Automating Mathematical Program Transformations

Ashish Agarwal[1⋆], Sooraj Bhat[2], Alexander Gray[2], and Ignacio E. Grossmann[1]

[1] Dept. of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA 15213
[2] College of Computing, Georgia Institute of Technology, Atlanta, GA 30332

Abstract. Mathematical programs (MPs) are a class of constrained optimization problems that include linear, mixed-integer, and disjunctive programs. Strategies for solving MPs rely heavily on various transformations between these subclasses, but most are not automated because MP theory does not presently treat programs as syntactic objects. In this work, we present the first syntactic definition of MP and of some widely used MP transformations, most notably the big-M and convex hull methods for converting disjunctive constraints. We use an embedded OCaml DSL on problems from chemical process engineering and operations research to compare our automated transformations to existing technology—finding that no one technique is always best—and also to manual reformulations—finding that our mechanizations are comparable to human experts. This work enables higher-level solution strategies that can use these transformations as subroutines.

Key words: mathematical programming, program transformation, disjunctive constraints, convex hull method, mixed-integer constraints

1 Introduction

The equations governing engineering systems rarely dictate a unique solution. Usually, a designer needs to find the optimal solution amongst a space of feasible ones. Such constrained optimization problems are often expressed as mathematical programs (MPs), which consist of a numerical objective that is to be maximized (or minimized) subject to some constraints. Solving MPs efficiently is an important problem across science and engineering. The nature of the constraints allowed is a key issue affecting both the kinds of systems that can be represented and the efficiency of algorithms. An MP is more specifically called a linear program (LP) when the constraints and the objective are linear algebraic equations and inequalities on the reals. A mixed-integer linear program (MILP) additionally allows restricting variables to be integer valued, which allows expressing problems not possible in LP. We discuss a superset of these that also allows Boolean expressions and most importantly disjunctive constraints.

Throughout this work, the term *disjunctive constraint* refers to a disjunction over (in)equations involving reals, such as $x \leq 0 \vee y \leq 0$, and is unrelated to

⋆ Currently: Dept. of Computer Science, Yale University, ashish.agarwal@yale.edu

Boolean disjunction which is a statement purely over Boolean variables. Both are an important modeling tool. Unfortunately, most MP solvers cannot directly accept programs with Booleans or disjunctions as input. The currently best-known strategies reformulate the program into an equivalent MILP, for which there are good solvers.

One such efficient reformulation technique is Balas' convex-hull method [1]. Unfortunately, this technique presents some mechanization challenges: new variables need to be introduced, constraints must be modified, and new equations must be added. Balas' theory requires each disjunct to be bounded, which often is attained by adding a lower and upper bound for every variable in each disjunct; this increases the number of inequalities to be manipulated. In addition, one must decide how to handle nested disjunctions. The reformulation is error-prone not just because of the tedious algebra, but also because the resulting equations are non-intuitive. Even on small problems, it is challenging to recognize how the output represents the original constraint. Finally, one must of course be familiar with the reformulation methods to apply them. Automation is clearly called for.

The reformulations we present have been widely used by experts for many years. However, there has been limited to no support for them in MP software tools. We believe this is because current MP theory focuses on the study of the numerical behavior of algorithms and does not treat programs as syntactic objects. MPs are defined in a canonical matrix form, which does not support basic operations required for automating transformations such as variable introduction and compositional construction of programs. We demonstrate that the formal methods of language design capably address long standing needs in the mathematical programming community. Our contributions are the following:

- We provide the first, to our knowledge, formalization of the syntax, type system, and semantics of an MP language. The core theory contains useful constructs such as Boolean expressions and disjunctive constraints that allow practitioners to formulate programs in a more natural style and, more importantly, enables higher-level analysis.
- Enabled by this, we automate some important program transformations from our richer language to forms accepted by modern solvers. This is the primary contribution of the paper, and we hope to convince the reader that implementing them without a formal methods perspective would be difficult. The convex-hull and big-M methods are the most interesting, and we also provide others that are of practical importance.
- Finally, we provide an OCaml embedded domain-specific language (EDSL) for succinct construction of MPs, and a framework for applying the various reformulations. Our software outputs programs in the popular AMPL language and the industry standard MPS format, allowing us to pass the generated programs to existing solvers and study their behavior. We find that our software generates programs comparable to what a human expert would produce, and that no one technique always produces the most efficient reformulation, making it important to have a system that allows open experimentation.

2 Mathematical Programming

The standard definition of a linear program is

$$\max \left\{ c^T x \mid Ax \leq b, x \in \mathbb{R}^n \right\} \tag{1}$$

where c is a $n \times 1$ dimensional coefficient vector, x is an $n \times 1$ vector of real valued variables, A is an $m \times n$ coefficient matrix, and b is an $m \times 1$ vector of constants. Thus, $c^T x$ is a scalar, and the matrix inequality $Ax \leq b$ represents m individual inequalities. The inequalities represent a polyhedron, such as either region R^1 or R^2 in Figure 1a, and is called the feasible space of the LP.

Representing discrete choices requires a more expressive language than LP. We need a language that allows expressing not just R^1 or R^2 separately but their union $R^1 \cup R^2$. There are two rather distinct methods for accomplishing this. The first is to enrich LP with a discrete type, such as is done with mixed-integer linear programming (MILP). In MILP, variables may be integer or real valued. The standard definition [2] is

$$\max\{c^T x + h^T y \mid Ax + Gy \leq b, x \in \mathbb{R}^n, y \in \mathbb{Z}^p\} \tag{2}$$

where x and y represent vectors of real and integer variables, respectively.

However, integers are often not an intuitive model of discrete choice, and become prohibitively difficult for larger problems. Alternatively, LP can be enriched with disjunctive constraints, which lead to more compact and comprehensible models [1, 3]. The canonical matrix form of a disjunctive constraint is

$$\left[A^1 x \leq b^1\right] \vee \left[A^2 x \leq b^2\right] \tag{3}$$

We still do not have Boolean expressions, nor disjunctive constraints that are not in disjunctive normal form (DNF), nor an obvious way to insert new constraints or extract specific ones to manipulate. In short, these definitions do not provide an abstract syntax that can be operated on formally. These shortcomings are addressed in the following section.

3 A Language for Mathematical Programming

Our mathematical programming language consists of refined types ρ, expressions e, constraints c (called propositions in logic), and programs p:

$$\begin{aligned}
\rho ::= {} & [r_L, r_U] \mid [r_L, \infty) \mid (-\infty, r_U] \mid \texttt{real} \mid \langle r_L, r_U \rangle \mid \langle r_L, \infty) \mid (-\infty, r_U \rangle \\
& \mid \texttt{int} \mid \{\texttt{true}\} \mid \{\texttt{false}\} \mid \texttt{bool} && \text{(4a)} \\
e ::= {} & x \mid r \mid \texttt{true} \mid \texttt{false} \mid \texttt{not}\, e \mid e_1 \,\texttt{or}\, e_2 \mid e_1 \,\texttt{and}\, e_2 \\
& \mid -e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 && \text{(4b)} \\
c ::= {} & \texttt{T} \mid \texttt{F} \mid \texttt{isTrue}\, e \mid e_1 = e_2 \mid e_1 \leq e_2 \mid c_1 \vee c_2 \mid c_1 \wedge c_2 \mid \exists x : \rho \centerdot c && \text{(4c)} \\
p ::= {} & \max_{x_1:\rho_1,\ldots,x_m:\rho_m} \{e \mid c\} && \text{(4d)} \\
\Upsilon ::= {} & \bullet \mid \Upsilon, x : \rho && \text{(4e)}
\end{aligned}$$

A full discussion of the straightforward type system and semantics is available in [4]; here we present a high-level overview.

Programs A mathematical program p consists of an objective e that must be maximized subject to a constraint c. Minimizing is equivalent to maximizing $-e$. This definition is similar to (2) but the objective and constraint are not in a matrix form.

Expressions Expressions are either numeric or Boolean. They include variables, rational constants r, Boolean constants, and the usual numeric and Boolean operators. We wish only to support linear terms, and so the restriction on $e_1 * e_2$ is that e_1 has no free variables. Nonlinear programs are certainly important, but the transformations we are focusing on apply only to linear constraints.

Constraints The most common constraints are conjunctions or disjunctions over (in)equations on the reals. Disjunction $c_1 \vee c_2$ is the key novelty. Conjunction alone provides a language for expressing what is normally referred to as a system of linear equations in linear algebra.

In addition, we allow Boolean constraints in the form `isTrue` e, where e must be an expression of type `bool`. We distinguish between Boolean truth versus truth of numeric propositions (`true` and `false` versus `T` and `F`). This type distinction, embodied as a syntactic distinction in our definition, is essential since the algorithms for solving these classes of propositions are entirely different. The convex-hull and big-M methods are useful only for the disjunctive constraint $c_1 \vee c_2$ and should not be applied to the Boolean expression e_1 `or` e_2. Additionally, Boolean expressions can be negated, but there is no negation at the constraint level because MPs do not allow strict inequalities.

Although it is not common in the MP literature, we require that variables be explicitly introduced with an existential quantifier. This clarifies the semantics and provides the practical benefit of locally scoped variables. Universal quantifiers would extend our language to include semi-infinite programs, an interesting but less developed class of problems. Variables introduced at the program level behave as existentially quantified; the only distinction being that they can also be used in the objective.

Refined Types We use refined types—instead of simply using `bool` and `real`—so that we can provide a treatment of bounds (needed for both the convex-hull method and the big-M method) and to be able to represent integers classically (integers are a subset of the reals in conventional mathematics). Square brackets denote real intervals; angle brackets denote integer intervals.

Context We keep track of variable bounds with a refined type context, which is a list of variables associated with their bounds. This is more informative than the usual context used in typing judgments. It provides not just variables' types but also retains knowledge of restrictions on the variables' values.

Finally, we define free variables ($FV(e)$, $FV(c)$) and capture-avoiding substitution ($\{e/x\}\, e'$, $\{e/x\}\, c$) in the usual way.

Computation with Real Numbers Mathematical programs involve real numbers, which raises the issue of computing over them. This is a fundamental challenge being pursued by others in various contexts [5, 6]. It does not however affect the transformations we provide because they are purely syntactic manipulations, and all real expressions are carried through unaltered. We were careful to include only *rational* constants instead of reals in the syntax, but this is due to an unrelated issue: it is a specification of MILPs that constants be rational, else an optimum may not exist [2]. Despite the MP community's classical treatment of reals, it is interesting to note that their desired interpretation of disjunction and existential quantification is certainly constructive. It is expected that any MP solver explain how the constraints are satisfied by providing witnesses for all variables and information on which disjoint region the optimum was found in.

4 Transforming Syntactic Constructs

The class of programs covered by p include disjunctive constraints and Booleans, but the best solvers accommodate only mixed-integer linear programming (MILP) constraints which do not allow either of these forms. We pursue the standard strategy of transforming the richer constraint forms to lower-level MILP constraints, with the important distinction that our definitions lead to a software implementation.

We first turn our attention to transformations for disjunctive constraints $c_1 \vee c_2$. The methods make no use of standard logical laws, such as DeMorgan's (recall constraints cannot be negated). The general idea is that the dichotomy expressed by disjunction is embodied instead in the discrete nature of integer variables. An integer binary variable $y_i \in \{0, 1\}$ is associated with each i^{th} disjunct of a disjunction, and the disjunction is replaced by conjunction. Just one y_i is required to be 1 and only the constraints of the corresponding disjunct are enforced. Disjuncts $j \neq i$ are then reduced to tautologies. We now consider some specific methods; all preserve constraint linearity, which is important for solver efficiency.

Big-M Transformation. The big-M method states that (3) can be reformulated into the equivalent mixed-integer linear constraints

$$\begin{array}{ll} A^1x - b^1 \leq M^1(1 - y_1) & y_1 + y_2 = 1 \\ A^2x - b^2 \leq M^2(1 - y_2) & \end{array} \tag{5}$$

where $y_i \in \{0, 1\}$ and M^i are the so called big-M parameters. These are known upper bounds on $A^ix - b^i$. Consider $y_1 = 1$ and $y_2 = 0$. The second inequality reduces to $A^2x - b^2 \leq M^2$, which is trivially satisfied because, by definition, M^2 is an upper bound of its left-hand side. Effectively, the second disjunct is disregarded. The first inequality reduces to $A^1x - b^1 \leq 0$, which is the original first disjunct. Conversely, only the second disjunct is enforced when $y_1 = 0$.

The computational efficiency of this method is crucially dependent on the choice of the big-M parameters, of which there are quite a few since M^1 and

M^2 are vectors. Casual users often set them to some arbitrarily large value to avoid the effort of computing them. Even experts often resort to this because it preserves model modularity: changes to a variable's bounds would require searching through their entire program to verify that all the M's are still valid. A liberally large value mitigates this issue. In contrast, our automated solution preserves modeling simplicity while providing computational efficiency. We use interval arithmetic to compute tight big-M parameters automatically.

Our definition of the big-M method requires two auxiliary judgments to be first introduced. First, we need an operation for computing big-M parameters. Let $\Upsilon \vdash e \rightleftharpoons [\bar{r}_L, \bar{r}_U]$ be the judgment that computes lower and upper bounds $\bar{r}_L$ and $\bar{r}_U$ for the expression e in the refined context Υ, where $\bar{r}_L$ and $\bar{r}_U$ are from the affinely extended rationals; they may take on the values of $-\infty$ and ∞. Its definition uses interval arithmetic over unary negation and the binary operators $+$, $-$, and $*$ by propagating derived bounds from subterms to enclosing terms. For example, under the context $x : [-1, 2], y : [0, 100]$, the expression $-5 * x + y$ generates the interval $[-10, 105]$.

Second, we define an operation to convert an inequality to its big-M form. Let $\Upsilon \vdash e \otimes c \rightharpoonup c'$ be the judgment that rewrites constraint c to its big-M form c', where the e will supply the necessary $1 - y$ term. Its definition is

$$\frac{\Upsilon \vdash e_1 - e_2 \rightleftharpoons [\bar{r}_L, r_U]}{\Upsilon \vdash e \otimes e_1 \leq e_2 \rightharpoonup e_1 \leq e_2 + e * r_U} \tag{6a}$$

$$\frac{\left\{\Upsilon \vdash e \otimes c_j \rightharpoonup c'_j\right\}_{j \in \{A,B\}}}{\Upsilon \vdash e \otimes c_A \wedge c_B \rightharpoonup c'_A \wedge c'_B} \tag{6b}$$

$$\frac{\Upsilon, x : \rho \vdash e \otimes c \rightharpoonup c'}{\Upsilon \vdash e \otimes \exists x : \rho \centerdot c \rightharpoonup \exists x : \rho \centerdot c'} \tag{6c}$$

The first rule is the interesting one. It converts the inequality $e_1 \leq e_2$ by computing bounds for $e_1 - e_2$, where the upper bound is the desired big-M parameter. The lower bound is not needed. This upper bound multiplied by e, which will be of the form $1 - y$, is then added to the appropriate side of the inequality. Conjunctive constraints and existential constraints recurse into their subterms, where in the latter case we add the introduced variable to the context. The other cases are not needed as they will be compiled away beforehand. A finite upper bound on $e_1 - e_2$ must exist. Our software assures this and prints an informative message when a finite bound cannot be computed.

Finally, we define the main big-M compiler. Let $\Upsilon \vdash c \overset{\text{BIGM}}{\longmapsto} c'$ be a judgment converting a disjunctive constraint c to an MILP constraint c' via the big-M method:

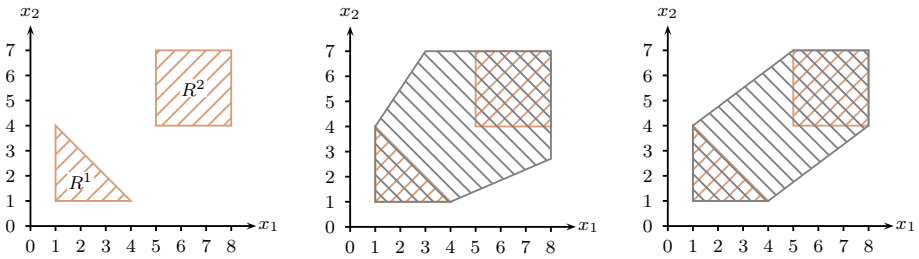
$$\frac{\left\{\Upsilon \vdash c_j \overset{\text{PROP}}{\longmapsto} c'_j\right\}_{j \in \{A,B\}} \qquad \Upsilon \overset{\text{CTXT}}{\longmapsto} \Upsilon' \qquad \left\{\Upsilon' \vdash (1 - y_j) \otimes c'_j \rightharpoonup c''_j\right\}_{j \in \{A,B\}}}{\Upsilon \vdash c_A \vee c_B \overset{\text{BIGM}}{\longmapsto} \exists y_A : \langle 0, 1\rangle \centerdot \exists y_B : \langle 0, 1\rangle \centerdot (y_A + y_B = 1) \wedge (c''_A \wedge c''_B)} \tag{7}$$

First, the disjuncts themselves are compiled using the overall constraint compiler $\overset{\text{PROP}}{\longmapsto}$, which merely recurses on subterms bottom-up, converting any Boolean

expressions and disjunctions to MILP form using the transformations described in this section. Then, we convert the context with the context compiler, which replaces occurrences of `bool` with $\langle 0, 1 \rangle$. This is necessary for the transformation of Boolean expressions and is motivated subsequently. For each disjunct c_j we introduce a corresponding binary variable and rewrite c_j to a big-M form. Finally, the overall result is constructed with appropriate introduction of the y's, the equation forcing the sum of y's to be 1, and the original disjunction $c_A \vee c_B$ replaced with $c''_A \wedge c''_B$.

Indicator Constraint Transformation Recently, the CPLEX system has been extended to natively handle a new constraint form known as an *indicator constraint*. They are of the form $(y = k) \Rightarrow (e_1 \texttt{ op } e_2)$ where y is a binary variable, $k \in \{0, 1\}$, and $\texttt{op} \in \{\leq, =, \geq\}$ A disjunctive constraint can be written as two indicator constraints whose heads are mutually exclusive.

Though we find indicator constraints less natural than disjunction in many cases (e.g. they cannot be nested), CPLEX can handle them in a way that avoids numerical problems when users choose liberally large big-M parameters. Both numerical accuracy and computation times are substantially improved in many problems[3]. To utilize this feature, we have implemented a variant of our big-M transformation which generates indicator constraints from disjunction.



(a) Feasible region. (b) Big-M relaxation. (c) convex-hull relaxation.

Fig. 1: A disjunctive region and two reformulations.

Convex-Hull Transformation We mentioned that the big-M parameters significantly affect the computational efficiency of the resulting program. This is because of a basic step in MILP algorithms involving *relaxation*, a term that refers to allowing integer variables to take any continuous value. The big-M parameter affects the size of the feasible space for these relaxations, and thus computational efficiency. Figure 1b shows this space for the big-M reformulation

[3] Based on comments from the ILOG company's website. We are not aware of any published literature on indicator constraints.

of an example constraint $R^1 \vee R^2$ with the best possible values for the big-M parameters. The convex-hull method is able to produce an even tighter relaxation, shown in Figure 1c. Indeed this is the tightest possible relaxation, the convex-hull of the original disjunctive space, and hence the name of the method. This often leads to even more computationally efficient programs, but is unfortunately substantially more involved. In fact, the number of new variables and equations that must be generated can be so large that it offsets the benefits of its tighter reformulation in some problems. Thus, it is important for MP software to support a breadth of transformations, as there is no single best choice.

The convex-hull method states that (3) can be transformed into the equivalent mixed-integer constraints

$$\begin{array}{ll} A^1\bar{x}^1 \leq b^1 y_1 & y_1 + y_2 = 1 \\ A^2\bar{x}^2 \leq b^2 y_2 & x = \bar{x}^1 + \bar{x}^2 \end{array} \tag{8}$$

where $y_i \in \{0,1\}$. This reformulation is valid only when the inequalities represent bounded regions. The method appears speciously simple when stated on a canonical matrix form. However, in practice models are never written in matrix form, and there is no uniform structure to the equations involved Additionally, various details are omitted such as the need to declare the new variables and add constraints bounding each variable within disjuncts.

The basic idea is to *disaggregate* the disjuncts. In each of the i^{th} disjuncts, vector x has been replaced with a new vector of variables $\bar{x}^i$. This causes the inequalities of each disjunct to be disaggregated, meaning they have no variables in common. For this reason, the $\bar{x}^i$'s are called the disaggregated variables. Finally, the original x is defined to be a sum of the new $\bar{x}^i$'s, and the y's are required to sum to 1. We will have to provide judgments for each of these operations, as well as for the above mentioned omissions in this informal definition.

Since our compiler works on non-DNF forms and allows Booleans, we should more precisely state that it is motivated by the convex-hull method. When the disjuncts are each a conjunction of linear equations and inequalities on the reals, it is Balas' convex-hull method. It is so only for each disjunction separately. When there are multiple disjunctions, i.e. a conjunction of disjunctions, it does not produce the convex-hull overall.

We begin with the main judgment $\Upsilon \vdash c \overset{\text{CVX}}{\longmapsto} c'$, followed by the several auxiliary judgments required. The rule is

$$\frac{\begin{array}{ll} \left\{\Upsilon \vdash c_j \overset{\text{PROP}}{\longmapsto} c'_j\right\}_{j\in\{A,B\}} & \Upsilon \overset{\text{CTXT}}{\longmapsto} \Upsilon' \\ \left\{\Upsilon' \vdash c'_j \multimap_{x_1^j,\ldots,x_m^j} c''_j\right\}_{j\in\{A,B\}} & \left\{y^j \circledast \{\boldsymbol{x}^j/\boldsymbol{x}\}\, c''_j \hookrightarrow c'''_j\right\}_{j\in\{A,B\}} \end{array}}{\Upsilon \vdash c_A \vee c_B \overset{\text{CVX}}{\longmapsto} \left(\begin{array}{l} \exists \boldsymbol{x}^A : \boldsymbol{\rho} \centerdot \exists \boldsymbol{x}^B : \boldsymbol{\rho} \centerdot \exists y^A : \langle 0,1\rangle \centerdot \exists y^B : \langle 0,1\rangle \centerdot \\ \quad (\boldsymbol{x} = \boldsymbol{x}^A + \boldsymbol{x}^B) \wedge (y^A + y^B = 1) \wedge (c'''_A \wedge c'''_B) \end{array}\right)} \tag{9}$$

The notation used assumes the context Υ is $x_1 : \rho_1, \ldots, x_m : \rho_m$. For each x_j, two disaggregated variables x_j^A and x_j^B are created, which must not be free in $c_A \vee c_B$. Also, two binary variables y^A and y^B are created, such that the chosen

names are not free in $c_A \vee c_B$ and are also unique from the x_j^A's and x_j^B's. We have also used vector notation in the meta-language: $\exists \boldsymbol{x} : \boldsymbol{\rho}$ refers to a sequence of existential quantifiers introducing multiple variables each with their own type, $\boldsymbol{x} = \boldsymbol{x}^A + \boldsymbol{x}^B$ refers to the conjunction of equations for each individual x, and $\{\boldsymbol{x}^j/\boldsymbol{x}\}c_j''$ refers to the substitution of a vector of variables $\boldsymbol{x}^j$ for their respective variables in $\boldsymbol{x}$. The constraint output by (9) can roughly be seen to correspond to the matrix reformulation (8).

First, the disjuncts are themselves transformed, producing the MILP constraints c_A' and c_B', and then the context is transformed. Next, bounding constraints are added to each disjunct using $\multimap$, and the disaggregated constraints are created by using $\hookrightarrow$. These themselves require some auxiliary judgments that we define next.

To add constraints bounding a variable, we introduce a judgment that converts a refined type declaration to a constraint. Let $x : \rho \simeq c$ return the bounding information provided by $x : \rho$ in the form of a constraint c. The definition of $\simeq$ is by case on the form of ρ,

$$\begin{array}{rlrl} x:[r_L,r_U] &\simeq r_L \le x \wedge x \le r_U & x:\langle r_L,r_U\rangle &\simeq r_L \le x \wedge x \le r_U \\ x:[r_L,\infty) &\simeq r_L \le x & x:\langle r_L,\infty\rangle &\simeq r_L \le x \\ x:(-\infty,r_U] &\simeq x \le r_U & x:(-\infty,r_U\rangle &\simeq x \le r_U \\ x:\texttt{real} &\simeq \texttt{T} & x:\texttt{int} &\simeq \texttt{T} \end{array} \tag{10}$$

The first rule states that the declaration $x : [r_L, r_U]$ corresponds to specifying bounds with the constraint $r_L \le x \wedge x \le r_U$. There is just a single inequality when the variable is bounded on only one side. The type declaration $x : \texttt{real}$ generates the propositional truth constant $\texttt{T}$, which means this declaration does not constrain the values of x. Definitions for integer types are similar, and the Boolean cases are omitted as they will not be needed.

Let $\Upsilon \vdash_{x_1,\ldots,x_m} c \multimap c'$ be a quaternary judgment adding to c bounding constraints for all the given variables, returning the result as c'. Its definition is

$$\frac{\{x_j : \rho_j \simeq c_j\}_{j=1}^m}{\Upsilon \vdash_{x_1,\ldots,x_m} c \multimap (c_1 \wedge \cdots \wedge c_m \wedge c)} \tag{11}$$

where $\Upsilon(x_j) = \rho_j$ for $j = 1, \ldots, m$.

Finally, let $e \circledast e_1 \hookrightarrow e_2$ be a judgment that multiplies e to the constant part of e_1, producing e_2. For example, $(1+2)*(3+4+(5+6)*x)$ gets converted to $(1+2)*(3*e+4*e+(5+6)*x)$. The judgment $e \circledast c_1 \hookrightarrow c_2$ is the corresponding judgment for constraints, recursing on subterms in a straightforward way. These judgments correspond to the multiplication of the right hand sides of the matrix inequations by binary variables in (8).

Boolean Expressions We convert Boolean expressions to linear inequalities involving only binary variables by first converting them to conjunctive normal form (CNF), then rewriting the clauses—which are in disjunctive literal form (DLF)—as integer constraints in the usual way, and finally lifting Boolean `and` to

constraint-level $\wedge$. For example, (y `and` z) `or not` x becomes (y `or not` x) `and` (z `or not` x) in CNF, which is then converted to the constraint $(y + 1 - x \geq 1) \wedge (z + 1 - x \geq 1)$. The types of the variables are changed from `bool` to $\langle 0, 1 \rangle$ with the refined context compiler $\overset{\text{CTXT}}{\longmapsto}$.

Program Transformation The objective of a MP must be of type `real`, so it is already in MILP form and need not be transformed. The types and constraints are transformed using their respective procedures. Essentially, Boolean expressions and disjunctive constraints are replaced by pure MILP equivalents in a bottom-up fashion.

5 Results

We now present examples from chemical process engineering and operations research that we model using the intuitive Boolean and disjunctive constraints supported by our software. We compare our automated transformations to both manually performed transformations, and an existing automated solution. We find that our automated transformations are comparable to those done by a human expert. We also find that no single transformation always produces the most efficient reformulation, so it is advantageous to have a system such as ours in which the high-level MP can be stated once, and then different solution strategies can be pursued.

Implementation We have implemented our object language as an embedded domain-specific language (EDSL) in OCaml. Once a program is specified in our EDSL, one of the various constraint transformations we have defined can be applied selectively or to the whole program. The transformed program, whether a pure MILP or one enhanced with indicator constraints, can be printed to the industry-standard MPS format or the AMPL modeling language. All source code is freely available from the first author's website.

Performance Metrics We look at the following metrics:

- *Number of continuous variables, number of constraints.* These give a rough picture of the potential computational difficulty of the program. Indexed variables are distinct from each other, e.g. $x_1, \ldots, x_n$ counts as n variables.
- *Number of discrete variables.* This is especially relevant to computational complexity because solvers spend a large portion of their time branching on different possible values of discrete variables.
- *CPU time needed for solving.* This of course is the primary metric of interest. However, the other metrics give a better picture of what the transformations are actually doing. All experiments were run on a machine running Linux 2.6.18 with 8GB of RAM, 4GB of swap space, and eight 2.6GHz Intel Xeon processors with 4MB caches.

5.1 Comparison of Automated Solutions

First we compare our transformations to one of only a few existing automated means of solving MPs that use Booleans and disjunction. We use an example inspired by problems from chemical process engineering.

Consider a simple switched flow process: a tank is being filled by two pumps, α and β, whose flow rates switch depending on the mode the pump is in, which is affected by other requirements of the system. Running each pump incurs different costs in each mode. In addition, the tank is being emptied continuously at a constant rate. There are several constraints: the material level in the tank must remain between the minimum and maximum levels; pump α must not be run longer than a certain length of time to avoid over-heating; and so on. We wish to study how the material level changes over time and to minimize the cost of running the system for $T^{\max}$ time units. The most natural formulation of the problem involves disjunctive constraints and Boolean variables. For instance, we have constraints that govern the transition dynamics of pump α and enforce the definition of "dummy" transitions (where the pump actually does not change mode):

```
(* disjunction over transitions of α *)
conj(I_{-n}, λi →
    (isTrue(YY(α,i)) ∧ ĉ(α,i) = 0.0 ∧ r̂(α,i) = 0.0)
  ∨ (isTrue(Z(α,on,off,i)) ∧ ĉ(α,i) = 0.0 ∧ r̂(α,i) = -R(e,i))
  ∨ (isTrue(Z(α,off,on,i)) ∧ ĉ(α,i) = 50.0 ∧ r̂(α,i) = -R(e,i)
      ∧ R(e,i) ≥ 2.0) )

(* definition of YY, which indicates dummy transitions *)
  conj(I_{-n},λi → isTrue(YY(α,i)) ⇔ Z(α,on,on,i) || Z(α,off,off,i)))
∧ conj(I_{-n},λi → isTrue(YY(β,i)) ⇔ Z(β,hi,hi,i) || Z(β,lo,lo,i)))
```

This code is directly from our EDSL; only operators, literals, and variable names have been replaced with more mathematical typesetting for readability. The `conj` function implements a meta-level indexed conjunction operator. The constraint for the transition dynamics has several cases; one of them is a special case for when a dummy transition occurs. Modeling such logical conditions between disjuncts of real inequations would be unwieldy without Booleans or disjunction. Full details on the example can be found in [4].

To examine computational efficiency, we will take the MP for the switched flow process and reformulate it to MILP form using the different techniques and then solve the resulting MILP programs using ILOG's CPLEX solver—a widely used, efficient solver for, among other things, LP and MILP problems. We compare four transformation strategies:

- Three are our automations of the big-M, convex-hull, and indicator constraint transformations. Only one input specification, coded in our EDSL and compiled with different options, is needed to produce all three.
- The fourth is CPLEX's Concert Technology. CPLEX offers a C++ API to their solver which allows the use of objects and overloaded operators to write

down models in an intuitive manner. Booleans and logical conditions over linear inequalities are automatically transformed into equivalent forms that use indicator constraints. The software is proprietary and their conversion to indicator constraints likely differs from the one we described in Section 4.

We do not compare to other software because either they do not support Boolean and disjunctive constraints or they call out to CPLEX making the comparison redundant. Mosel, another popular MP software, has an extension called Kalis that does support disjunctions, but only over finite domain variables (not reals).

The methods perform largely as expected: tighter formulations are solved faster (Table 1). Indeed, convex-hull is the fastest formulation despite generating the largest number of constraints. As expected, the big-M method uses the same number of binary variables as the indicator constraint transformation, but needs a larger number of constraints because it handles equality constraints as a pair of inequalities, while the indicator constraint transformation handles equalities directly. Curiously, the Concert formulation introduces more binary variables than the convex hull method, more indicator constraints than our indicator constraint transformation, and is the slowest. Overall, we can see that for this example our transformations perform reasonable reformulations that in fact outperform an existing automated transformation provided by a state-of-the-art solver.

Method	#vars (#binary)	#constr. (#IC)	solve time (sec)
flow-Concert	1061 (874)	1080 (718)	36.85
flow-IC	477 (291)	1001 (438)	11.60
flow-BM	477 (291)	1198	3.37
flow-CH	1194 (631)	2747	1.09
pack12-IC	289 (264)	342 (264)	1.83
pack12-BM	289 (264)	342	1.22
pack12-CH	1345 (264)	2718	168.38
pack12-BM-expert	289 (264)	342	1.82
pack12-CH-expert	1345 (264)	1662	149.57
pack21-IC	883 (840)	1071 (840)	24.44
pack21-BM	883 (840)	1071	55.01
pack21-CH	4243 (840)	8631	991.68
pack21-BM-expert	883 (840)	1071	29.56
pack21-CH-expert	4243 (840)	5271	$\geq$ 3600.00

Table 1: Running times and program sizes of MPs compiled via different methods. Transformations: IC = indicator constraint, BM = big-M, CH = convex-hull, Concert = CPLEX Concert, expert = human expert. Examples: flow = switched flow process, pack$N$ = strip packing with N rectangles.

5.2 Comparison of Human Expert vs. Automated Solutions

The convex-hull method can perform poorly on problems with a large number of disjunctions. We investigate this with the strip packing problem. Strip packing involves packing n rectangles without rotation or overlap into a strip of width W that is unbounded to the right while attempting to minimize the length of the strip needed to pack the rectangles. This is a frequently studied problem and we have available reformulations done manually by experts, which allows us to compare our automatically generated programs with expertly generated ones. The constraints in strip packing ensure that the length of the strip extends past the end of each rectangle and that the rectangles do not overlap (i.e. at least one of: are to the left/right of one another or above/below one another):

```
conj(I, λi → length ≥ x(i) + l(i)) ∧
conj(I, λj → conj(1--(j-1), λi →
    x(i) + l(i) ≤ x(j) ∨ x(j) + l(j) ≤ x(i) ∨
    y(i) - h(i) ≥ y(j) ∨ y(j) - h(j) ≥ y(i) ))
```

For our experiments, we implemented the MP form of strip packing with our EDSL and compared it to reformulations manually performed by an expert of both the big-M and convex-hull methods. The manual reformulations were taken from [7], and we used them verbatim, with no modifications. We then ran the reformulations on a medium problem consisting of 12 rectangles and a large problem consisting of 21 rectangles.

The results show that convex-hull is indeed not the optimal solution technique in all scenarios. The number of constraints and variables outweighs any benefits from having a tight formulation per disjunction. Also, we can see that the automatic versions of the big-M and convex-hull transformations are on par with the expertly coded versions. The number of binary variables is equal across all methods because they all introduce one binary variable per disjunct, and there are no Boolean variables in the source program. Many of the numbers are identical between the expertly coded and automated versions, as expected with the simple program structure of strip packing. Also, the expertly coded convex-hull method contains fewer constraints because the expert is able to reason that some constraints are redundant given their bounds, e.g. $0 * y \leq x_i$ is redundant if x_i has been declared to be nonnegative.

In general, it is hard to tell a priori which methods will work well on a given program, so it is useful to have a tool such as ours that enables experimentation without the manual overhead. In fact, anecdotal evidence suggests that once the object language has been properly formalized, adding reformulations is quite easy, so there is a lower barrier to trying new ideas.

6 Related & Future Work

Egon Balas first described the convex-hull method in a technical report [1], which was made available in published form much later [8]. The theory presented there has had significant impact on MILP algorithms. Although Balas acknowledged

that disjunctive constraints are useful for modeling, the focus has been on the insights they provide to more computationally efficient formulations. Thus, those working on MP theory have had little motivation to automate transformations and have not considered the differences arising from programs written in non-matrix forms. Raman and Grossmann popularized this method amongst the chemical processing industry and demonstrated that complex real-world problems could be modeled effectively [3]. They also included the use of Boolean constraints, and provided a method for tying these to disjunctive constraints.

Vecchietti and Grossmann describe an implementation of this alternative formulation with similar goals to this work in a software called LogMIP [9], implemented as an extension of the GAMS language. They support the convex-hull method, but it is not difficult to find examples where the software provides erroneous answers [4], and the semantics of the input language are rather unclear. It is our hope that the theory developed in this work can be employed as a foundation for future development of LogMIP.

There exist numerous transformations for MPs in addition to the big-M and convex-hull methods [7, 10]. Many are related to forms other than disjunctive constraints and so we feel our syntactic formulation can have wider benefits. Nemhauser and Wosley, among others, discuss the importance of *cuts* [2], which our syntactic foundation should be able to support. Hooker discusses the promising idea of employing constraint programming (CP) techniques to solve mathematical programs [11]. One of the challenges for these efforts has been that traditional MP theory does not allow referring to constraints as objects which is essential to CP. Our syntactic formulation immediately provides this. Brand et al. describe a system for exploring alternate linearizations of constraint programs, including the big-M method [12]. Like many other CP techniques, they only deal with finite-domain variables.

We have compared our software to CPLEX[4], which is considered the state-of-the-art MILP solver. In addition, with respect to the language features we are considering, its API is the most expressive. It supports Booleans and disjunctive constraints to the full generality that we do. It also provides a syntactic conversion of these (to indicator constraints) and was thus the most appropriate tool for comparing our transformations to. Note however that CPLEX has numerous other features making it an effective algorithm. Our goal is to supplement those capabilities with operations benefiting from a syntactic perspective.

There are other works that focus specifically on language design. The most widely used are GAMS [13], AMPL [14], Mosel [15], and OPL [16]. Kallrath provides a comprehensive overview [17]. All these support indexing, an essential requirement of any good MP language. It is interesting that although these are the leading languages, they have limited or no support for important features such as Booleans and disjunctive constraints. Although our goal in this work was not to provide a superior object language, we believe our use of formal programming language methods can lead to better languages.

[4] http://www.ilog.com

Acknowledgments We thank Robert Harper (Computer Science, Carnegie Mellon University) for his essential contributions to this work.

References

1. Balas, E.: Disjunctive programming: Properties of the convex hull of feasible points. Technical Report MSRR 348, Carnegie Mellon University (1974)
2. Nemhauser, G.L., Wolsey, L.A.: Integer and combinatorial optimization. Wiley-Interscience series in discrete mathematics and optimization. Wiley, NY (1999)
3. Raman, R., Grossmann, I.E.: Modelling and computational techniques for logic based integer programming. Computers & Chemical Engineering **18**(7) (1994) 563–578
4. Agarwal, A.: Logical Modeling Frameworks for the Optimization of Discrete-Continuous Systems. PhD thesis, Carnegie Mellon University (2006)
5. Potts, P., Edalat, A., Escardo, M.: Semantics of exact real arithmetic. In: LICS '97., 12th Annual IEEE Symp. on Logic in Computer Science, Warsaw (1997) 248–257
6. Nanevski, A., Blelloch, G., Harper, R.: Automatic generation of staged geometric predicates. In: Proceedings of the sixth ACM SIGPLAN International Conference on Functional programming, ICFP 2001. ACM, Florence, Italy (2001) 217–228
7. Sawaya, N.: Reformulations, Relaxations and Cutting Planes for Generalized Disjunctive Programming. PhD thesis, Carnegie Mellon University (2008)
8. Balas, E.: Disjunctive programming: Properties of the convex hull of feasible points. Discrete Applied Mathematics **89**(1-3) (1998) 3–44
9. Vecchietti, A., Grossmann, I.E.: Modeling issues and implementation of language for disjunctive programming. Computers & Chemical Engineering **24**(9–10) (2000) 2143–2155
10. Liberti, L.: Techniques de Reformulation en Programmation Mathématique. L'habilitation à diriger des recherches (hdr), Université Paris IX, Lamsade (2007) Language: English.
11. Hooker, J.N.: Logic-based methods for optimization: combining optimization and constraint satisfaction. Wiley-Interscience series in discrete mathematics and optimization. John Wiley & Sons (2000)
12. Brand, S., Duck, G.J., Puchinger, J., Stuckey, P.J.: Flexible, rule-based constraint model linearisation. In: Tenth International Symposium on Practical Aspects of Declarative Languages. (2008)
13. Bisschop, J., Meeraus, A.: On the development of a general algebraic modeling system in a strategic-planning environment. Mathematical Programming Study **20**(Oct) (1982) 1–29
14. Fourer, R., Gay, D.M., Kernighan, B.W.: A modeling language for mathematical programming. Management Science **36**(5) (1990) 519–554
15. Colombani, Y., Heipcke, T.: Mosel: an extensible environment for modeling and programming solutions. In Jussien, N., Laburthe, F., eds.: 4th Intl. Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'02), Le Croisic, France (2002) 277–290
16. van Hentenryck, P., Lustig, I.: The OPL optimization programming language. MIT Press, Cambridge, Mass. (1999) With contributions by Irvin Lustig, Laurent Michel, and Jean-Francois Puget.
17. Kallrath, J.: Modeling languages in mathematical optimization. Volume 88 of Applied optimization. Kluwer Academic Publishers, Boston (2004)