Reducing LP Models to Normal Forms

John Starks

May 15, 2007

Abstract

Agarwal (2006) defines a framework and general language for optimization problems, and a series of transformations to reduce that language’s complexity. However, even this reduced form cannot be used by most linear programming solvers, limiting the utility of the language. We complete this missing link by introducing a variety of additional transformations. As a result, we are able to produce linear programming and mixed-integer linear programming programs that can be computed with standard solvers.

1 Introduction

There are a variety of linear programming front-ends available today, including AMPL, Mosel, and OPL. However, none of these front-ends have a well-defined, formally defined language as input, meaning transformations are often poorly-defined and confusing. To remedy this, Agarwal (2006) defines a general language for optimization problems that includes conjunction, disjunction, Boolean logic, index sets, lambdas, quantifiers, and a variety of other features not commonly seen in other front-ends. He further defines a variety of reductions to the language, in particular the removal of Boolean variables, disjunctions, existential quantifiers and lambdas, among others. The result is a simple, well-defined language for mixed integer programs (MIP), both linear and nonlinear.

However, there are still features of this language that are not understood by most solvers, and even when solvers do support all the necessary features, there is risk of a semantics mismatch. In this paper, we define translations that further reduce the complexity of the language. In particular, we work toward the goal of producing a linear program (LP) or mixed integer linear program (MILP) in standard form.

In section 2, we define and discuss these two problem classes. In section 3, we review Agarwal’s reduced language MIP (with some simplifying modifications). In section 4, we define and discuss the various translations necessary to reduce MIP to LP or MILP, and in section 5, we show the application of these transformations to a modest modeling problem defined by Agarwal.
2 Optimization Problems

2.1 Linear Programming

In an LP problem, one maximizes or minimizes over a linear objective function subject to linear constraints. That is, we consider a problem of the form:

max \( x \in \mathbb{R}^n \) \( c^T x \), \( c \in \mathbb{R}^n \)

\[ a_1^T x + \ldots + a_m^T x = b \]

These equations can also be inequalities; however, most algorithms require that there are no inequalities, as does much of the formal analysis. We can (and do) eliminate the inequalities by introducing slack variables[2]; for instance,

\[ a^T x \leq b \Leftrightarrow a^T x + s = b, s \geq 0. \]

Standard algorithms (though not solvers, in general) also require that all variables be bounded on at least one side; to eliminate completely unbounded variables, we rewrite them as a difference:

\[ x = u - v, u, v \geq 0. \]

It is easy to see that this is equivalent algebraically, although this could double the number of variables in the worst case. An alternative is to actually eliminate unbounded variables through algebraic manipulation[2]; currently our software takes the former approach.

We often prefer to have all variables to be precisely bounded by zero from below and infinity from above. To do this, we can translate variables or negate them as necessary; if they have two bounds, we write one as a constraint rather than a bound.

It is also easy to see that we can write the constraints and objective function in matrix form, with the matrices representing variable coefficients (and values, in case of the result vector \( b \)).

So with all of these potential transformations in mind, we define the standard form for LP to be

\[ \max_{x \in \mathbb{R}^n} c^T x, c \in \mathbb{R}^n \]

\[ Ax = b, A \in \mathbb{R}^{n \times m}, b \in \mathbb{R}^m \]

\[ x \geq 0. \]
2.2 Mixed Integer Linear Programming

An MILP problem is a linear program with a further constraint that some variables must take integer values. Most formally, we can add an additional constraint matrix and objective function vector for the integer variables similar to

\[ \max_x c^T x + f^T y, Ax + Dy = b, x_i \geq 0, y_i \geq 0, x \in \mathbb{R}^n, y \in \mathbb{Z}^m. \]

However, it is often easier in practice to use the LP standard form, plus a set of variable indices that must take integer values. This is the approach we take in this paper.

3 Mixed Integer Programming Language (MIP)

In interest of brevity, we will not define Agarwal’s full language MP, but only the reduced language MIP without Boolean variables, disjunction, or lambdas. We make a few further simplifications here for brevity’s sake.

\[ \tau ::= \text{real} \mid \text{int} \mid < r_1, r_2 > \mid < r, \infty) \mid (-\infty, r) \mid [n_1, n_2] \mid [n, \infty) \mid (-\infty, n) \]

\[ e ::= x \mid r \mid -e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid \#n e \mid e[\varepsilon] \mid \text{case}_i \varepsilon \text{ of } \{l_j \rightarrow e_j\}_j=1 \]

\[ c ::= T \mid F \mid e_1 = e_2 \mid e_1 \leq e_2 \mid e_1 \geq e_2 \mid \exists x : \tau . c \mid \bigwedge_{i : \sigma} c \mid \text{case}_i \varepsilon \text{ of } \{l_j \rightarrow c_j\}_j=1 \]

\[ p ::= \delta_{x_1 : \tau_1, \ldots, x_m : \tau_m} (e, c) \]

Here, \( e \) is an expression, \( \tau \) is an expression type, and \( c \) is a proposition. \( x \) is a variable, \( r \) is a constant, and \( n \) is an integer. \( l_j \) are elements of an index set, and \( \varepsilon \) is an index expression; for details, see Agarwal (2006). For our purposes, we simply need that there exists a translation \( \varepsilon \mapsto \text{real} \) that maps \( \varepsilon \) to \( i \in \sigma \), where \( \sigma \) is an index set.

One simplifying assumption we have made is that there are no lambdas, and there are no applications other than variable indexing. This is a consequence of
a preexisting translation by Agarwal, putting the program in application normal form (similar to the head normalized form of \(\lambda\)-calculus, this guarantees that all well-typed expressions are free of lambdas and applications). In practice, we actually apply this translation as we apply the ones outlined below, but we elide the details of this for brevity. The same is true of \texttt{let} bindings; we eliminate these trivially in the code, so we do not spend much time discussing them.

One important consideration is the underlying representation of real numbers. Throughout this paper, we define coefficients as real constants, which are associative, commutative, etc. However, in reality coefficients are stored using floating point numbers, which are neither associative nor commutative. Thus, we may lose precision in many of the transformations below; indeed, even the order of combining terms could be important in some cases to avoid total precision loss. One solution is to compute coefficients using infinite precision math. Other solutions are provided in detail by Neumaier and Shcherbina [3]. Our implementation does not address this, however.

\section{Transformations}

\subsection{Substitution}

Given a problem in MIP, we would like to produce a problem in LP standard form, including the \(x \geq 0\) constraints. To do this, we will eliminate all variables not already of this form, replacing them with expressions that contain such variables. Although this step is not strictly necessary for use with solvers, producing a more standard form makes adding future optimization and presolving steps easier.

We could have defined this transformation later in the chain, but the mapping makes use of special features of the MIP language to simplify the handling of index and projection types; furthermore, this transformation in other contexts where flattening (the next transformation) is undesirable.

\subsubsection{Variables}

The primary step is transforming variables based on their type. The idea is to transform a typed variable to a new typed variable, a substitution expression for the old variable, and optionally a extra proposition that must be satisfied in the translated program.

\[
< a, b > c \rightarrow (< 0, \infty), [x], x + a, x \leq b - a
\]

\[
< a, \infty > c \rightarrow (< 0, \infty), [x], x + a, T
\]

\[
(\infty, b > c \rightarrow (< 0, \infty), [x], b - x, T)
\]
\[ (-\infty, \infty) \rightarrow (0, \infty), [x, y], x - y, T) \]

\[ \tau \rightarrow (\tau', x, y, e, c) \]

\[ i : \sigma \rightarrow \tau \rightarrow (i : \sigma \rightarrow \tau', x, y, \lambda x \{ x[i]/x \} \in x \in x e, \bigwedge_{i \in \sigma} \{ x[i]/x \} \in x \in x c) \]

\[ \tau_1 \rightarrow (\tau'_1, x, y, e_1, c_1), \tau_2 \rightarrow (\tau'_2, x, y, e_2, c_2) \]

\[ \tau_1 * \tau_2 \rightarrow (\tau'_1 * \tau'_2, x, y, \bigcup x, e_1 \cup e_2, \bigcup c_1 \wedge c_2) \]

where

\[ P(x, n, s) = \{ \#n x/x \} \in x s. \]

The first four translations constitute the actual substitution in the real case. (We elide here the translations for integer ranges; they are analogous). We use simple algebra to define new variables that are bounded below at zero. Note that in the doubly-bounded case, we have an additional proposition to act as the upper bound. Also note that \( x \) and \( y \) are assumed to be unique variable identifiers within the program; here we elide the issues of carrying around a context to simplify the translations.

In the totally unbounded case (sometimes known as a free variable in the literature, although we will avoid this notation to prevent confusion), we actually introduce two new variables, noting that an unbounded variable can be expressed as the difference of two bounded-below variables. Unfortunately, this somewhat complicates the translation, since we now must account for the case where two variables are introduced.

The final two translations handle indexing and projection. Both translations first recursively translate their constituent types, yielding an expression and a proposition. The indexing substitution then adds an indexed lambda to the expression; this is of type \( \sigma \rightarrow \tau' \), as desired.

It also adds a indexed conjunction to the proposition, so that the proposition is evaluated for each label in the index set. Projections behave similarly, although they combine expressions into a tuple. This mapping is actually sometimes too broad; for example \( < 0, \infty) * (-\infty, \infty) \) uses two projection variables for a total of four expressions, whereas it really only needs three. This does not affect the program’s solution.\(^1\)

In practice, it is also important to keep track of the top-level substitutions in order to recover the original variables’ values. We do not discuss this here.

### 4.1.2 Propositions

We do not need to perform any explicit transformations on expressions, other than substitution of our new variables. We will do all this in the context of propositions.

\(^1\)The careful reader will note that we have not defined \( \lambda_x \), nor tuples. These are easily removed with the ANF transformation described above; again, see Agarwal (2006).
\[ T \rightarrow T \]
\[ F \rightarrow F \]
\[ e_1 \rho e_2 \rightarrow e_1 \rho e_2 \]
\[ c_1 \rightarrow c'_1, c_2 \rightarrow c'_2 \]
\[ c_1 \land c_2 \rightarrow c'_1 \land c'_2 \]
\[ c \rightarrow c' \]
\[ \bigwedge_{i, \sigma} c \rightarrow \bigwedge_{i, \sigma} c' \]
\[ \forall j \in \sigma c_j \rightarrow c'_j \]
\[ \text{case } \varepsilon \text{ of } \{ l_j \rightarrow c_j \}_{j=1}^m \rightarrow \text{case } \varepsilon \text{ of } \{ l_j \rightarrow c'_j \}_{j=1}^m \]
\[ \tau \rightarrow (\tau', x, \varepsilon, c') \]
\[ \exists x : \tau c \rightarrow \exists x_1 : \tau' \exists x_2 : \tau' \ldots \exists x_m : \tau' \{ e/x \} c \land c' \]

The only nontrivial case is the quantifier; the conjunction and case forms simply pass the translation through. Note that we potentially have to build a series of quantifiers if more than one variable is used in the substitution; other than that, we replace the now-free variable \( x \) with its new expression in the proposition and add a conjunction of the auxiliary proposition.

### 4.1.3 Program

We omit here the formal program transformation as it is essentially identical to \( n \) existential quantifiers. The only additional feature is the objective function, but this is a trivial substitution, \( \{ c'/x \} e \) for each variable \( x \).

### 4.2 Flattening the AST

Our first goal is to simplify and flatten the abstract syntax tree. Specifically, we wish to eliminate quantifiers, sums, case statements, and indexed conjunctions, and distribute multiplication over addition. To do this, we first define a simplified syntax:

\[
\begin{align*}
  v & ::= x \mid \#n x \mid x[i] \\
  \bar{v} & ::= [] \mid v \bar{v} \\
  t & ::= r \ast \bar{v} \\
  \nu & ::= [] \mid t \nu \\
  \rho & ::= \leq \mid = \mid \geq \\
  R & ::= (\rho, \nu)
\end{align*}
\]
Here we call $t$ a *term*, and $v$ a *term variable*. A term variable is just a variable, a projection, or an index; a term is a product of term variables with a coefficient. $\nu$, as a list of terms, represents a polynomial. $R$ represents an equation; specifically, that the polynomial $\nu$ is either less-than-or-equal, equal, or greater-than-or-equal to zero (naturally, any polynomial equation or inequality can be written in this form).

Our subsequent translations will aim to translate expressions to polynomials and propositions to sets of equations.

### 4.2.1 Expression to Polynomial

Below is the translation of an MIP expression to a polynomial.

\[
\begin{align*}
  x & \rightarrow [(1,0, [x])] \\
  r & \rightarrow [(r, [])] \\
  -1 \ast e & \rightarrow \nu \\
  -e & \rightarrow \nu \\
  e_1 & \rightarrow \nu_1, \ e_2 \rightarrow \nu_2 \\
  e_1 + e_2 & \rightarrow \nu_1 \sqcup \nu_2 \\
  e_1 + (-e_2) & \rightarrow \nu \\
  e_1 - e_2 & \rightarrow \nu \\
  e_1 \ast e_2 & \rightarrow [(r_1 \ast r_2, t_1 \sqcup t_2)]((r_1,t_1),(r_2,t_2)) \in \nu_1 \times \nu_2 \\
  \forall j \in \sigma \{j/i\} e & \rightarrow \nu_j \\
  \sum_{i \in \sigma} e & \rightarrow [\nu_j]_{j \in \sigma} \\
  \varepsilon & \text{eval} \ l_k, e_k \rightarrow \nu_k \\
  \text{case}_i \varepsilon \ of \ \{l_j \rightarrow e_j\}_{j=1}^{m} & \rightarrow \nu_k \\
  e & \rightarrow [r, [x]], \varepsilon \text{ eval} \ i \\
  e[\varepsilon] & \rightarrow [r, [x][\varepsilon]] \\
  e & \rightarrow [r, [x]] \\
  \#n e & \rightarrow [r, [\#n x]]
\end{align*}
\]

Here, addition just unions the two polynomials together; we define $\sqcup$ to be a term union, combining terms with the same term variables by adding coefficients. Multiplication takes the Cartesian product of the two polynomials and multiplies each pair within; this is just distribution of multiplication over addition. Sums are expanded at this point, and case statements are evaluated. The projection and index operations just modify the underlying term variable; here $r$ should actually always be 1. MIP’s typing rules ensure that projection or indexing will never occur on a polynomial with more than one term (and that term must be linear), so the translation is completely defined on MIP.
4.2.2 Proposition to Equations

Since we are dealing with quantifiers, we now have to be more careful about free variables. We let $\Gamma$ be a context $^2$, associating a type with each encountered variable. Our translation takes a context and a proposition as input, and outputs a modified context and a set of equations.

$$
\begin{align*}
\Gamma, T & \rightarrow \Gamma, [] \\
\Gamma, F & \rightarrow \Gamma, \bot \\
e_1 & \rightarrow \nu_1 - e_2 \rightarrow \nu_2 \\
\Gamma, e_1 \rho e_2 & \rightarrow \Gamma, ([\rho, \nu_1 \sqcup \nu_2]) \\
\Gamma, c_1 & \rightarrow \Gamma', \bar{R}_1 \Gamma', c_2 \rightarrow \Gamma'', \bar{R}_2 \\
\Gamma, c_1 \land c_2 & \rightarrow \Gamma'', \bar{R}_1 \sqcup \bar{R}_2 \\
\forall_{j \in \sigma} \Gamma_{j-1} \{j/i\} c & \rightarrow \Gamma_j, \bar{R}_j \\
\Gamma_0, \bigwedge_{i, \sigma} & \rightarrow \Gamma_m, \text{flatten}[\bar{R}_{j|j=1}] \\
\Gamma, x : \tau, c & \rightarrow \Gamma', \bar{R} \\
\Gamma, \exists x : \tau . c & \rightarrow \Gamma', \bar{R}
\end{align*}
$$

Much of this translation is obvious. A relation proposition is translated by translating each side (we negate the right side and add it to the left since our polynomials always have zero on the right). A conjunction unions the resultant two equation sets of its operands, but N.B.: we carry the context through to the second operand from the first. It is the same with the indexed conjunction, although here we use the flatten function as a notational convenience; it simply unions a set of sets into one set.

The existential transformation should be considered in greater detail. We cannot completely eliminate quantifiers; we need to maintain a record of each variable’s existence and type for later use and to ensure that further transformations do not shadow or otherwise reuse the variable’s identifier. Instead, we wish to move quantifiers to the top-level. In logic, this is known as prenex normal form.

The rules for moving quantifiers outward depend on the proposition operations available. Since we are only dealing with conjunction at this point, the rule is simple: we can directly move a quantifier up one level if doing so does not bind any free variables.

We do this all in one step. By propagating the context linearly as we walk the tree, we keep a record of each variable identifier used throughout the program. This allows us to give each variable a unique identifier as we encounter it,

---

$^2$This is somewhat of an abuse of notation. Since we are passing the context linearly through the entire program, it is more of a variable map than a context, which usually only contains variables currently in scope; of course, since we are lifting existentials, all variables will eventually be in scope.
guaranteeing that all variables will be unique, and thus that we will never bind a free variable. By passing the context back as part of the translation, at the end we have a list of quantified variables, and we are immediately in prenex normal form.

4.2.3 Program

We will not define the program translation formally, since it is largely notational. Instead, recall that a program in MIP is a set of variables, a direction, an objection equation, and a proposition. In our simplified language, a program is a list of variables, an objective polynomial, and a list of equations. In our simplified form, we assume the direction is maximize, and we negate the objective polynomial during translation in the other case. The original variables form the initial context for the proposition translation.

4.3 Standard Form

The final transformation is one to standard LP or MILP form. This is somewhat ad-hoc, since the main reason to put it in standard form is to then output it to an input file for a solver. Additionally, our polynomial equation form is already very close to a matrix form; the primary step is to convert term variables to integers (representing \(x_1, x_2, \ldots\)). We enumerate the term variables of a type and use the term variable’s position in this list as its integer index.

We define the final form to be a coefficient vector for the objective function, a coefficient matrix for the constraints (all equalities), and a result vector. We map to these vectors and matrix by using the above term variable map. It is easy to read back the term variables this way as well, once we receive a solution from a solver. Combined with the substitution expressions from the first translation, we fully express the solution in the form originally specified by the user. (A similar technique could be useful if implementing various optimizations or presolve techniques).

After this, it is generally a solver-specific translation to generate an input file. We chose to write code to generate lp_solve input files, since lp_solve is freely available and the input format fairly simple, being just a textual list of constraints. We must be careful to mark the correct variables as integers in the input file, however, by referring to the type of each term variable as it is converted. Similarly, one must convert back from the solutions reported by the solver; this is just some minor parsing.

5 Results

We were able to translate a variety of small examples to MILP and solve them with the lp_solve open source solver. The largest solved example was the switched flow process model developed in Agarwal (2006).

Due to some issues with performance and difficulties finding satisfiable models, we chose to use rather simple parameters for the flow model. In particular,
we chose only three steps, with a maximum time of 20. It is not clear whether 
these issues are due to a bug in the code or an issue with the problem itself. 
Smaller examples seem to work perfectly, but are of less interest. 

Below is the code in MP form.

let

expri n = 3 (* rather small *)
set N = {1,...,n}
set N1 = {1,...,n-1}
set N2 = {1,...,n-2}

set CLOCK = {'R', 'S'}
set P = {'s','e'}
set AUT = {'alpha', 'beta'}

typei funi MODES (a) : [AUT] -> set =
case a of
  'alpha' => {'on', 'off'}
| 'beta' => {'lo', 'hi'}

expr matlFlowa:[MODES['alpha']]->real=
  fni q.
  case _{i.real} q of 'on'=>2.0|'off'=>0.0

expr matlFlowb:[MODES['beta']]->real=
  fni q.
  case _{i.real} q of 'hi'=>4.0|'lo'=>0.5

expr costFlowa:[MODES['alpha']]->real=
  fni q.
  case _{i.real} q of 'on' =>10.0|'off'=>0.0

expr costFlowb:[MODES['beta']]->real=
  fni q.
  case _{i.real} q of 'hi'=>15.0|'lo'=>2.0

expr Fout = 1.8
expr Minit = 20.0
expr Mmax = 150.0
expr Mmin = 10.0
expr Tmax = 20.0 (* rather small *)
var Cost : [P * N] -> real
var isInMode : [a:AUT * MODES[a] * N] -> bool (* Y *)
var clock : [CLOCK] -> [P * N] -> <0.0, 1000.0>

min Cost['e',n] subject_to

exists costJmp : [AUT * N] -> <0.0, 50.0>
exists clockJmp : [CLOCK] -> [N] -> <=1000.0, 0.0>
exists delt : [N] -> <0.0, 500.0>
exists goesFromTo : [a:AUT * MODES[a] * MODES[a] * N1] -> bool (* Z *)
exists isDummyTrans : [AUT * N1] -> bool (* YY *)
exists isDummyEvent : [N1] -> bool (* YYY *)

exists wmbar : [AUT * N] -> <0.0, 2000.0>
exists wcbar : [AUT * N] -> <0.0, 7500.0>

(* timeline constraints *)
(CONJ i:N . t['s',i] <= t['e',i]),
(CONJ i:N1 . t['e',i] = t['s',i+1]),
( CONJ i:N . delt[i] = t['e',i] - t['s',i]),
t['e',n] = Tmax,
t['s',i] = 0.0,

(* disjunction over modes of alpha *)
let
expr funi Rmax (q) : [MODES['alpha']] -> real =
case _{i.real} q of 'on' => 30.0 | 'off' => 1000.0
in
CONJ i:N . DISJ q:MODES['alpha'] .
isTrue isInMode['alpha', q, i],
CONJ p:P . clock['R'][p,i] <= Rmax[q]
end,

(* disjunction over transitions of alpha *)
( CONJ i:N1 .
let
prop p_OnOff =
isTrue goesFromTo['alpha', 'on', 'off', i],
costJmp['alpha', i] = 0.0,
clockJmp['R'][i] = ~clock['R']['e', i]
prop p_OffOn =
isTrue goesFromTo['alpha', 'off', 'on', i],
clock['R']['e', i] >= 2.0,
costJmp['alpha', i] = 50.0, 
clockJmp['R'][i] = ~clock['R'][e, i] 

prop p_Dummy = 
isTrue isDummyTrans['alpha', i], 
costJmp['alpha', i] = 0.0, 
clockJmp['R'][i] = 0.0 
in 
p_OnOff disj p_OffOn disj p_Dummy 
end),

(* disjunction over modes of beta *) 
let 
expr funi Smax (q) : [MODES['beta']] -> real = 
case _{i.real} q of 'hi' => 40.0 | 'lo' => 1000.0 
in 
CONJ i:N . DISJ q:MODES['beta'] . 
isTrue isInMode['beta', q, i], 
CONJ p:P . clock['S'][p,i] <= Smax[q] 
end, 

(* disjunction over transitions of beta *) 
(CONJ i:N1 . 
let 
prop p_HiLo = 
isTrue goesFromTo['beta', 'hi', 'lo', i], 
costJmp['beta', i] = 0.0, 
clockJmp['S'][i] = ~clock['S'][e, i] 

prop p_LoHi = 
isTrue goesFromTo['beta', 'lo', 'hi', i], 
clock['S'][e, i] >= 3.0, 
costJmp['beta', i] = 40.0, 
clockJmp['S'][i] = ~clock['S'][e, i] 

prop p_Dummy = 
isTrue isDummyTrans['beta', i], 
costJmp['beta', i] = 0.0, 
clockJmp['S'][i] = 0.0 
in 
p_HiLo disj p_LoHi disj p_Dummy 
end 
), 

(* clock dynamics *) 
(CONJ c:CLOCK . 
12
\[
\text{(CONJ } i : N . \ \text{clock}\{c\}[s', i] = \text{clock}\{c\}[s', i] + \text{delt}\{i\}),
\text{(CONJ } i : N_1 . \ \text{clock}\{c\}[e', i+1] = \text{clock}\{c\}[e', i] + \text{clockJump}\{c\}[i]),
\text{clock}\{c\}[s', 1] = 0.0
\)

(* material level dynamics *)
\[
\text{(CONJ } i : N . \ \text{Matl}\{e\}[i] = \text{Matl}\{s\}[i] + \text{SUM } a : \text{AUT} . \ \text{wmbar}\{a\}[i]) + \text{Fout} * \text{delt}[i]),
\text{(CONJ } i : N_1 . \ \text{Matl}\{s\}[i+1] = \text{Matl}\{e\}[i]),
\text{Matl}\{s\}[1] = \text{Minit},
\text{(CONJ } i : N . \ \text{CONJ } p : P . \ \text{Mmin} \leq \text{Matl}\{p\}[i], \ \text{Matl}\{p\}[i] \leq \text{Mmax}),
\]

(* cost dynamics *)
\[
\text{(CONJ } i : N . \ \text{Cost}\{e\}[i] = \text{Cost}\{s\}[i] + \text{SUM } a : \text{AUT} . \ \text{wcbar}\{a\}[i]),
\text{(CONJ } i : N_1 . \ \text{Cost}\{s\}[i+1] = \text{Cost}\{e\}[i] + \text{SUM } a : \text{AUT} . \ \text{costJump}\{a\}[i]),
\text{Cost}\{s\}[1] = 0.0
\]

(* definition of wmbar and wcbar *)
\[
\text{(CONJ } i : N . \ \text{DISJ } q : \text{MODES}'\alpha' . \ (\text{isTrue isInMode}'\alpha', q, i),
\text{wmbar}'\alpha', i] = \text{matlFlow}\{a\}[q] \ast \text{delt}[i],
\text{wcbar}'\alpha', i] = \text{costFlow}\{a\}[q] \ast \text{delt}[i])
\]

\[
\text{(CONJ } i : N . \ \text{DISJ } q : \text{MODES}'\beta' . \ (\text{isTrue isInMode}'\beta', q, i),
\text{wmbar}'\beta', i] = \text{matlFlow}\{a\}[q] \ast \text{delt}[i],
\text{wcbar}'\beta', i] = \text{costFlow}\{a\}[q] \ast \text{delt}[i])
\]

(* make sure each automaton is only in one mode in each interval *)
\[
\text{(CONJ } i : N . \ \text{let expr on } = \text{isInMode}'\alpha', 'on', i
\text{expr off } = \text{isInMode}'\alpha', 'off', i
\text{in isTrue (on or off) and not (on and off)
end,
let expr hi = isInMode'\beta', 'hi', i
\text{expr lo } = \text{isInMode}'\beta', 'lo', i
\text{in isTrue (hi or lo) and not (hi and lo)
end
)
\]

(* symmetry breaking *)
\[
\text{(CONJ } i : N_2 . \ \text{isTrue isDummyEvent}[i] \implies \text{isDummyEvent}[i+1]),
\text{(CONJ } i : N_1 . \ \text{isTrue not isDummyEvent}[i] \disj \text{delt}[i+1] = 0.0))
\)

(* definition of isDummyTrans *)
(CONJ i:N1 . isTrue isDummyTrans['alpha', i] <==>
(goesFromTo['alpha','on','on',i] or goesFromTo['alpha','off','off',i])),
(CONJ i:N1 . isTrue isDummyTrans['beta', i] <==>
(goesFromTo['beta','hi','hi',i] or goesFromTo['beta','lo','lo',i])),

(* definition of isDummyEvent *)
(CONJ i:N1 . isTrue isDummyEvent[i] ==>
(isDummyTrans['alpha',i] and isDummyTrans['beta',i])),

(* definition of goesFromTo *)
(CONJ i:N1 . CONJ a:AUT . CONJ q1:MODES[a] . CONJ q2:MODES[a] .
isTrue goesFromTo[a,q1,q2,i] ==>
(isInMode[a,q1,i] and isInMode[a,q2,i+1]))

end

The resulting lp_solve input file is too large to include in this document; it
would increase the length by 84 pages of text. However, the results after solving
are as follows:

t['(e',1)] = 0.0
t['(e',2)] = 2.84217E^14
t['(e',3)] = 20.0
t['(s',1)] = 0.0
t['(s',2)] = 0.0
t['(s',3)] = 2.84217E^14
Matl['(e',1)] = 20.0
Matl['(e',2)] = 20.0
Matl['(e',3)] = 66.0
Matl['(s',1)] = 20.0
Matl['(s',2)] = 20.0
Matl['(s',3)] = 20.0
Cost['(e',1)] = 0.0
Cost['(e',2)] = 0.0
Cost['(e',3)] = 40.0
Cost['(s',1)] = 0.0
Cost['(s',2)] = 0.0
Cost['(s',3)] = 0.0
isInMode['(alpha','off',1)] = 0.0
isInMode['(alpha','off',2)] = 0.0
isInMode['(alpha','off',3)] = 1.0
isInMode['(alpha','on',1)] = 1.0
isInMode['(alpha','on',2)] = 1.0
isInMode['(alpha','on',3)] = 0.0
isInMode['(beta','hi',1)] = 1.0
isInMode['(beta','hi',2)] = 0.0
isInMode[('beta','hi',3)] = 0.0
isInMode[('beta','lo',1)] = 0.0
isInMode[('beta','lo',2)] = 1.0
isInMode[('beta','lo',3)] = 1.0

clock[‘R’][('e',1)] = 0.0
clock[‘R’][('e',2)] = 0.0
clock[‘R’][('e',3)] = 20.0
clock[‘R’][('s',1)] = 0.0
clock[‘R’][('s',2)] = 0.0
clock[‘R’][('s',3)] = 0.0
clock[‘S’][('e',1)] = 0.0
clock[‘S’][('e',2)] = 0.0
clock[‘S’][('e',3)] = 20.0
clock[‘S’][('s',1)] = 0.0
clock[‘S’][('s',2)] = 0.0
clock[‘S’][('s',3)] = 0.0

6 Conclusions and Further Research

We have presented a variety of translations used in converting Agarwal’s MIP language to a MILP program in standard form. We have demonstrated that these transformations, in conjunction with the ones defined in Agarwal, are able to produce a working LP program that can be solved with an external solver.

A major shortcoming of the current software is speed. Many algorithms were chosen out of convenience and not efficiency; for example, let-bindings are eliminated separately from other tree recursions, rather than walking the tree with an appropriate context. In the worst case, this probably leads to a quadratic number of recursions, which can be substantial when there are thousands of propositions. Similarly, some of the alpha conversion code is quadratic in nature, and slows down substantially in large problems. The existing transformations suffer from some of these same problems, but not to the same degree as our new ones.

Solvers exist for a variety of other types of convex optimization problems other than MILP; there are even solvers for the general case where the user provides the non-linear function as C code. It would be useful to support translations for at least some of these problem classes, particularly quadratic optimization which is commonly supported and has a well-defined standard form. This would not be difficult with the current framework; only the matrix transformation would need to be replaced, since the other transformations already support non-linear terms.

Another area of expansion is integration with solvers. The current version of the software generates and outputs a LP file for the lp_solve solver, runs the solver, and parses the results. Not only is this not robust, but it would be tedious to write parsing code for each solver’s output format. A better approach would
be to take advantage of modern solvers’ APIs and link the software directly with various solvers, as is done in existing tools such as AMPL. This would be an important step toward making the software usable for serious projects.

One difficulty in using the software is that it is very difficult to get useful feedback if the solver indicates that the system is infeasible. This was a major difficulty in debugging the transformations, as the resultant system would often have a contradiction, but due to their complexity it was nearly impossible to find it. It would be useful to have a consistency check within the solver that could point out potential conflicts before sending the matrix to the solver.

References

