# Automating Mathematical Program Transformations

Ashish Agarwal [1,*], Ignacio E. Grossmann

*Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA*

**Abstract**

Program transformations are often not automated because their definitions do not specify the many details needed to implement them correctly in code. We define two, the conversion of Boolean expressions into integer inequalities and the convex hull transformation of disjunctive constraints, using a novel style based on set theory. This treats constraints as syntactic objects, allowing mappings between constraint spaces to be defined in a precise way. As a result, our definitions are directly amenable to a computer implementation. Examples of our software's input and output are provided.

*Key words:* Boolean propositions, disjunctive constraints, mathematical induction, set theory

[*] Corresponding author.
*Email addresses:* `ashish.agarwal@yale.edu` (Ashish Agarwal), `grossmann@cmu.edu` (Ignacio E. Grossmann).

[1] Present Address: Department of Genetics, and Department of Molecular Biophysics and Biochemistry, Yale University, New Haven, CT 06520, USA.

## 1 Introduction

Methods for transforming mathematical programs (MPs) are generally not stated in a manner amenable to computer implementation. As a result, there is little support for these methods in existing MP software, restricting their use to experts familiar with the relevant literature. Even for experts, manual application of these techniques is tedious and error-prone.

We believe there are two main reasons preventing more widespread implementation of program transformations. Firstly, most are stated on programs in a canonical form, with matrices, not on the more natural syntax used in practice. Second, matrix forms only support numerical operations, but transformation procedures often require other types of operations. For example, a numerical function cannot formalize variable name generation as needed for transforming disjunctive constraints.

We develop a framework for automating two transformations: disjunctive constraints are converted to mixed-integer constraints, and Boolean propositions into linear integer constraints. Our transformations are applicable to a fairly broad class of programs—including nonlinear terms, nested disjunctions, and Booleans intermixed with disjunctions—but we have not yet attempted to provide a good reformulation in the general case. However, when a disjunction consists purely of linear inequalities on the reals, our method corresponds to the convex hull reformulation (Balas; 1974). Our main result is to define these methods in a novel style based on set theory. We show how these kinds of definitions lead more directly to a computer implementation.

There have been limited efforts to automate transformations even in the most widely used MP software: GAMS (Bisschop and Meeraus; 1982), AMPL (Fourer et al.; 1990), Mosel (Colombani and Heipcke; 2002), and OPL (van Hentenryck and Lustig; 1999). None of these transform Boolean expressions, and only OPL transforms disjunctive constraints, although it occasionally gives incorrect results (see Agarwal (2006) for an example). SIMPL (Aron et al.; 2004) also converts disjunctive constraints but only of a very particular form. LogMIP (Vecchietti and Grossmann; 2000) is an extension to GAMS and converts disjunctions using either the big-M or convex hull (Balas; 1974) methods. However, in the latter case, it assumes bounds on variables when none are provided, thereby returning an incorrect reformulation to the unknowing user. Most importantly, none of these software have been accompanied with a mathematical framework enabling a general study and systematic implementation of program transformations.

First we will briefly review a definition for a set of mathematical programs provided in Agarwal's (2006) dissertation. In contrast to previous definitions, this set contains MPs as written in practice[2] and uniformly integrates a rich variety of constraints. Crucially, it allows treating MPs as syntactic objects that formally encode all information needed for program transformations.

The usual matrix-based definitions do not. For example, Nemhauser and Wolsey (1999, p. 3) define a mixed-integer linear program (MILP) as

$$\max \left\{ cx + hy : Ax + Gy \leq b, x \in \mathbb{Z}_+^n, y \in \mathbb{R}_+^p \right\}$$

The mathematical theory supporting this definition allows using only the coefficient matrices as formal arguments to a function. The variable names $x$ and $y$, operators $+$ and $\leq$, or data types $\mathbb{Z}$ and $\mathbb{R}$ cannot be. However, program transformations are functions operating literally on constraints, such as $f(Ax + Gy \leq b)$. Function $f$ needs to be a mapping between constraint spaces, and these spaces must be first defined.

With these definitions in place, we define binary relations on them. The two main relations we provide convert Boolean expressions into linear integer inequalities, and disjunctive constraints into mixed-integer inequalities. These are combined to provide a transformation on programs with an arbitrary mix of such constraints. We conclude by showing how our definitions lead directly to automation and present some examples of our software's output.

## 2 Review of Inductive Set Definitions

We provide a brief review of induction since it is used in virtually every definition of this paper. Induction is often explained over the natural numbers (e.g. Rosen; 1995) but we require a more general version that allows multiple base cases and allows the inductive step to proceed in multiple directions (Andrews; 2002). We use induction to define sets and set relations, and so it is discussed from this perspective.

A judgement $J$ is a statement declaring that a certain object has some property, i.e. that it lies in some set. For example, we might let "$e$ expr" be a judgement meaning that $e$ is an expression, i.e. that it lies in the set of expressions. The objects satisfying this judgement are defined by a set of inference rules, each of which takes the form

$$\frac{J_1' \quad \cdots \quad J_m'}{J} \tag{1}$$

[2] For simplicity we omit here the very important feature of indices, which is covered in Agarwal (2006).

where $J'_1, \ldots, J'_m$ are the preconditions required for a conclusion $J$. For example, we might choose to define the judgement $e$ expr by the rules

$$\frac{}{k \text{ expr}} \tag{2a}$$

$$\frac{e_1 \text{ expr} \qquad e_2 \text{ expr}}{e_1 + e_2 \text{ expr}} \tag{2b}$$

$$\frac{e_1 \text{ expr} \qquad e_2 \text{ expr}}{e_1 - e_2 \text{ expr}} \tag{2c}$$

where $k$ is any integer. The first rule states that an integer $k$ by itself is an expression without condition. The second states that if $e_1$ and $e_2$ are known to be expressions, then so is "$e_1 + e_2$", and the third rule is similar. The definition is inductive because expressions are constructed from other expressions. An infinite set of expressions has been defined, which contains for example the elements "1", "2", "$1 + 2$", "$(1 + 2) + 1$", etc. Later we will define a richer set of expressions that includes variables and additional operators.

A shorthand is often used when the preconditions are simple enough to allow it. Rules (2) can be written compactly as

$$e ::= k \mid e_1 + e_2 \mid e_1 - e_2 \tag{3}$$

Here it is understood that any use of the symbol "$e$" refers to an object satisfying the judgement $e$ expr. This notation can be read as "an expression $e$ can be of the form $k$, or of the form $e_1 + e_2$, or of the form $e_1 - e_2$". But usually rules have to be written out in their full form (1).

The same technique can be used to define subsets or set relations. For example, let $e$ val mean $e$ is a value, and define this judgement with the single rule

$$\frac{}{k \text{ val}} \tag{4}$$

This states that the subset of expressions of the form "$k$" are what we consider to be values.

Next, we can define a binary relation $\searrow$ on expressions. Let $e_1 \searrow e_2$ mean $e_1$ evaluates to $e_2$, where $e_2$ satisfies $e_2$ val. One rule defining $\searrow$ is

$$\frac{}{k \searrow k} \tag{5a}$$

which states that values evaluate to themselves. Another rule would be

$$\frac{}{1 + 1 \searrow 2} \tag{5b}$$

which states that the expression "$1 + 1$" evaluates to "2". Many more rules are needed to complete this definition; there are techniques for providing these in a finite way but we will not need them here.

More thorough and rigorous introductions to induction are available in many texts, e.g. Andrews (2002); Pierce (2002).

## 3 Set of Mathematical Programs

We now review Agarwal's (2006) definition of a set of MPs, where each element is a syntactical object representing a particular MP. Mathematical programs are comprised of constraints, which are themselves comprised of expressions, and so we need to first define these sets.

The set of expressions is defined by

$$\begin{aligned} e ::= {} & x \mid r \mid \texttt{true} \mid \texttt{false} \\ & \mid -e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \\ & \mid \texttt{not}\, e \mid e_1 \,\texttt{or}\, e_2 \mid e_1 \,\texttt{and}\, e_2 \end{aligned} \tag{6}$$

which contains several more forms than the one of our tutorial example. Firstly, any alphanumeric symbol $x$ is declared to be an expression. Also, we allow numerical constants $r$, as well as the Boolean constants `true` and `false`. The other forms are the familiar arithmetic and Boolean operations.

According to this definition, the expression "$\texttt{not}\, 3.4$" is an expression, which we do not want. Agarwal (2006) defines an important judgement known as a typing judgement that specifies the subset of expressions that are well formed. In particular, the judgement allows checking that an expression is of some type, either real or Boolean. The expression "$\texttt{not}\, 3.4$" is ill formed because it is not of any type.

The set of constraints, which we call propositions in this work, is defined by

$$\begin{aligned} c ::= {} & \texttt{T} \mid \texttt{F} \\ & \mid \texttt{isTrue}\, e \mid e_1 = e_2 \mid e_1 \leq e_2 \\ & \mid c_1 \vee c_2 \mid c_1 \wedge c_2 \\ & \mid \exists x : \rho \,.\, c \end{aligned} \tag{7}$$

Propositions `T` and `F` are propositional truth constants, which are distinct from the Boolean constants `true` and `false`. Propositions of the form "$\texttt{isTrue}\, e$" mark a Boolean expression as a proposition. Next, two expressions $e_1$ and $e_2$ can be used to construct an equation or an inequality. Following that, we define $c_1 \vee c_2$ to be a proposition given that $c_1$ is and that $c_2$ is. There is a distinction between propositional disjunction $\vee$ versus Boolean disjunction `or`, and similarly conjunction. These distinctions are crucial for program transformations. We will see that converting Boolean expressions into linear integer inequalities requires lifting Boolean conjunction `and` into propositional conjunction $\wedge$. Also, Balas's method regards $c_1 \vee c_2$, and is not relevant to $e_1 \,\texttt{or}\, e_2$, a disjunction of an entirely different nature.

Finally, the last form $\exists x : \rho \centerdot c$ is how variables get introduced and is read "there exists $x$ of type $\rho$ such that $c$ holds". Importantly for transformations, the scope of the variable $x$ is local to $c$. When new variables are generated, we need not avoid name clashes with any part of the program except $c$. This notation for introducing variables greatly facilitates the definition of the disjunctive constraint transformation we give later.

The types $\rho$ are defined by

$$\begin{aligned}\rho ::= &\ \langle r_L, r_U\rangle \mid \langle r_L, \infty) \mid (-\infty, r_U\rangle \mid \texttt{real} \\ &\mid [r_L, r_U] \mid [r_L, \infty) \mid (-\infty, r_U] \mid \texttt{int} \\ &\mid \{\texttt{true}\} \mid \{\texttt{false}\} \mid \texttt{bool}\end{aligned} \tag{8}$$

where angle brackets have been used to denote real intervals, and square brackets integer intervals. Essentially variables can be of type `real`, `int`, or `bool`. The interval forms allow bounds to also be specified, and the singleton sets $\{\texttt{true}\}$ and $\{\texttt{false}\}$ are provided for completeness.

Given expressions $e$, propositions $c$, and types $\rho$, we can now define the syntax for a mathematical program. An MP is of only a single form given by

$$p ::= \delta_{x_1:\rho_1,\ldots,x_m:\rho_m} \{e \mid c\} \tag{9}$$

where $\delta ::= \texttt{min} \mid \texttt{max}$. The notation means that an objective $e$ will be minimized (or maximized) over some variables $x_1, \ldots, x_m$ subject to proposition $c$. A program is well formed only if the objective is of type `real` and the proposition is also well formed. This judgement is denoted $p$ MP and has been defined in Agarwal (2006); we assume subsequently that we are working only with well formed programs.

Compare (9) with the usual matrix style definition of an MP. For example, Raman and Grossmann's (1994) definition of what they call a generalized disjunctive program (GDP) is

$$\begin{aligned}
&\min Z = \sum_k c_k + d^{\mathrm{T}} x \\
&\text{s.t. } Bx \le b \\
&\qquad \bigvee_{i \in D_k} \begin{bmatrix} Y_{ik} \\ A_{ik} x \le a_{ik} \end{bmatrix}, \quad k \in SD \\
&\qquad \Omega(Y) = true \\
&\qquad x \in \mathbb{R}^n, \quad c \in \mathbb{R}^m, \quad Y \in \{\text{true}, \text{false}\}^m
\end{aligned} \tag{10}$$

This includes disjunctive constraints and Boolean expressions, so includes programs similar to those included by definition (9). However, (9) is a formal set definition, and can be used as the domain and codomain of a function. Also, the use of induction nicely includes additional constraint forms useful in practice, e.g. nested disjunctions. Definition (10) does not, and consequently a transformation defined on this form may be difficult to apply to programs as written in practice.

We define one other construct needed subsequently. A context

$$\Upsilon ::= \emptyset \mid \Upsilon, x : \rho \tag{11}$$

is either an empty list or a context appended with an item "$x : \rho$". In other words, a context is a list of variables with their declared types. It is assumed that variable names are unique within any context. Let $\Upsilon(x) = \rho$ mean, in context $\Upsilon$, variable $x$ is declared to be of type $\rho$.

Finally, there are two basic operations needed on the set elements we have defined. It is necessary to know which variables occur in any construct. We let $FV(e)$ and $FV(c)$ refer respectively to the free variables of an expression and proposition. For example, $FV(x + (1 - y)) = \{x, y\}$. Bound variables are not included. For example, $FV(\exists x : \rho \,.\, x + (1 - y)) = \{y\}$ because $x$ is not free in the overall proposition. Another basic operation allows replacing variables with another expression. Let $\{e/x\}\, e'$ mean that we substitute $e$ for $x$ in $e'$. For example $\{3/x\}(x + (1 - y))$ would return $3 + (1 - y)$. Similarly, $\{e/x\}\, c$ substitutes $e$ for $x$ in proposition $c$.

## 4 Transforming Syntactic Constructs

The class of programs covered by $p$ include disjunctive propositions and Booleans. However, most solvers accommodate only mixed-integer programming (MIP) constraints, which do not allow either of these forms. Formally, we define the MIP types $\rho^{\text{MIP}}$ to be as above but exclude $\{\texttt{true}\}$, $\{\texttt{false}\}$, and `bool`; MIP expressions $e^{\text{MIP}}$ disallow Boolean constants and Boolean operators, and MIP propositions $c^{\text{MIP}}$ disallow $\texttt{isTrue}\ e$ and $c_1 \vee c_2$. Essentially, the only constraints included in $c^{\text{MIP}}$ are conjunctions of equations and inequalities. Finally, let $p^{\text{MIP}}$ refer to programs containing only MIP types, expressions, and propositions.

Our goal now is to provide a method for transforming programs in the full syntax $p$ to those in the restricted syntax $p^{\text{MIP}}$. We know this can be done, under a mild condition, because of previous results by Balas (1974), Raman and Grossmann (1994), and others. However, the transformation procedures in these works have been stated in English. We now provide corresponding mathematical definitions.

Transforming a program requires transforming the types, expressions, and propositions that comprise it. We provide a transformation procedure for each construct in turn.

## *4.1 Type Transformation*

Let $\rho \overset{\text{RTYPE}}{\longmapsto} \rho^{\text{MIP}}$ be a binary relation on types meaning the general type $\rho$ can be transformed to the MIP type $\rho^{\text{MIP}}$. The definition of this judgement is given by the rules

$$\frac{}{\langle r_L, r_U\rangle \overset{\text{RTYPE}}{\longmapsto} \langle r_L, r_U\rangle} \tag{12a}$$

$$\frac{}{\langle r_L, \infty) \overset{\text{RTYPE}}{\longmapsto} \langle r_L, \infty)} \tag{12b}$$

$$\frac{}{(-\infty, r_U\rangle \overset{\text{RTYPE}}{\longmapsto} (-\infty, r_U\rangle} \tag{12c}$$

$$\frac{}{\texttt{real} \overset{\text{RTYPE}}{\longmapsto} \texttt{real}} \tag{12d}$$

$$\frac{}{[r_L, r_U] \overset{\text{RTYPE}}{\longmapsto} [r_L, r_U]} \tag{12e}$$

$$\frac{}{[r_L, \infty) \overset{\text{RTYPE}}{\longmapsto} [r_L, \infty)} \tag{12f}$$

$$\frac{}{(-\infty, r_U] \overset{\text{RTYPE}}{\longmapsto} (-\infty, r_U]} \tag{12g}$$

$$\frac{}{\texttt{int} \overset{\text{RTYPE}}{\longmapsto} \texttt{int}} \tag{12h}$$

$$\frac{}{\{\texttt{true}\} \overset{\text{RTYPE}}{\longmapsto} [1, 1]} \tag{12i}$$

$$\frac{}{\{\texttt{false}\} \overset{\text{RTYPE}}{\longmapsto} [0, 0]} \tag{12j}$$

$$\frac{}{\texttt{bool} \overset{\text{RTYPE}}{\longmapsto} [0, 1]} \tag{12k}$$

The final rule declares that the type `bool` will be converted into the type $[0, 1]$, which is the set containing the two integers 0 and 1. We convert $\{\texttt{false}\}$ into $[0, 0]$, and $\{\texttt{true}\}$ into $[1, 1]$. The other types remain unchanged because they are already MIP types.

Let $\Upsilon \overset{\text{CTXT}}{\longmapsto} \Upsilon^{\text{MIP}}$ be a context transformation. Its definition is

$$\frac{}{\emptyset \overset{\text{CTXT}}{\longmapsto} \emptyset} \tag{13a}$$

$$\frac{\Upsilon \overset{\text{CTXT}}{\longmapsto} \Upsilon^{\text{MIP}} \qquad \rho \overset{\text{RTYPE}}{\longmapsto} \rho^{\text{MIP}}}{\Upsilon, x : \rho \overset{\text{CTXT}}{\longmapsto} \Upsilon^{\text{MIP}}, x : \rho^{\text{MIP}}} \tag{13b}$$

This simply transforms each of the declared types in a context.

### *4.2 Expression Transformation*

Only Boolean expressions need to be transformed; numerical ones are already in MIP form. Boolean expressions are first converted into conjunctive normal form. Let $e$ CNF mean $e$ is a conjunctive normal form, and let $e_1 \curvearrowright e_2$ be a relation converting any Boolean expression $e_1$ into $e_2$ such that $e_2$ CNF. These judgements have been defined by Agarwal (2006). Also, we will partition CNF expressions into disjunctive literal forms DLF and conjunctive forms CONJ. Different methods for transforming DLF versus CONJ expressions are required (if we wish to generate linear inequalities).

Let $e \overset{\text{DLF}}{\longmapsto} e^{\text{MIP}}$ be a judgement converting DLF expressions into integer MIP expressions. Its definition is motivated by the following decisions:

- LITERAL expressions taking the value `false` and `true` correspond to numeric expressions taking the value 0 and 1, respectively
- DLF expressions taking the value `false` and `true` correspond to numeric expressions taking the value 0 and $\geq 1$, respectively.

$$\frac{}{x \overset{\text{DLF}}{\longmapsto} x} \tag{14a}$$

$$\frac{}{\texttt{true} \overset{\text{DLF}}{\longmapsto} 1} \tag{14b}$$

$$\frac{}{\texttt{false} \overset{\text{DLF}}{\longmapsto} 0} \tag{14c}$$

$$\frac{e \overset{\text{DLF}}{\longmapsto} e'}{\texttt{not}\, e \overset{\text{DLF}}{\longmapsto} 1 - e'} \tag{14d}$$

$$\frac{e_1 \overset{\text{DLF}}{\longmapsto} e_1' \quad e_2 \overset{\text{DLF}}{\longmapsto} e_2'}{e_1\, \texttt{or}\, e_2 \overset{\text{DLF}}{\longmapsto} e_1' + e_2'} \tag{14e}$$

A variable $x$ is left as is. But since its declared type will also be compiled, it will take a $[0, 1]$ value instead of a `bool` value. The Boolean constants `true` and `false` are transformed to the integer constants 1 and 0. The expression $\texttt{not}\, e$ is converted by first converting $e$ to $e'$, and the result is $1 - e'$. The `or` is converted to a $+$ after first converting its arguments. Lemmas 5.3 and 5.4 of Agarwal (2006) prove that this definition adheres to the decisions above.

As an example, let us convert $x\, \texttt{or}\, (\texttt{not}\, y)$. The above rules define how this can be done mechanically. The expression is an `or` expression, so the last rule applies. It requires that we first convert $x$, which returns $x$, and convert $\texttt{not}\, y$, which returns $1 - y$. Then the last rule says the overall answer is $x + (1 - y)$.

CNF expressions of the form $e_1\, \texttt{and}\, e_2$, which we call CONJ, are transformed directly into propositions. Let $e \overset{\text{CONJ}}{\longmapsto} c^{\text{MIP}}$ be the judgement doing so. Its definition is by the single rule

$$\frac{\left\{\emptyset \vdash \texttt{isTrue}\ e_j \overset{\text{PROP}}{\longmapsto} c_j\right\}_{j=1}^{2}}{e_1\, \texttt{and}\, e_2 \overset{\text{CONJ}}{\longmapsto} c_1 \wedge c_2} \tag{15}$$

All that is done is to replace Boolean conjunction `and` with propositional conjunction $\wedge$. This requires first converting $e_1$, $e_2$ to propositions $c_1$, $c_2$, which is done by marking the expressions to be propositions and then using the proposition transformation defined next. An alternative would have been to convert `and` expressions to multiplication expressions, analogously to how `or` was converted to $+$, but that would generate nonlinear inequalities. The separate transformations we provide for DLF versus CONJ expressions produce linear inequalities.

## *4.3 Proposition Transformation*

Let $\Upsilon \vdash c \overset{\text{PROP}}{\longmapsto} c^{\text{MIP}}$ mean within context $\Upsilon$, proposition $c$ can be converted to the MIP proposition $c^{\text{MIP}}$. Unlike expressions, transforming propositions requires knowledge of the variables' types (only because of disjunctive constraints). The definition is given by the rules

$$\frac{}{\Upsilon \vdash \mathtt{T} \overset{\text{PROP}}{\longmapsto} \mathtt{T}} \tag{16a}$$

$$\frac{}{\Upsilon \vdash \mathtt{F} \overset{\text{PROP}}{\longmapsto} \mathtt{F}} \tag{16b}$$

$$\frac{e \curvearrowright e' \quad e'\ \text{DLF} \quad e' \overset{\text{DLF}}{\longmapsto} e''}{\Upsilon \vdash \mathtt{isTrue}\ e \overset{\text{PROP}}{\longmapsto} e'' \geq 1} \tag{16c}$$

$$\frac{e \curvearrowright e' \quad e'\ \text{CONJ} \quad e' \overset{\text{CONJ}}{\longmapsto} c'}{\Upsilon \vdash \mathtt{isTrue}\ e \overset{\text{PROP}}{\longmapsto} c'} \tag{16d}$$

$$\frac{}{\Upsilon \vdash e_1 = e_2 \overset{\text{PROP}}{\longmapsto} e_1 = e_2} \tag{16e}$$

$$\frac{}{\Upsilon \vdash e_1 \leq e_2 \overset{\text{PROP}}{\longmapsto} e_1 \leq e_2} \tag{16f}$$

$$\frac{\Upsilon \vdash c_1 \vee c_2 \overset{\text{DISJ}}{\longmapsto} c'}{\Upsilon \vdash c_1 \vee c_2 \overset{\text{PROP}}{\longmapsto} c'} \tag{16g}$$

$$\frac{\Upsilon \vdash c_1 \overset{\text{PROP}}{\longmapsto} c_1' \quad \Upsilon \vdash c_2 \overset{\text{PROP}}{\longmapsto} c_2'}{\Upsilon \vdash c_1 \wedge c_2 \overset{\text{PROP}}{\longmapsto} c_1' \wedge c_2'} \tag{16h}$$

$$\frac{\rho \overset{\text{RTYPE}}{\longmapsto} \rho' \quad \Upsilon, x : \rho' \vdash c \overset{\text{PROP}}{\longmapsto} c'}{\Upsilon \vdash \exists x : \rho \,.\, c \overset{\text{PROP}}{\longmapsto} \exists x : \rho' \,.\, c'} \tag{16i}$$

Boolean propositions are first converted into CNF. A CNF expression will be either DLF or CONJ; see Theorem 8.5 of Agarwal (2006). The first rule handles the DLF case. Truth corresponds to a positive integer value. So the converted expression is required to be greater than or equal to 1. The CONJ case calls $\overset{\text{CONJ}}{\longmapsto}$, which produces a proposition directly.

Inequalities and equations are already in MIP form, so no work is required to convert them. Compilation of a disjunctive proposition is sufficiently complex to justify packaging it into a separate judgement $\xmapsto{\text{DISJ}}$, discussed next. Conjunctive propositions simply recurse into their sub-propositions. Similarly for existential propositions, but the introduced variable must be added to the context.

## *4.4 Disjunctive Proposition Transformation*

Our disjunctive proposition compiler is motivated by the convex hull method (Balas; 1974; Raman and Grossmann; 1994). When the disjuncts are each a conjunction of linear equations and inequalities on the reals, it is the 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.

Consider the disjunctive constraint

$$\left[A^1 x \le b^1\right] \vee \left[A^2 x \le b^2\right], \tag{17}$$

where $A^i$ is an $m \times n$ coefficient matrix, $x$ is an $n \times 1$ vector of real variables, and $b^i$ is an $m \times 1$ vector of constants. Using the convex hull method, this disjunction can be transformed into the mixed-integer constraints

$$A^1 \bar{x}^1 \le b^1 \lambda_1 \tag{18a}$$
$$A^2 \bar{x}^2 \le b^2 \lambda_2 \tag{18b}$$
$$\lambda_1 + \lambda_2 = 1 \tag{18c}$$
$$x = \bar{x}^1 + \bar{x}^2 \tag{18d}$$

where $\lambda_i \in \{0, 1\}$. In each of the $i^{\text{th}}$ 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. The idea is that $i^{\text{th}}$ disjunct gets enforced only if $\lambda_i = 1$.

We can see that several operations are involved in the convex hull method. Constants are multiplied by binary variables. New disaggregated variables must be created. These must replace the original variables they correspond to in each disjunct. Also, when inequalities do not represent bounded regions, constraints corresponding to known bounds on a variable must be inserted into the disjuncts. Finally, equations relating the original and disaggregated variables must be produced. Our goal is to define this transformation in a formal way, as a mapping on propositions of the form $c_1 \vee c_2$. We first define several auxiliary judgements, which will be employed in the overall transformation.

A type declaration restricts the values a variable can take. The declaration $x : \rho$ implies a constraint on $x$, and it will be necessary to extract this information explicitly. Let $x : \rho \backsimeq c$ associate a declaration $x : \rho$ with a proposition $c$. The definition of $\backsimeq$ is inductive on $\rho$,

$$\frac{}{x : \langle r_L, r_U \rangle \backsimeq r_L \leq x \wedge x \leq r_U} \tag{19a}$$

$$\frac{}{x : \langle r_L, \infty) \backsimeq r_L \leq x} \tag{19b}$$

$$\frac{}{x : (-\infty, r_U \rangle \backsimeq x \leq r_U} \tag{19c}$$

$$\frac{}{x : \mathtt{real} \backsimeq \mathtt{T}} \tag{19d}$$

$$\frac{}{x : [r_L, r_U] \backsimeq r_L \leq x \wedge x \leq r_U} \tag{19e}$$

$$\frac{}{x : [r_L, \infty) \backsimeq r_L \leq x} \tag{19f}$$

$$\frac{}{x : (-\infty, r_U] \backsimeq x \leq r_U} \tag{19g}$$

$$\frac{}{x : \mathtt{int} \backsimeq \mathtt{T}} \tag{19h}$$

$$\frac{}{x : \{\mathtt{true}\} \backsimeq \mathtt{isTrue}\ x} \tag{19i}$$

$$\frac{}{x : \{\mathtt{false}\} \backsimeq \mathtt{isTrue}\ (\mathtt{not}\ x)} \tag{19j}$$

$$\frac{}{x : \mathtt{bool} \backsimeq \mathtt{T}} \tag{19k}$$

The first rule states that the declaration $x : \langle r_L, r_U \rangle$ is equivalent to stating the proposition $r_L \leq x \wedge x \leq r_U$, and all other rules are similar. Our theory distinguishes between the concept of variable bounds and a constraint that happens to contain the same information. Variables' bounds declarations are not constraints, but they do have a correspondence given by $\backsimeq$.

Let $\Upsilon \vdash c \multimap c'$ be a judgement adding to $c$ bounding propositions for all variables free in $c$, returning the result as $c'$. Its definition is

$$\frac{\{x_j : \rho_j \backsimeq c_j\}_{j=1}^{m}}{\Upsilon \vdash c \multimap (c_1 \wedge \cdots \wedge c_m \wedge c)} \tag{20}$$

where $FV(c) = \{x_1, \ldots, x_m\}$ and $\Upsilon(x_j) = \rho_j$ for $j = 1, \ldots, m$.

Let $e \circledast e_1 \hookrightarrow e_2$ be a judgement that multiplies $e$ to all constant terms in $e_1$, producing $e_2$. Both $e$ and $e_1$ must be numeric expressions. The definition is inductive on $e_1$,

$$\frac{}{e \circledast x \hookrightarrow x} \tag{21a}$$

$$\frac{}{e \circledast r \hookrightarrow e * r} \tag{21b}$$

$$\frac{e \circledast e_1 \hookrightarrow e_2}{e \circledast -e_1 \hookrightarrow -e_2} \tag{21c}$$

$$\frac{e \circledast e_1 \hookrightarrow e_1' \quad e \circledast e_2 \hookrightarrow e_2'}{e \circledast (e_1 \,\mathsf{op}\, e_2) \hookrightarrow (e_1' \,\mathsf{op}\, e_2')} \text{ for } \mathsf{op} \in \{+,-\} \tag{21d}$$

$$\frac{}{e \circledast (e_1 * e_2) \hookrightarrow e * (e_1 * e_2)} \text{ if } FV\,(e_1 * e_2) = \emptyset \tag{21e}$$

$$\frac{e \circledast e_2 \hookrightarrow e_2'}{e \circledast (e_1 * e_2) \hookrightarrow (e_1 * e_2')} \text{ if } FV\,(e_1) \neq \emptyset \tag{21f}$$

$$\frac{e \circledast e_1 \hookrightarrow e_1'}{e \circledast (e_1 * e_2) \hookrightarrow (e_1' * e_2)} \text{ if } FV\,(e_2) \neq \emptyset \tag{21g}$$

The result of $e \circledast x$ is $x$. Since $x$ is not a constant, it does not get multiplied by the given expression. $e \circledast r$ gives $e * r$ since $r$ is a constant. For negation, addition, and subtraction expressions, the procedure simply recurses into the sub-expressions. The result of $e \circledast (e_1 * e_2)$ depends on whether $e_1 * e_2$ has any free variables. If it does not, $e_1 * e_2$ is a constant term and it is multiplied by $e$. If it does, $e_1$ and $e_2$ are each considered separately to handle nested terms.

Let $e \circledast c_1 \hookrightarrow c_2$ be an analogous judgement for a proposition. We omit the definition; it simply employs $e \circledast e_1 \hookrightarrow e_2$ on all nested expressions.

Finally, let $\Upsilon \vdash (c_A \vee c_B) \overset{\text{DISJ}}{\longmapsto} c^{\text{MIP}}$ be a disjunctive proposition transformation. The definition is by the single rule

$$\frac{\begin{array}{c}\left\{\Upsilon \vdash c_j \overset{\text{PROP}}{\longmapsto} c_j'\right\}_{j\in\{A,B\}} \qquad \Upsilon \overset{\text{CTXT}}{\longmapsto} \Upsilon' \\ \left\{\Upsilon' \vdash c_j' \multimap c_j''\right\}_{j\in\{A,B\}} \qquad \left\{y^j \circledast \{\vec{x}^j/\vec{x}\}\, c_j'' \hookrightarrow c_j'''\right\}_{j\in\{A,B\}}\end{array}}{\Upsilon \vdash (c_A \vee c_B) \overset{\text{DISJ}}{\longmapsto} \left(\begin{array}{r}\exists \vec{x}^A : \vec{\rho}\,\textbf{.}\, \exists \vec{x}^B : \vec{\rho}\,\textbf{.}\, \exists y^A : [0,1]\,\textbf{.}\, \exists y^B : [0,1]\,\textbf{.} \\ \left(\vec{x} = \vec{x}^A + \vec{x}^B\right) \wedge \left(y^A + y^B = 1\right) \wedge (c_A''' \wedge c_B''')\end{array}\right)} \tag{22}$$

The notation used assumes the context $\Upsilon$ 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, but these 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.

In the first line of the preconditions, the disjuncts are themselves transformed, producing the MIP propositions $c'_A$ and $c'_B$, and the context is transformed. In the second line, bounding constraints are added to each disjunct. Then, each of the $j^{\text{th}}$ disjuncts is disaggregated by performing the substitution $\{\vec{x}^j/\vec{x}\}\, c''_j$. Finally, constants are multiplied by the binary $y^j$.

The results of these operations are used to produce the result proposition. The disaggregated variables are related to the original by the equation $\vec{x} = \vec{x}^A + \vec{x}^B$, which is an abbreviation for the conjunction of equations $x_j = x_j^A + x_j^B$ for $j = 1, \ldots, m$. The binary variables must sum to 1. Finally, the disjunctive proposition $c_A \vee c_B$ is replaced with the conjunctive proposition $c'''_A \wedge c'''_B$.

The disjunctive transformation $\Upsilon \vdash (c_A \vee c_B) \overset{\text{DISJ}}{\longmapsto} c^{\text{MIP}}$ is valid only when all variables occurring within the disjunction have bounds specified in $\Upsilon$. Our software checks for this precondition and warns the user if it is not satisfied. This is sufficient but not necessary. Corollary 2.1.1 of Balas (1974) requires that the region represented by each disjunct be bounded. Our method for including variable bounds explicitly in each disjunct is just one way to satisfy this.

### *4.5 Program Transformation*

Transforming a program is now straightforward. Let $p \overset{\text{PROG}}{\longmapsto} p^{\text{MIP}}$ represent a program transformation. It is defined only for well formed programs. The definition is

$$\frac{\left\{\rho_j \overset{\text{RTYPE}}{\longmapsto} \rho'_j\right\}_{j=1}^{m} \qquad x_1 : \rho_1, \ldots, x_m : \rho_m \vdash c \overset{\text{PROP}}{\longmapsto} c'}{\delta_{x_1:\rho_1,\ldots,x_m:\rho_m}\, \{e \mid c\} \overset{\text{PROG}}{\longmapsto} \delta_{x_1:\rho'_1,\ldots,x_m:\rho'_m}\, \{e \mid c'\}} \tag{23}$$

Since the objective $e$ must be of type `real`, it is already in MIP form and need not be transformed. The types and propositions are transformed using their respective procedures.

## 5 Software Implementation

The mathematical definitions we have provided support automation because they are more precise. For example, the disjunctive transformation (22) includes every detail of the convex hull method: variable name generation, insertion of bounding constraints, multiplication of constants by the new binary variables, etc. Transforming these definitions into code is now less challenging.

In object oriented programming languages such as C++ and Java, the inductively defined sets can be coded by using inheritance. For example, the set of expressions can be coded by declaring an abstract base class called `Expr`. Then, a derived class is defined for each of the specific forms, e.g. `PlusExpr` whose members are `Expr e1` and `Expr e2`. The transformation procedures have one rule for each of these forms. So one can simply declare a virtual method called `transform` and provide the specific implementation for this in each of the derived classes.

Actually, functional programming languages are a superior choice for this work. The programming language ML (Harper et al.; 1989) has explicit support for inductive definitions. The "|" notation used in several of our definitions is directly supported and case constructs allow defining rules for each syntactic form. Our software is implemented in ML, and the ML code closely follows our mathematical definitions.

Minor variations do exist. The syntactic objects we have defined include mathematical symbols, as where parsers usually work only with ASCII text. For example, the set of mathematical programs according to definition (9) includes

$$\texttt{min}_{x:\texttt{real},w:\texttt{real}}\,\{x + w \mid x \le w \vee x \ge w + 4.0\}$$

but our software requires the form shown in Example 2. The $x$ and $w$ variables are declared prior to the `min` to avoid subscripting, and the $\vee$ symbol is written `disj`. Other differences are $\exists$ is written `exists`, "|" is written `subject_to`, $\wedge$ is written as a plain comma "`,`", and the expression `e1 implies e2` is allowed but immediately interpreted as `not e1 or e2`. These are relatively minor details; our software's examples should be recognized as directly corresponding to the mathematical definitions given in this work.

We give two examples: the transformation of a Boolean expression and a disjunctive proposition. These are small examples due to space limitations; a larger example of a continuous process with discrete switching is provided in Agarwal (2006).

**Example 1** *Consider the program*

```
min 0.0 subject_to

exists y1:bool . exists y2:bool . exists y3:bool .
  isTrue ((y1 and y2) implies y3) and y1
```

*A dummy objective has been chosen because we are concerned only with the Boolean expression in this example. The transformation procedure* $p \overset{\text{PROG}}{\longmapsto} p^{\text{MIP}}$ *is applied, and the returned program is*

```
min 0.0 subject_to

exists y1:[0,1] . exists y2:[0,1] . exists y3:[0,1] .
```

```
  ((1 - y1) + (1 - y2)) + y3 >= 1,
  y1 >= 1
```

*Each Boolean variable has been converted into a* $[0, 1]$ *variable. The conjunctive normal form is derived internally, giving*

```
((not y1 or not y2) or y3) and y1
```

*which is a* CONJ *expression. Thus,* $e \overset{\text{CONJ}}{\longmapsto} c^{\text{MIP}}$ *gets applied to produce the proposition shown.*

**Example 2** *Now consider a disjunctive proposition in which a variable* `x` *has to be less than or greater than some function of* `w`*,*

```
var x:real
var w:real

min x + w subject_to
(x <= w) disj (x >= w + 4.0)
```

*The program fails the precondition required for disjunctive propositions. The following error messages are printed,*

```
ERROR: variable in disjunct must be bounded
  variable: w
  is of unbounded type: real

ERROR: variable in disjunct must be bounded
  variable: x
  is of unbounded type: real
```

*Variables* `x` *and* `w` *must have bounds explicitly declared. These must be obtained from an understanding of the physical system. We change the declaration* `x:real` *to* `x:<10.0, 100.0>`*, and* `w:real` *is changed to* `w:<2.0, 50.0>`*.*

*Now, the precondition passes, and the software performs the transformation producing the pure mixed-integer program*[3]

```
var x:<10.0, 100.0>
var w:<2.0, 50.0>

min x + w subject_to

exists y1:[0, 1]
```

[3] Due to formatting reasons, the output of our software is not easily legible. The ouput we are showing has been modified in two minor ways: unnecessary parentheses have been deleted, and indenting has been adjusted.

```
 7 exists y2:[0, 1]
 8 exists x1:<10.0, 100.0>
 9 exists x2:<10.0, 100.0>
10 exists w1:<2.0, 50.0>
11 exists w2:<2.0, 50.0>
12   w = w1 + w2,
13   x = x1 + x2,
14   y1 + y2 = 1,
15
16   10.0 * y1 <= x1,
17   x1 <= 100.0 * y1,
18   2.0 * y1 <= w1,
19   w1 <= 50.0 * y1,
20   x1 <= w1,
21
22   10.0 * y2 <= x2,
23   x2 <= 100.0 * y2,
24   2.0 * y2 <= w2,
25   w2 <= 50.0 * y2,
26   x2 >= w2 + 4.0 * y2
```

*Several new variables are generated. Since these are not used in the objective, they are introduced locally with existential quantifiers. Lines 12–13 relate the new disaggregated variables to the original $x$ and $w$. On line 14, the sum of the binary variables is required to be equal to 1. Lines 16–20 represent the disaggregation of the first disjunct and lines 22–26 of the second.*

## 6 Conclusions

Our main goal was to automate program transformations. As we discussed, even transformations of well known importance have not been supported in any software because they were not defined on the syntax in which programs are written in practice and because many aspects of the transformations were explained only conceptually. Definitions that formally encode every detail of a transformation first require a definition of programs that itself encodes all features of a program, not just the numerical aspects. The inductive set definitions we provided do this by treating programs as syntactic objects. Given this, we were able to define a precise mapping between programs $p \overset{\text{PROG}}{\longmapsto} p^{\text{MIP}}$. We discussed how our definition of this transformation can be implemented in a programming language, thus automating it. The automation is a direct consequence of the fact that our set theoretic definitions are more precise than those previously provided.

**Acknowledgements**

We thank Robert Harper, Department of Computer Science, Carnegie Mellon University, for his essential contributions to this work. His expertise was fundamental to producing the results we have presented.

## References

Agarwal, A. (2006). *Logical Modeling Frameworks for the Optimization of Discrete-Continuous Systems*, PhD thesis, Carnegie Mellon University.

Andrews, P. B. (2002). *An introduction to mathematical logic and type theory: to truth through proof*, Vol. 27 of *Applied logic series*, 2nd edn, Kluwer Academic Publishers.

Aron, I., Hooker, J. N. and Yunes, T. H. (2004). SIMPL: A system for integrating optimization techniques, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Vol. 3011 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 21–36.

Balas, E. (1974). Disjunctive programming: Properties of the convex hull of feasible points, *Technical Report MSRR 348*, Carnegie Mellon University. Published version available as Balas (1998).

Balas, E. (1998). Disjunctive programming: Properties of the convex hull of feasible points, *Discrete Applied Mathematics* **89**(1-3): 3–44. Published version of Balas (1974).

Bisschop, J. and Meeraus, A. (1982). On the development of a general algebraic modeling system in a strategic-planning environment, *Mathematical Programming Study* **20**(Oct): 1–29.

Colombani, Y. and Heipcke, T. (2002). Mosel: an extensible environment for modeling and programming solutions, *in* N. Jussien and F. Laburthe (eds), *4th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'02)*, Le Croisic, France, pp. 277–290.

Fourer, R., Gay, D. M. and Kernighan, B. W. (1990). A modeling language for mathematical programming, *Management Science* **36**(5): 519–554.

Harper, R., Milner, R. and Tofte, M. (1989). *The definition of Standard ML: Version 3*, LFCS report series, Laboratory for Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh, Edinburgh, Scotland.

Nemhauser, G. L. and Wolsey, L. A. (1999). *Integer and combinatorial optimization*, Wiley-Interscience series in discrete mathematics and optimization, Wiley, New York.

Pierce, B. C. (2002). *Types and programming languages*, MIT Press, Cambridge, Mass.

Raman, R. and Grossmann, I. E. (1994). Modelling and computational techniques for logic based integer programming, *Computers & Chemical Engineering* **18**(7): 563–578.

Rosen, K. H. (1995). *Discrete mathematics and its applications*, 3rd edn, McGraw-Hill, New York.

van Hentenryck, P. and Lustig, I. (1999). *The OPL optimization programming language*, MIT Press, Cambridge, Mass. With contributions by Irvin Lustig, Laurent Michel, and Jean-Francois Puget.

Vecchietti, A. and Grossmann, I. E. (2000). Modeling issues and implementation of language for disjunctive programming, *Computers & Chemical Engineering* **24**(9–10): 2143–2155.