and debugging time.

Highly structured subsets of constraints, as well as simpler constraints, can be written as metaconstraints, which are similar to global constraints in constraint programming. Syntactically, a metaconstraint is written much as linear or global constraints are written, but it is accompanied by parameters that specify how the constraint behaves during the solution process. For example, a metaconstraint can specify how it is to be relaxed, how it will filter domains, and how the search procedure will branch in case it becomes violated in the current problem relaxation. For example, let \( x \in \mathbb{R}^n \) and consider a constraint given by the following disjunction of two inequalities:

\[
(a^1 x \leq b_1) \lor (a^2 x \leq b_2)
\]

where a binary variable \( y \) controls which disjunct is enforced (the first if \( y = 1 \), the second if \( y = 0 \)). Assume the user wants the convex hull relaxation of this constraint to be automatically added to the model’s overall linear relaxation. In a modeling language supporting metaconstraints, the syntax to represent this disjunction and its treatment by the solver might be

\[
\text{disj}(y, a^1 x \leq b_1, a^2 x \leq b_2) : \text{relax} = \text{convhull};
\]

where the \text{relax} keyword specifies the type of relaxation (there could also be a \text{branch} keyword to specify a way to branch on the disjunction).

When such parameters are omitted, a pre-specified default behavior is used. The relaxation, inference, and branching techniques are devised for each constraint’s particular structure. For example, a metaconstraint may be associated with a tight polyhedral relaxation from the integer programming literature and/or an effective domain filter from constraint programming. Because metaconstraints can also control the search, if a branching method is explicitly indicated, the search will branch accordingly. Recent versions of existing modeling languages and systems already provide some support for metaconstraints as described above (see Section 5 for specific examples).

Although metaconstraint-based modeling offers several advantages, it raises a fundamental issue of variable management that must be addressed before its full potential can be realized. As the solver relaxes and/or reformulates metaconstraints, it often creates auxiliary variables. Variables created for different constraints may actually have the same
meaning, or they may relate in some more complicated way to each other and to variables in the original model. The solver must recognize these relationships among variables if it is to produce a tight overall relaxation of the problem. Going back to the disjunctive constraint example above, the automatic convex hull relaxation of (1) would create auxiliary copies of $x$, say $x^1$ and $x^2$, and use them to write
\begin{align*}
a^1 x^1 & \leq b_1 y, \quad a^2 x^2 \leq b_2 (1 - y) \\
x & = x^1 + x^2, \quad 0 \leq y \leq 1
\end{align*}

Imagine, however, what would happen if this model contained another disjunction on $x$ that is controlled by the same $y$ variable, such as
\begin{align*}
(c^1 x \leq d_1) \lor (c^2 x \leq d_2)
\end{align*}

To write the relaxation of (2), copies of $x$ would have to be introduced, just as they were for the relaxation of (1). But would you want the former copies of $x$ to be related or unrelated to the latter copies of $x$? (Answer: they should not only be related; they should be the same.) The high-level modeler should not have to worry about this, and other kinds of, low-level bookkeeping; the modeling language/system should do that automatically.

The primary purpose of this paper to address this problem with a semantic typing scheme. We view a model as organized around user-defined multi-place predicates, whose terms include one or more variables. A variable is declared by specifying a predicate with which it is associated, which creates a semantic type for the variable. The user assigns types to variables that are originally in the model, and the solver assigns types to auxiliary variables it generates while processing metaconstraints. Relationships between variables are then deduced from their semantic types.

In Section 2 we describe several frequently used relaxations and reformulations that produce auxiliary variables. A complete example motivating the need for semantic typing is included in Section 3. We formalize the relationship between semantic types and predicates in Section 4, and review related work in Section 5. We then illustrate the use of semantic typing on a wide range of situations in Section 6. Section 7 generalizes some of the relationships between variables discussed earlier, and Section 8 concludes the paper.
2. Sources of Auxiliary Variables

Relaxation and reformulation are key elements of optimization methods (Hooker 2005, 2012), and both can introduce auxiliary variables. Some examples follow:

- A general integer variable \( x_i \) can be reformulated as a collection of new binary variables \( y_{ij} \) for each value of \( j \) in the domain of \( x_i \), so that \( x_i = \sum_j j y_{ij} \). The \( y_{ij} \)'s may be equivalent to variables that occur in the model or relaxations of other constraints.

- Disjunctions of linear systems such as \( \bigcup_{k \in K} A^k x \geq b^k \) can be given a convex hull relaxation:
  \[
  A^k x \geq b^k y_k, \text{ for all } k \in K
  \]
  \[
  x = \sum_{k \in K} x^k, \quad \sum_{k \in K} y_k = 1
  \]
  \[
  y_k \geq 0, \text{ for all } k \in K
  \]
  Note the introduction of the new variables \( x^k \) and \( y_k \). Disjunctions of non-linear systems are handled in a similar way. Frequently, different constraints are based on the same set of alternatives (e.g., configurations of a factory), and the corresponding auxiliary variables should be identified.

- Disjunctions can also be given big-\( M \) relaxations, which introduce binary variables but no new continuous variables:
  \[
  A^k x \geq b^k - (1 - y_k) M^k, \text{ for all } k \in K
  \]
  \[
  \sum_{k \in K} y_k = 1, \quad L \leq x \leq U
  \]
  \[
  y_k \geq 0, \text{ for all } k \in K
  \]

- A popular nonlinear optimization technique is McCormick factorization (McCormick 1983), which replaces nonlinear subexpressions with auxiliary variables to obtain a linear relaxation. For example, the bilinear term \( xy \) can be linearized by replacing it with a new variable \( z \) and adding the following constraints to the relaxation:
  \[
  L_y x + L_x y - L_x L_y \leq z \leq L_y x + U_x y - L_x U_y
  \]
  \[
  U_y x + U_x y - U_x U_y \leq z \leq U_y x + L_x y - U_x L_y
  \]
  where \( x \in [L_x, U_x] \) and \( y \in [L_y, U_y] \). Factorizations of different constraints may create variables for identical subexpressions, and these variables must be identified to obtain a tight relaxation.
• Piecewise-linear functions are commonly modeled with auxiliary variables. A piecewise-linear function \( f(x) \) defined on a set of breakpoints \( \{d_k \mid k \in K\} \) can be modeled

\[
x = \sum_{k \in K} d_k \lambda_k, \quad \sum_{k \in K} f(d_k) \lambda_k
\]

\[
\sum_{k \in K} \lambda_k = 1, \quad \lambda_k \geq 0, \text{ for all } k \in K
\]

where the new variables \( \lambda_k \) form an SOS2 set (Beale and Tomlin 1970). When the problem contains two functions \( f(x), g(x) \) based on the same break points, their reformulations should use the same \( \lambda_k \)s.

• Constraint programmers frequently model a problem using two or more related sets of variables, only one of which is necessary to formulate the problem. The auxiliary variables allow the user to write redundant constraints that result in more effective propagation and therefore faster solution. For example, an assignment problem can use variables \( x_i \) that indicate which job is assigned to worker \( i \), and variables \( y_j \) that indicate which worker is assigned to job \( j \). The two sets of variables are related by channeling constraints \( j = x_i y_j \) and \( i = y_j x_i \), which should be deduced by the solver if they are not explicitly written by the modeler.

• The modeling languages of several modern optimization packages allow for convenience statements, which may require the modeling system to introduce auxiliary variables. For example, to index a vector \( v \) with a variable \( y \), systems such as AMPL (Fourer et al. 2002), OPL (Van Hentenryck et al. 1999), and Comet (Van Hentenryck and Michel 2005) allow the user to write a variably indexed expression \( v[y] \) instead of having to explicitly use the well-known \textit{element} constraint (Van Hentenryck and Carillon 1988). The modeling system replaces \( v[y] \) with a new variable \( z \), which is then related to \( v \) and \( y \) through the constraint \textit{element}(y,v,z). This constraint sets \( z \) equal to the \( y \)th element of the array \( v \). When \( v[y] \) occurs repeatedly, it should be replaced by the same variable \( z \), and only one \textit{element} constraint generated.

• Modeling systems, particularly in constraint programming, commonly provide high-level statements for modeling temporal constraints in scheduling problems. For example, global constraints may be written in terms of interval-valued variables that represent a period of time (IBM 2009a). Constraints that use the same interval variables may give rise to auxiliary variables that should be identified.
These and other situations can be accommodated by writing specialized code for each one. However, semantic typing provides a general and principled method for managing auxiliary variables. The typing mechanism can also help to structure the modeler’s thinking and avoid modeling mistakes.

3. A Motivating Example

We begin with a simple modeling example that illustrates predicates and semantic typing. We use a rudimentary modeling pseudo-language written in teletype font in which reserved words appear underlined. We follow the convention that numbered pseudocode statements are written by the user, while unnumbered statements are automatically generated by the modeling system.

A company would like to determine how to allocate 10 advertising spots to 5 products, with at most 4 spots for any one product. To concentrate resources, it will purchase spots for at most 3 of the 5 products, and it will purchase 4 spots for at least one product. Because the additional profit generated is nonlinearly related to the number of spots, we will suppose the objective function is given in tabular form. Specifically, $P_{ij}$ is the additional profit generated by allocating $j$ spots to product $i$, and the objective is to maximize total additional profit.

The problem can be formulated with a two-place predicate, allocate, that relates each product to the number of spots allocated to it. The optimization model can begin as follows:

1. spots in \{0..4\}; # Number of spots
2. product in \{A,B,C,D,E\}; # Product IDs
3. data P\{product, spots\}; # Matrix containing the profit data
4. $x[i]$ is howmany spots allocate(product i);

Lines 1 and 2 associate sets with the user-defined concepts spots and product, and line 3 retrieves the profit data. Line 4 declares $x_i$ to be the number of spots allocated to product $i$. The keyword is indicates that $x_i$ is a variable, and the phrase howmany spots indicates that $x_i$ is an integer quantity connected with spots (howmuch would indicate a continuous variable). We will see that such keywords as howmany, which, when, and whether provide a great deal of flexibility for defining variable types in terms of predicates.

Variable indices can be used to model the objective max $\sum_{i=1}^{5} P_{ix_i}$. 
5. maximize profit: \( \text{sum}\{\text{product } j\} \ P[i,x[i]] \);  
To enforce the limit on the total number of spots, the user writes  
6. maxspot: \( \text{sum}\{\text{product } i\} \ x[i] \leq 10; \)
where maxspot is the name given to this constraint. To model the remaining constraints, we will suppose that the modeler takes the traditional approach of using 0-1 variables. The user lets binary variable \( y_{ij} \) be 1 when \( j \) spots are allocated to product \( i \) and declares \( y_{ij} \) as follows:

\[
y[i,j] \text{ is whether } \text{allocate}(\text{product } i, \text{ spots } j);
\]
The predicate name allocate now occurs in two declarations, containing the keywords which and whether. This tells the system how \( y_{ij} \) is related to \( x_i \) and generates linking constraints if the user forgets to write them explicitly:

- assignment\{product i\}: \( \text{sum}\{\text{spots } j\} \ y[i,j] = 1; \)
- link\{product i\}: \( x[i] = \text{sum}\{\text{spots } j\} \ j*y[i,j]; \)
Now the user can write the remaining constraints:

7. choose3: \( \text{sum}\{\text{product } i\} \ y[i,0] \geq 2; \)
8. choose1: \( \text{sum}\{\text{product } i\} \ y[i,4] \geq 1; \)
Line 7 ensures that at most 3 products receive spots, and line 8 requires that at least one product receive 4 spots.

When this model is loaded into the solver, the objective function in line 5 must be linearized so that a linear relaxation, and a corresponding lower bound, is obtained. The solver rewrites line 5 as

\[
\text{maximize profit: } \text{sum}\{\text{product } i\} \ z[i];
\]
and posts constraints that relate the new \( z[i] \) variables to \( x[i] \) and \( P \):

\[
\text{elem}\{\text{product } i\}: \text{element}(x[i],P[i,*],z[i]);
\]
where \( P[i,*] \) represents the \( i \)th row of matrix \( P \). To complete the linear relaxation of the model, the solver relaxes the element constraint in line 12. One possible relaxation splits \( x_i \) into binary variables \( w_{ij} \) that indicate whether \( x_i = j \), and relates \( x_i \) and \( z_i \) to the \( w_{ij} \)s as follows:

\[
\sum_{j=0}^{4} w_{ij} = 1, \quad x_i = \sum_{j=0}^{4} jw_{ij}, \quad \text{for all } i 
\]

\[
z_i = \sum_{j=0}^{4} P_{ij}w_{ij}, \quad \text{for all } i
\]
Note that $w_{ij}$ is functionally the same variable as $y_{ij}$, which means the two should be identified. Furthermore, (3) is equivalent to lines 9 and 10. Semantic typing allows the modeling system to recognize these equivalences, resulting in a tighter linear relaxation. When the solver generates relaxation (3)–(4), it assigns $w_{ij}$ a semantic type as follows:

$$w[i,j] \text{ is whether allocate(product i, spots j);}$$

Because this type exactly matches the one for $y_{ij}$ in line 4, the solver replaces all occurrences of $w_{ij}$ in the relaxation with $y_{ij}$, for each pair $(i,j)$. The solver also generates the linking constraints in lines 7–8 if they are not already present. Examples in Section 6 illustrate how semantic typing can deduce more complicated relationships among variables.

4. Semantic Types and Predicates

Optimization models typically declare a variable by giving it a name and a canonical type, such as real, integer, binary, or string. However, stating that variable $x_i$ is integer does not indicate whether that integer is the ID of a machine or the start time of an operation. In other words, variable declarations say little about what the variable means. Some of its meaning may be recovered by examining the constraints in which the variable appears, but this is often ineffective. We argue that giving a more specific meaning to variables through semantic typing can be beneficial for a number of reasons, including its ability to address the variable management issue described above.

Semantic types can be supported by adding keywords and constructs to the grammar of the modeling language, or through menus and a point-and-click interface. We follow the modeling language approach throughout this paper.

We propose defining a variable’s semantic type by associating it with a predicate, generally a multi-place predicate. The variable is defined by relating it to the predicate by means of a keyword. In the advertising example, declaring $x_i$ to be howmany spots allocate(product i) creates a 2-place relation allocate(product, spots) and indicates that $x_i$ is the number of spots.

A predicate denotes a relation, or set of tuples. For instance, the predicate allocate denotes a set of pairs consisting of a product identifier and an assigned number of spots. We schematically indicate this relation

$$\begin{align*}
\text{product} & \quad \text{spots} \\
i & \quad x_i
\end{align*}$$
The relation can be viewed as a matrix in which the two columns are labeled as above and the rows are pairs \((i,x_i)\).

Keywords like \texttt{howman}y and \texttt{whether} pose queries to the relation, much as one might query a relational database. For example, by declaring a variable \(x_i\) to be \texttt{howman}y \texttt{spots allocate(product i)}, we ask what is the \texttt{spots} entry of the row whose \texttt{product} entry is \(i\). By declaring \(y_{ij}\) to be \texttt{whether allocate(product i, spots j)}, we ask whether \(j\) is the \texttt{spots} entry of the row whose \texttt{product} entry is \(i\).

Normally, when a declaration identifies a column with a subscripted variable such as \(x_i\), that column should be a \textit{function} of the other columns. That is, no two rows should contain different \(x_i\) entries when the other entries are the same. Thus when \(x_i\) is identified with the \texttt{spots} column, \texttt{spots} should be a function of the \texttt{product} column. The same principle is illustrated by the assignment problem mentioned earlier. The variable declarations are

1. \texttt{x[i] is which job assign(worker i)};
2. \texttt{y[j] is which worker assign(job j)};

Both declarations use the predicate \texttt{assign}, which denotes the relation

\[
\begin{array}{c}
\text{job} \quad \text{worker} \\
\text{j, } x_i \quad i, y_j
\end{array}
\]

Because either term of the relation is a function of the other, we have a bijection given by \(j = x_i\) or \(i = y_j\). Substituting the latter into the former yields the channeling constraint \(j = x_{y_j}\), and substituting the former into the latter yields \(i = y_{x_i}\).

If we wish to allow several jobs to be assigned to one worker, we can declare \(x_i\) to be a \textit{set-valued} variable:

1. \texttt{x[i] is whichset job assign(worker i)}

This means that \(x_i\) is the set of jobs assigned to worker \(i\). In this case, the \texttt{job} column need not be a function of the \texttt{worker} column. The channeling constraints are

\[
\begin{align*}
\text{For all } j & \quad j \in x_{y_j} \\
\text{For all } i, j & \quad i = y_j, \text{ with } j \in x_i
\end{align*}
\]

In practice, it is sometimes convenient to name a predicate after one of its terms. For example, the cost \(z_i\) incurred by activity \(i\) could be declared by introducing a predicate \texttt{incurs(activity,cost)}:
z[i] is howmuch cost incurs(activity i)

However, there is no real need to introduce a separate predicate name in this context. A simpler alternative is to name the predicate cost(activity,cost) and use the declaration

z[i] is howmuch cost(activity i)

which is shorthand for the formal declaration

z[i] is howmuch cost cost(activity i)

A special case is an unsubscripted cost variable z. We could introduce a predicate incurs(cost) and declare z to be howmuch cost incurs. However, a simpler alternative is to name the predicate cost and declare z to be simply howmuch cost, which is shorthand for howmuch cost(cost).

Every time the system detects relationships between variables, an alert (e.g. a pop-up window) can be displayed to the user asking for confirmation. This is useful both for error detection and training the user in the practice of semantic typing. If two variables are identified by mistake because the user assigned them (or other variables related to them) incorrect semantic types, such an alert would aid the user in finding and correcting the problem. Because omitting a semantic relationship does not make the model incorrect, when in doubt about the validity of a proposed variable relationship, the user can always choose not to enforce it. Similar types of alerts, or error messages, can be generated when other kinds of inconsistencies are detected in the user’s model, such as assigning identical semantic types to distinct variables.

5. Related Work

The idea of communicating problem structure to a solver is not new; it is underexploited. Modern linear and integer programming software such as CPLEX (IBM 2009b) and Xpress-Optimizer (Fair Isaac Corporation 2009) can detect network structure in an optimization model and use the more efficient network simplex method (Dantzig 1951). Special ordered
sets of type 1 or 2 (Beale and Tomlin 1970) convey additional information to a solver with the intent of improving performance.

Metaconstraints, however, provide a more general mechanism for exploiting specific problem structure within a model. They are a standard feature of constraint programming, where they are known as global constraints (Beldiceanu et al. 2011) and are key to the success of the field. Metaconstraints are also supported in one form or another by several high-level modeling systems that go beyond constraint programming. These include AMPL (Fourer et al. 2002), ECLiPSe (Ajili and Wallace 2003), SIMPL (Yunes et al. 2010) (prototype), Xpress-Kalis (Heipcke 2009), and Zinc (Marriott et al. 2008). For example, when using Gecode (Gecode Team 2006) as the constraint programming solver in AMPL, the user can impose an \textit{alldifferent} constraint on a vector of variables \(x\) and pick a bounds-consistent propagation algorithm by using a suffix notation as follows:

\[
\text{alldiff}\{i \text{ in } 1..n\} \; x[i] \; \text{suffix icl icl\_bnd};
\]

In ECLiPSe the user can write

\[
\text{[eplex,ic]}: (x + 2 \geq y)
\]

to indicate that the constraint \(x + 2 \geq y\) should be sent to both the linear programming solver (\texttt{eplex}) and the constraint programming solver (\texttt{ic}). In Zinc, the code

\[
\text{var int: x :: bounds}
\]

\[
\text{constraint(x >= y) :: solver(lp) :: solver(fd)}
\]

indicates that bounds propagation is to be performed on the domain of variable \(x\), and the constraint \(x \geq y\) will be handled by an LP and a finite domain (FD) solver. In SIMPL's prototype modeling language, the user writes

\[
\text{knapsack means} \{
\text{sum i a[i]*x[i] <= C}
\text{relaxation = \{lp, cp\}}
\text{inference = \{cover\}}
\}
\]

to declare a metaconstraint named \texttt{knapsack} that consists of a knapsack constraint whose relaxation will be handled by an LP and a CP solver, and that will infer cover inequalities during search.

Typed modeling languages have been proposed as an approach to \textit{model management}, which is inspired by concepts from object-oriented programming. The primary goal of
model management is to allow one to combine models or use inheritance as in C++. In an early study (Bradley and Clemence 1988), the authors present straightforward object-oriented modeling and use types to manage variables. In Bhargava et al. (1998), the authors give formal semantics for Ascend, which is a strongly-typed object-oriented modeling language. SML (Geoffrion 1992a,b) is an implementation of the structured-modeling framework that exploits the advantages of strong typing in detecting numerous kinds of errors and inconsistencies in models. Semantic types are analyzed in Bhargava et al. (1991) under the name of *quiddity*, a concept from medieval philosophy. This work addresses the basic issue of how variable typing can allow synonymous variables to be identified when models are combined. They show how difficult it is to design valid sufficient conditions for identification, and they in fact do not attempt to provide valid conditions. They only flag variables that the user may want to identify. The key idea is to describe the quiddity of a variable with nested functions, such as cost(labor(production(truck))). Indices are given quiddities as well as variables.

Our goal is more general than model management in one sense, and more restricted in another. It is more general because we want to identify relations between variables other than simple identities. It is more restricted because we are not interested in combining models. We assume that the user writes a single model and takes care that a single name and declaration are used for each variable. We are primarily concerned with the management of auxiliary variables introduced by metaconstraints.

A few attempts to convey variable semantics to the solver already exist in high-level modeling languages. In AIMMS (Bisschop and Entriken 1993, Heerink 2012), the declaration of a set includes the declaration of an indexing variable for that set that cannot be used elsewhere. Therefore, by stating that $J$ is a set of jobs with index $j$, AIMMS tells the solver that $j$ is not only an integer, but also the ID of a job. When modeling job scheduling problems, OPL (Van Hentenryck et al. 1999), Comet (Van Hentenryck and Michel 2005), IBM ILOG CP Optimizer (IBM 2009a), and Xpress-Kalis (Heipcke 2009) (among others) have a special entity known as an *activity*, which possesses special variables named *start* and *end*. Hence, if $a$ is an activity, the variable $a.start$ is not only an integer or rational number, it represents the start time of $a$ in the schedule. The extent to which this specific meaning is exploited by the solver in each of the above systems is not always clear, but the developers certainly found them to be useful in some way. In the AIMMS
example, using \( j \) for something other than indexing \( J \) would trigger an error, which helps the user. In the activity example, the start and end variables can trigger the use of efficient scheduling-specific algorithms such as edge-finding (Carlier and Pinson 1990). In SymChaff (Sabharwal 2005, 2009), a SAT solver especially designed to efficiently handle symmetries, high-level descriptions of AI planning problems written in a Planning Domain Description Language (PDDL) can be annotated with special tags to indicate which variables (or variable groups) are symmetric or interchangeable. These symmetries are then used by the solver to improve branching decisions, enable symmetric learning, and reduce the search space. In (Sabharwal 2009), the author also uses the term “semantic meaning” to refer to the association between variables and the high-level objects they represent, which is lost when a formula is converted to the input format of a SAT solver (e.g. the DIMACS format).

In the F# programming language, the user can declare units of measure and attach them to variables or constants (Kennedy 2010). For example, we can define the units \( m \) (meter) and \( s \) (second) and then write the declaration `let gravityOnEarth = 9.808<\text{m/s}^2>`.

In Lopes and Fourer (2009), the authors propose a graphical modeling language based on the Unified Modeling Language (Object Management Group, Inc. 2010) to facilitate the communication of multistage stochastic linear programs with recourse between diverse stakeholders in an OR project. Their extended diagrams allow the modeler to achieve a significant level of detail by using *adornments*, which are optional graphical markers that “add semantic value” to the representation of elements in the model. With the aid of adornments, many algebraic expressions can be easily derived from the model’s diagrams. Another, secondary, role of adornments is to make it easy to spot inconsistencies between the graphical and algebraic descriptions of the problem.

From a certain perspective, the kind of modeling language we propose would be, to a traditional modeling language, what XML (Bray et al. 2004) is to HTML: it introduces new grammar in a way that allows users to create their own predicates (their own sub-language). This perspective is similar to the meta-language idea behind the embedded-languages approach in Bhargava and Kimbrough (1993).

Several past innovations in modeling have been about how to better communicate with a solver using a modeling language. In addition to contributing toward this goal, metaconstraints and semantic typing also contribute to another, related and equally important goal: better translating what resides in the modeler’s head to a modeling
language representation. Semantic typing as presented here differs from earlier work in that it provides semantic information necessary for managing auxiliary variables in the context of metaconstraint-based modeling. It supports the thoroughgoing use of metaconstraints as a mechanism to convey problem structure to the solver.

6. Additional Modeling Examples

6.1. Latin Squares

A Latin Square of order $n$ is an $n \times n$ square of numbers ranging from 1 to $n$ such that the numbers in each row and column are distinct. Latin Squares (a.k.a. Euler squares of degree 1) were first proposed by Euler (Euler 1849). They have many practical applications such as experimental design, error-correcting codes, and parallel processor scheduling. The problem can be formulated in at least three ways: by assigning numbers $x_{ij}$ to row-column pairs $(i, j)$, by assigning columns $y_{ik}$ to row-number pairs $(i, k)$, and by assigning rows $z_{jk}$ to column-number pairs $(j, k)$. To obtain stronger propagation, we will formulate the problem in all three ways simultaneously and allow the solver to deduce channeling constraints. For this we need only one 3-place predicate $\text{assign}$. The declarations are:

1. row, column, number in $\{1..n\}$;
2. $x[i,j]$ is which number $\text{assign}(\text{row } i, \text{column } j)$;
3. $y[i,k]$ is which column $\text{assign}(\text{row } i, \text{number } k)$;
4. $z[j,k]$ is which row $\text{assign}(\text{column } j, \text{number } k)$;

The three formulations can now be written using the well-known all-different constraint:

5. numrow$\{\text{row } i\}$: $\text{alldiff}(x[i,*])$; numcol$\{\text{column } j\}$: $\text{alldiff}(x[^*,j])$;
6. colrow$\{\text{row } i\}$: $\text{alldiff}(y[i,*])$; colnum$\{\text{number } k\}$: $\text{alldiff}(y[^*,k])$;
7. rowcol$\{\text{column } j\}$: $\text{alldiff}(z[j,*])$; rownum$\{\text{number } k\}$: $\text{alldiff}(z[^*,k])$;

The $\text{assign}$ predicate denotes the relation

$$i, z_{jk} \quad j, y_{ik} \quad k, x_{ij}$$

Because the three terms correspond to which variables, channeling constraints connect all three variables. For example, substituting $j = y_{ik}$ and $k = x_{ij}$ into $i = z_{jk}$ yields the first channeling constraint below:

$$i = z_{yk} x_{ij}, \quad j = y_{zk} x_{ij}, \quad k = x_{zj} y_{ik}, \quad \text{for all } i, j, k$$
The remaining constraints are similarly derived. To create a linear relaxation of the channeling constraints, the system introduces three new sets of binary variables, \( \delta^x_{ijk} \), \( \delta^y_{ijk} \), \( \delta^z_{ijk} \), as well as the following additional constraints:

\[
\begin{align*}
  x_{ij} &= \sum_k k \delta^x_{ijk} \quad \text{and} \quad \sum_k \delta^x_{ijk} = 1, \quad \text{for all } i,j \\
  y_{ik} &= \sum_j j \delta^y_{ijk} \quad \text{and} \quad \sum_j \delta^y_{ijk} = 1, \quad \text{for all } i,k \\
  z_{jk} &= \sum_i i \delta^z_{ijk} \quad \text{and} \quad \sum_i \delta^z_{ijk} = 1, \quad \text{for all } j,k.
\end{align*}
\]

Here is where semantic typing makes a difference. Given the semantic types of \( x, y, \) and \( z \), together with the semantics of the variable-indexing constraints being relaxed, variables \( \delta^x_{ijk}, \delta^y_{ijk}, \) and \( \delta^z_{ijk} \) automatically receive the same semantic type \text{whether assign}(\text{row } i, \text{column } j, \text{number } k). Hence, the system infers that the problem relaxation can be strengthened by adding

\[
\delta^x_{ijk} = \delta^y_{ijk} = \delta^z_{ijk}, \quad \text{for all } i,j,k.
\]

### 6.2. Nurse Scheduling

This example was taken from Section 4.6 of Hooker (2011). Nurses are to be assigned to shifts on each day of the week. The assignments can be indicated with variables \( w_{sd} \) that indicate which nurse to assign to shift \( s \) on day \( d \), or variables \( t_{id} \) that indicate which shift to assign to nurse \( i \) on day \( d \). Some of the constraints can be written with standard global constraints using \( w_{sd} \), some using \( t_{id} \), and some using either set of variables. The global constraints are therefore combined in a single model that contains both types of variables, which are declared:

1. nurse in \{a,b,c,...\}; shift in \{1,2,3\}; day in \{Mon,...,Sun\};
2. \( w[s,d] \) is which nurse assign(shift s, day d);
3. \( t[i,d] \) is which shift assign(nurse i, day d);

The assign predicate denotes the relation

\[
\text{assign} \quad \text{nurse shift day} \\
i, w_{sd} \quad s, t_{id} \quad d
\]

Because only two columns are associated with \text{which} variables, the system deduces two sets of channeling constraints:

\[
i = w_{tid}, \quad \text{for all } i,d
\]
\[ s = t_{w,s,d}, \text{ for all } s,d \]

The first is obtained by substituting \( s = t_{id} \) into \( i = w_{sd} \), and similarly for the second. When relaxing these constraints, the system creates auxiliary binary variables \( \delta_{sid}^t \) for \( t_{id} \) and \( \delta_{isd}^w \) for \( w_{sd} \) and infers the semantic types

\[
\begin{align*}
deltat[s,i,d] & \text{ is whether assign(nurse } i, \text{ day } d, \text{ shift } s); \\
deltaw[i,s,d] & \text{ is whether assign(shift } i, \text{ day } d, \text{ nurse } i); \\
\end{align*}
\]

The terms of the predicate `assign` are listed in a different order, but it is nonetheless the same predicate because the same multiset of terms appears. The system posts the linking constraints

\[
\begin{align*}
t_{id} &= \sum_s s \delta_{sid}^t \text{ and } \sum_s \delta_{sid}^t = 1, \text{ for all } i,d \\
w_{sd} &= \sum_i i \delta_{isd}^w \text{ and } \sum_i \delta_{isd}^w = 1, \text{ for all } s,d \\
\end{align*}
\]

### 6.3. Piecewise-Linear Optimization

Because of their importance and wide-ranging applicability, piecewise-linear meta/global constraints are already present in many modern modeling systems. To exemplify the usefulness of semantic typing in this context, we consider two relaxations of piecewise-linear constraints: one for the continuous case, and another for the discontinuous case. In both cases, we analyze a model with two piecewise-linear constraints that share variables.

**Continuous Functions** Suppose that cost \( f(x) \) is a continuous piecewise-linear function of output \( x \). The breakpoints are given in the array \( A = (a_1, \ldots, a_n) \), and the corresponding values of \( f(x) \) are given in the array \( C = (c_1, \ldots, c_n) \). Thus \( f(x) \) is linear on each interval \([a_i, a_{i+1}]\), with \( f(a_i) = c_i \) and \( f(a_{i+1}) = c_{i+1} \). We use a metaconstraint `piecewise` to model the function and write

1. `index in {1..2}`;
2. `data A{i in index}, C{i in index};`
3. `x is howmuch output;`
4. `z is howmuch cost;`
5. `piecewise(x,z,A,C);`
where \( z \) is a new variable that plays the role of \( f(x) \). This \textit{piecewise} constraint can be relaxed as follows:

\[
x = a_1 + \sum_{i=1}^{n-1} \bar{x}_i, \quad z = c_1 + \sum_{i=1}^{n-1} \frac{c_{i+1} - c_i}{a_{i+1} - a_i} \bar{x}_i
\]

\((a_{i+1} - a_i) \delta_{i+1} \leq \bar{x}_i \leq (a_{i+1} - a_i) \delta_i, \quad \delta_i \in \{0, 1\}, \quad \text{for } i = 1, \ldots, n - 1\)

where \( \delta_i \) indicates whether \( x \geq a_i \), and \( \bar{x}_1, \ldots, \bar{x}_{n-1} \) is a disaggregation of \( x \) corresponding to the break points in \( A \).

The \textit{piecewise} constraint induces the system to create a 2-place predicate \texttt{output.A} and declare auxiliary variables \( \bar{x}_i, \delta_i \) as follows:

\begin{verbatim}
xbar[i] is howmuch output.A(index i);

delta[i] is whether lastpositive output.A(index i);
\end{verbatim}

The predicate name \texttt{output.A} is inherited from the original predicate name and the array \( A \) of breakpoints. The declaration of \( \bar{x}_i \) says that \( \bar{x}_i \) is the amount of the value of \( x \) allocated to \( \bar{x}_i \). Formally, it creates the new predicate \texttt{output.A(index,output)} from the original predicate \texttt{output(output)} and declares \( \bar{x}_i \) to be \texttt{howmuch output output.A(index i)}. The new keyword \texttt{lastpositive} queries \texttt{output.A} to determine the last interval that receives a positive allocation. Thus \( \delta_i \) indicates whether \( i \) is the last such interval. One could also define a variable \( \epsilon \) with the declaration

\begin{verbatim}
epsilon is which lastpositive output.A
\end{verbatim}

to indicate which is the last interval to receive a positive allocation, but such a variable is not used in the relaxation.

Now let us assume the model contains another piecewise-linear constraint on \( x \) that uses the same breakpoints, such as \texttt{piecewise(x,z',A,C')} . When this constraint is relaxed, it introduces auxiliary variables \( \bar{x}'_i \) and \( \delta'_i \), as well as a linear relaxation that is analogous to (5). The semantic types of \( \bar{x}'_i \) and \( \delta'_i \) will match the semantic types above, and the system will automatically infer that \( \bar{x}_i = \bar{x}'_i \) and \( \delta_i = \delta'_i \), for all \( i \).

**Discontinuous Functions** Let \( f(x) \) be a piecewise-linear cost function of flow variable \( x \). The function is linear on possibly disjoint intervals \([\ell_1, u_1], \ldots, [\ell_n, u_n]\), where \( c_i = f(\ell_i) \), \( d_i = f(u_i) \), and \( d_i = c_{i+1} \) when \( u_i = \ell_{i+1} \). We let \( L = (\ell_1, \ldots, \ell_n) \), and similarly for \( U, C \), and \( D \). We use a metaconstraint \texttt{piecewise2} that accommodates this kind of discontinuity:
1. index in \{1..m\};
2. data L\{i in index\}, U\{i in index\}, C\{i in index\}, D\{i in index\};
3. x is howmuch flow;
4. z is howmuch cost;
5. piecewise2(x,z,L,U,C,D);

One possible linear relaxation of this constraint is

\[ x = \sum_{i=1}^{n} (\lambda_i \ell_i + \mu_i u_i), \quad z = \sum_{i=1}^{n} (\lambda_i c_i + \mu_i d_i) \]

\[ \lambda_i + \mu_i = \delta_i, \quad \text{for } i = 1, \ldots, n \]

\[ \sum_{i=1}^{n} \delta_i = 1 \]

where \( \lambda_i, \mu_i, \) and \( \delta_i \) are new auxiliary variables. To implement this relaxation, the piecewise2 constraint introduces a predicate flow.L.U(index,flow). The new predicate could be used to define a variable

\[ xbar[i] \text{ is howmuch flow.L.U(index i)} \]

but no such variable is used in the relaxation. Rather, the system declares auxiliary variables

\[ \text{lambda[i] is lowermult flow.L.U(index i)}; \]
\[ \text{mu[i] is uppermult flow.L.U(index i)}; \]
\[ \text{delta[i] is whether positive flow.L.U(index i)}; \]

The keywords lowermult and uppermult query the values of multipliers that yield the flow allocated to interval \( i \). The keyword positive queries which interval receives positive flow.

Now if the same variable \( x \) appears in another piecewise-linear constraint piecewise2(x,z'L,U,C',D') defined on the same intervals, new auxiliary variables \( \lambda'_i, \mu'_i, \) and \( \delta'_i \) are created, as well as a new set of constraints resembling (6). These auxiliary variables receive the same semantic types as \( \lambda_i, \mu_i, \) and \( \delta_i \), respectively, and the variables are identified.
6.4. Disjunctions of Linear Systems

Let \( x \) be a vector of decision variables, and let \( \{1, \ldots, n\} \) index a set of mutually exclusive scenarios in an optimization problem. Assume that a model for this problem contains the following two constraints:

\[
\bigvee_i (A^i x \geq b^i) \tag{7}
\]
\[
\bigvee_i (C^i x \geq d^i) \tag{8}
\]

where both constraints depend on the same choice of scenario from the same set. While the user could have combined both constraints into a single disjunctive statement, there is no such guarantee. Therefore, we will assume (7) and (8) appear as separate constraints to exemplify the benefits of semantic typing. Another case in which (7) and (8) might appear as separate disjuncts arises when the system itself creates disjunctive representations of metaconstraints (for example, as an intermediate step toward a linear relaxation).

To make the example more concrete, assume that \( x = (x_1, \ldots, x_m) \), where \( x_j \) is the production level of a given product \( j \), and that \( \{1, \ldots, n\} \) indexes a set of configurations of the production environment. Therefore, we can write

1. product in \( \{1..m\} \);
2. config in \( \{1..n\} \);
3. \( x[j] \) is howmuch output(product \( j \));
4. disjunction1: or \{config \( i \)\} \( (A[i,*]x \geq b[i]) \);
5. disjunction2: or \{config \( i \)\} \( (C[i,*]x \geq d[i]) \);

Because the disjunctions in lines 4 and 5 are over the same set of alternatives (config), the solver assumes that the same disjunct is selected in each.

Using the standard convex hull formulation shown in Section 2, the solver would reformulate lines 4 and 5 as (9) and (10), respectively:

\[
x = \sum_i x^A_i, \quad A^i x^A_i \geq b^i \delta^A_i, \quad \sum_i \delta^A_i = 1, \quad \delta^A_i \in \{0, 1\}, \text{ all } i
\]  

\[
x = \sum_i x^C_i, \quad C^i x^C_i \geq d^i \delta^C_i, \quad \sum_i \delta^C_i = 1, \quad \delta^C_i \in \{0, 1\}, \text{ all } i
\]  

The corresponding relaxations are obtained by making \( \delta \) nonnegative rather than binary. Because the set of scenarios is the same in both cases, it is correct (and beneficial) to set
\( x_i^A = x_i^C \) and \( \delta_i^A = \delta_i^C \) for all \( i \). Semantically, \( x_A \) and \( x_C \) are associated with a new predicate `output.config` that is inherited from the predicate `output` and the index set `config`:

\[
xA[i,j] \text{ is howmuch output.config(config i, product j)};
\]
\[
xC[i,j] \text{ is howmuch output.config(config i, product j)};
\]

Because the types are the same, \( x_A \) and \( x_C \) are identified, as desired. To declare semantic types for \( \delta_i^A \) and \( \delta_i^C \), the system creates a predicate `choice.config` that is inherited from `config` but not from `output`. This is because the same set `config` of alternatives may appear in disjunctions that use different variables than \( x \). The declarations are:

\[
deltaA[i] \text{ is whether choice.config(config i)};
\]
\[
deltaC[i] \text{ is whether choice.config(config i)};
\]

This results in the identification of \( \delta_i^A \) and \( \delta_i^C \).

In some modeling contexts, the user may wish to enforce additional constraints \( C_i \) when configuration \( i \) is chosen in disjunctions (Hooker 2011). The user need only introduce variables \( y_i \), declare them `whether choice.config(config i)`, and write constraints of the form \( y_i \rightarrow C_i \). The modeling system will identify \( y_i \) with \( \delta_i \) and enforce \( C_i \) appropriately.

### 6.5. Temporal Modeling with Interval Variables

Variables that represent time intervals have proved useful for the formulation of scheduling problems (IBM 2009a). Interval variables can give rise to auxiliary variables, which can then be managed by their semantic types.

Suppose, for example, we wish to formulate a scheduling problem in which the processing of job \( j \) must occur entirely within a time interval \( W_j \). Each job has duration \( D_j \) and consumes resource at the rate \( R_j \). The jobs running at any one time must consume resources at a rate no greater than \( L \). If \( x_j \) is the time interval occupied by the processing of job \( j \), the model is:

1. job in \{1..n\};
2. time in \{1..T\};
3. data \( W\{\text{job } j\}, D\{\text{job}\}, R\{\text{job}\}, L; \)
4. running in [time, time];
5. \( x[j] \text{ is when running schedule(job j) subset } W[j]; \)
6. \text{cumulative}(x, D, R, L);
where \([\text{time, time}]\) in line 4 is the set of intervals \([i, j]\) with \(i < j\) and \(i, j \in \text{time}\). Because \text{running} is an interval, the declaration in line 5 implies that \(x_j\) is an interval-valued variable. The declaration also sets an initial domain \(W_j\) for \(x_j\) and so imposes a time window. The cumulative constraint in line 6 is well known in constraint programming and requires that the resource consumption at any one time be at most \(L\).

Let us assume that the solver reformulates the cumulative constraint as a mixed integer program. One formulation uses binary variable \(\delta_{jt}\) to indicate whether job \(j\) starts at time \(t\), and \(\phi_{jt}\) to indicate whether job \(j\) is running at time \(t\). Variables \(\delta_{jt}\), \(\phi_{jt}\) for \(t \notin W_j\) do not appear. The problem can be formulated

\[
\sum_t \delta_{jt} = 1, \text{ all } j \\
\phi_{jt} \geq \delta_{jt'}, \text{ all } t, t' \text{ with } 0 \leq t - t' < D_j, \text{ all } j \\
\sum_j R_j \phi_{jt} \leq L, \text{ all } t
\]

The new variables are linked to the old ones by

\[
\delta_{jt} = \begin{cases} 
1 & \text{if } x[j].\text{start} = t \\
0 & \text{otherwise}
\end{cases} \quad \phi_{jt} = \begin{cases} 
1 & \text{if } t \in x_j \\
0 & \text{otherwise}
\end{cases}
\]

where \(x[j].\text{start}\) is the start time of interval \(x_j\). The new variables are declared as follows:

\text{delta}[j,t] is whether \text{running.start schedule}(job j, time t);

\text{phi}[j,t] is whether \text{running schedule}(job j, time t);

These declarations introduce two new 3-place predicates that are denoted by \text{schedule} but distinguished by the terms they relate.

So far, there is no need for these semantic types. But suppose we want job finish times to be separated by at least \(T_0\) minutes, to allow employees to unload the jobs. This can be modeled

\text{unload}(job j, job k): j < k \implies |x[j].\text{end} - x[k].\text{end}| \geq T_0;

A possible mixed integer formulation introduces a binary variable \(\epsilon_{jt}\) to indicate whether job \(j\) ends at time \(t\). The constraint becomes

\[
\epsilon_{jt} + \epsilon_{kt'} \leq 1, \text{ all } t, t' \text{ with } 0 < t' - t < L_0, \text{ all } j, k \text{ with } j \neq k.
\]
These new variables are linked to the old ones by
\[
\epsilon_{jt} = \begin{cases} 
1 & \text{if } x[j].\text{end} = t \\
0 & \text{otherwise}
\end{cases}
\]

However, when (11) and (12) are combined to obtain a mixed integer formulation of the entire problem, nothing in the formulation captures the relationship between \(\epsilon_{jt}\) and the other variables. This is remedied when the solver generates a semantic type for \(\epsilon_{jt}\):

- \texttt{epsilon[j,t] is whether running.end schedule(job j, time t);} 

The solver associates the predicate \texttt{schedule(running.end, job j, time t)} with the predicate \texttt{schedule(running.start, job j, time t)} in the type declaration of \(\delta_{jt}\) and deduces that
\[
\epsilon_{j,t+D_j} = \delta_{jt}, \, \text{all } j,t.
\] (13)

It also associates \texttt{schedule(running.end, job j, time t)} with the predicate \texttt{schedule(running, job j, time t)} in the declaration of \(\phi_{jt}\) and deduces the redundant constraints
\[
\phi_{jt} \geq \epsilon_{jt'}, \, \text{all } t,t' \text{ with } 0 \leq t' - t < D_j, \, \text{all } j.
\] (14)

Constraints (13)–(14) can now be added to the mixed integer formulation.

### 6.6. Traveling Salesman with Side Constraints

Consider a traveling salesman problem (TSP) defined over a graph \(G = (V,A)\) with distances \(D_{ij}\) between every pair of cities \(i,j \in V\). As our final example, we model this TSP with two additional side constraints: some cities must precede other cities in the tour, and some arcs are missing from \(A\) (i.e. \(G\) is not a complete graph).

The problem data are declared as

1. \texttt{data D{city, city};} # Distance between cities
2. \texttt{data Prec{city, city};} # Prec[i,j]=1 if i must precede j
3. \texttt{data Succ{city};} # Set of possible successors of each city

A city \(j\) is omitted from the set \(\text{Succ}[i]\) to indicate that arc \((i,j)\) is missing from \(G\). Given a city \(i\), let variables \(x_i\) and \(s_i\) represent, respectively, the position of city \(i\) and the successor of city \(i\) in the tour. Their semantic types introduce a predicate \texttt{ordering(city,position)} that relates each city to its position in the ordering:
4. city in {1..n}; position in {1..n};

5. x[i] is which position ordering(city i) in {1..n};

6. s[i] is successor city ordering(city i) in Succ[i];

The keyword successor queries the predicate ordering for the city that follows city i. The keyword presupposes that the predicate is introduced in another declaration and therefore assumes it has the form ordering(city,position) rather than ordering(city,city).

By initializing s_i to belong to Succ[i], line 6 requires that the tour avoid missing arcs.

We are now ready to write the constraints.

7. prec{city i, city j | Prec[i,j] = 1}: x[i] < x[j];

8. alldiff(x);

9. circuit(s);

Line 7 imposes the precedence constraints. It is not possible to represent the precedence constraints using only s_i variables and, conversely, it is not possible to restrict the successors of a city using only x_i variables. Therefore, this model requires the dual viewpoint provided by the two sets of variables. The constraint in line 8 states that each city must have a distinct position in the tour, and the global constraint circuit (Laurièr 1978) in line 9 ensures that the collection of successor values assigned to the s_i variables represents a single closed tour. To complete the model, we write the objective function as

10. minimize dist: sum{city i} D[i,s[i]];

The solver can give the alldiff a conventional assignment model by introducing 0-1 variables z_{ik} to represent whether city i is in position k:

\[ \sum_{k=1}^{n} z_{ik} = 1 \text{ for all } i, \sum_{i=1}^{n} z_{ik} = 1 \text{ for all } k, \quad x_i = \sum_{k=1}^{n} k z_{ik} \text{ for all } i. \]

The third set of constraints links x_i to the new variables, which are declared

z[i,k] is whether ordering(city i, position k);

The solver can model the circuit constraint by introducing 0-1 variables w_{ij} to represent whether city j immediately follows city i, and then generating valid inequalities for the TSP (Ruland and Rodin 1998), as well as cuts in the s-space that are specific to the circuit constraint (Genç-Kaya and Hooker 2014). The new variables are linked to s_i by the constraints

s_i = \sum_{j=1}^{n} j w_{ij} \text{ for all } i

and are declared
\( w[i,j] \) is whether successor ordering(city i, city j);

The variables \( z_{ik} \) and \( w_{ij} \) are not identified because they have different semantic types. However, the successor keyword in the declaration of \( w_{ij} \) allows the system to detect that they are related by the linking constraints

\[
(z_{ik} = 1 \land z_{j,k+1} = 1) \Rightarrow w_{ij} = 1, \text{ for all } i, j, k,
\]

which can be linearized to \( z_{ik} + z_{j,k+1} - w_{ij} \leq 1 \). Moreover, the system also detects a link between the auxiliary variables \( w_{ij} \) and the original variables \( x_i \):

\[
(x_j - x_i = 1) \Rightarrow w_{ij} = 1, \text{ for all } i, j
\]

which can also be linearized, treated directly by a CP solver, and/or used during branching.

The variable index in the objective function of line 11 is treated in a similar fashion to the one in Section 3: (i) an element constraint is created for every \( i \): element(s[i],D[i,*],r[i]); (ii) the objective function is replaced with \( \sum_i r_i \); and (iii) the relaxation of the element constraints introduces auxiliary variables \( w'_{ij} \) that are identified with \( w_{ij} \).

To further illustrate the power of semantic typing, suppose the user wishes to write constraints in terms of a variable \( y_k \) that represents the city that occupies position \( k \). Its declaration is simply

11. \( y[k] \) is which city ordering(position k);

The system deduces the standard channeling constraints, namely \( x_{yk} = k \) for all \( k \), and \( y_{xi} = i \) for all \( i \). Moreover, the \( y_k \) variables allow the user to write an alternative objective function

12. minimize dist2: sum\{position k\} D[y[k],y[k+1]];

provided \( y_{n+1} \) is identified with \( y_1 \). The objective function is unpacked by replacing it with \( \sum_k d_{k,k+1} \) and adding element constraints

\[
\text{elem\{position k\}}: \text{element((y[k],y[k+1]),D,d[k,k+1])};
\]

Although the objective functions in lines 10 and 12 are theoretically equivalent, including both of them in the model might be beneficial. Depending on how branching and variable domain propagation evolve, one objective value might increase faster than the other, and the lower bound at any point during the search can be taken as the maximum of the two.
7. Some General Channeling Constraints

We can generalize the procedures for deriving channeling constraints for multiple which, whichset, and whether variables that are associated with the same predicate.

Multiple which variables occur when a predicate has terms \( \text{term}_1, \ldots, \text{term}_n \), and some of the terms correspond to which variables. Let us say that the first \( k \) terms correspond to which variables \( \text{x}^1_{i(1)}, \ldots, \text{x}^k_{i(k)} \), where \( i(j) \) is shorthand for \( i_1 \cdots i_{j-1}i_{j+1} \cdots i_n \). The corresponding relation is

\[
\text{term}_1 \cdots \text{term}_k \cdots \text{term}_{k+1} \cdots \text{term}_n
\]

\[
i_1, \text{x}^1_{i(1)} \quad i_k, \text{x}^k_{i(k)} \quad i_{k+1} \quad i_n
\]

For example, \( k = 2 \) in the above nurse scheduling problem, where \( \text{term}_1, \text{term}_2, \) and \( \text{term}_3 \) are nurse, shift, and day, respectively, and variables \( \text{x}^1_{i(1)} \) and \( \text{x}^2_{i(2)} \) are \( w_{sd} \) and \( s_{id} \).

The derivation of channeling constraints follows the pattern illustrated by this and the Latin squares problems. For any given \( j \in \{1, \ldots, k\} \) we have \( i_j = x^j_{i(j)} \). For each subscript \( i_\ell \) of \( i(j) \) we substitute \( i_\ell = x^\ell_{i(\ell)} \) to obtain the channeling constraints

\[
i_j = x^j_{i(1)} \cdots x^{j-1}_{i(j-1)} x^{j+1}_{i(j+1)} \cdots x^k_{i(k)} x^{k+1} \cdots x_{i(n)}, \text{ for all } i_1, \ldots, i_n, j = 1, \ldots, k
\]  

(15)

The whichset keyword can be viewed as specialized projection operator. Suppose, for example, that exactly one worker \( w_{jt} \) is assigned to make products of type \( j \) on day \( d \). However, a worker may make several product types on a given day, or a given product type on several days. We have the relation

\[
\text{worker producttype day}
\]

\[
i, w_{jd} \quad j \quad d
\]

with declarations

1. \( \text{w}[j,d] \) is which worker make(producttype \( j \), day \( d \));
2. \( \text{p}[i,d] \) is whichset producttype make(worker \( i \), day \( d \));
3. \( \text{t}[i,j] \) is whichset day make(worker \( i \), producttype \( j \));

Thus \( p_{id} \) is the set of product types made by worker \( i \) on day \( d \), and \( t_{ij} \) is the set of days on which worker \( i \) makes product type \( j \). We can view \( p_{id} \) as the projection, onto the second
term of the relation, of all tuples in the relation for which the first and third terms have values \((i,d)\), and similarly for \(t_{ij}\). The channeling constraints are

\[
  i = w_{jd}, \quad \text{for all } i,j,d \text{ with } j \in p_{id}, \ d \in t_{ij}
\]

\[
  j \in p_{w_{jd}}, \quad \text{for all } j,d \text{ with } d \in t_{w_{jd}}
\]

\[
  d \in t_{w_{jd}}, \quad \text{for all } j,d \text{ with } j \in p_{w_{jd}}
\]

In general, we can suppose we have a predicate \(\text{pred}\) denoting a relation of the form

\[
  \text{term}_0 \ldots \text{term}_n
\]

\[
  i, x^0_{j_1 \ldots j_n} \quad j_1 \quad j_n
\]

where \(x^0_{j_1 \ldots j_n}\) is a \textbf{which} variable corresponding to \(\text{term}_0\). Let \(x^1_{ij(1)}, \ldots, x^n_{ij(n)}\) be \textbf{whichset} variables corresponding to \(\text{term}_1, \ldots, \text{term}_n\). Then the channeling constraints are

\[
  i = x^0_{j_1 \ldots j_n}, \quad \text{for all } i,j_1, \ldots, j_n \text{ such that } j_k \in x^k_{j(1)} \text{ for } k = 1, \ldots, n
\]

\[
  j_k \in x^k_{x^0_{j_1 \ldots j_n}}, \quad \text{for all } k,j_1, \ldots, j_n \text{ such that } j_\ell \in x^\ell_{x^0_{j_1 \ldots j_n}} \text{ for all } \ell \in \{1, \ldots, n\} \setminus \{k\}
\]

Projections can also be defined by specifying a proper subset of a variable’s indices. For example, we can let \(p^1_i\) be the set of product types made by worker \(i\) on any day, or we can let \(t^1_j\) be the set of days on which product type \(j\) is made by any worker. They are declared

\[
  \text{p}1[i] \textit{ is whichset producttype make(worker i)};
\]

\[
  \text{t}1[j] \textit{ is whichset day make(producttype j)};
\]

These declarations have the intended effect only if the full 3-place predicate \textit{make} appears in the model, as in any of the above declarations of \(x[j,d]\), \(p[i,d]\), or \(t[i,j]\). The channeling constraints that relate variables \(w_{jd}\), \(p^1_i\) and \(t^1_j\) are

\[
  i = w_{jd}, \quad \text{for all } i,j,d \text{ with } j \in p^1_i, \ d \in t^1_j
\]

\[
  j \in p^1_{w_{jd}}, \quad \text{for all } j,d \text{ with } d \in t^1_j
\]

\[
  d \in t^1_j, \quad \text{for all } j,d \text{ with } j \in p^1_{w_{jd}}
\]

The pattern is generalized along lines similar to the above.

Multiple \textbf{whether} variables can be associated with the same predicate if some of the indices are omitted. Using the above example, let \(\delta^w_{ijd}\) indicate whether worker \(i\) makes product type \(j\) on day \(d\). The declaration is

\[
  \text{deltaw}[i,j,t] \textit{ is whether make(worker i, producttype j, day d)};
\]
We could also define a variable $\delta_{j_{id}}$ that indicates whether product type $j$ is made by worker $i$ on day $d$, and a variable $\delta_{d_{ij}}$ that indicates whether day $d$ is a day on which worker $i$ makes product type $j$. But as in the nurse scheduling example, these variables have the same declaration and therefore the same meaning as $\delta_{i_{jd}}$.

However, we can obtain distinct whether variables by specifying fewer indices and projecting onto those indices. For example, we can let $\delta_{p_{id}}$ indicate whether worker $i$ makes a product of some type on day $d$, and $\delta_{t_{ij}}$ whether worker $i$ makes product type $j$ on some day. This results in declarations

\begin{align*}
\text{deltap1}[i,d] & \text{ is whether product make(worker i, day d);} \\
\text{deltat1}[i,j] & \text{ is whether day make(worker i, producttype j)}
\end{align*}

The channeling constraints that relate $\delta_{w_{ijd}}$, $\delta_{p_{id}}$, and $\delta_{t_{ij}}$ are

\begin{align*}
& \text{for all } i \text{ and } d, \delta_{p_{id}} = 1 \text{ if and only if } \delta_{w_{ijd}} = 1 \text{ for some } j \\
& \text{for all } i \text{ and } j, \delta_{t_{ij}} = 1 \text{ if and only if } \delta_{w_{ijd}} = 1 \text{ for some } d
\end{align*}

It is straightforward to generalize this pattern. We can use as few indices as desired, as in the declaration

\begin{align*}
\text{deltap2}[i] & \text{ is whether product make(worker i);} \\
\end{align*}

This defines a variable $\delta_{i_{p2}}$ that indicates whether worker $i$ makes any type of product on any day. The channeling constraints are analogous to the above.

8. Final Remarks

We show how the concept of semantic typing of variables can work as a generic solution to a problem that arises in the context of modeling with metaconstraints. Namely, semantic typing enables a modeling system to identify relationships between auxiliary variables created by constraint relaxations in a generic fashion, without pre-defining or hard-coding the possible ways in which the user can write a particular model. The generality of this relationship detection is very important because it is impossible to predict how each user will represent constraints in a modeling language. Moreover, the specific meaning intended for decision variables cannot always be recovered automatically from the model by current modeling systems, which justifies the need for such meaning to be provided by the user.

In addition to the above main benefit, semantic typing serves other important purposes. The inclusion of semantics in variable declarations enables the system to detect new kinds of errors and inconsistencies. Furthermore, variable semantics can help with structure
detection such as the identification of symmetries and other kinds of inefficiencies in a model. For example, the use of a weak collection of constraints to model a problem structure that is known to have a stronger polyhedral representation.

One might wonder whether or not writing semantic types is harder than writing the model itself or, in other words, whether it is reasonable to expect users to correctly write semantic types. We believe that this is a matter of having enough practice. If modeling is taught with semantic typing in mind to begin with, it may become a natural way of thinking about the role of decision variables in a model. That is, semantic types would not be harder to master than traditional modeling already is. To confirm this hypothesis, however, it would be necessary to experiment with these ideas in a classroom setting.

References


