Logical Modeling Frameworks for the Optimization of Discrete-Continuous Systems

Ashish Agarwal

DISSERTATION

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR

the degree of

DOCTOR OF PHILOSOPHY
in
CHEMICAL ENGINEERING

Carnegie Institute of Technology
CARNEGIE MELLON UNIVERSITY
Pittsburgh, PA, U.S.A.

Abstract

Often, it is very difficult to pose a model for a system even after the system is conceptually understood. The reason is the mathematical languages we employ have few forms of expression. We define more expressive languages, first for dynamical discrete-continuous systems, and then more rigorously for mathematical programs (MP). Our approach provides theoretical basis for designing MP software.

The first framework we define is called linear coupled component automata (LCCA). It supports finite domain constraints, explicitly handles dynamics, and enforces modular modeling. We show how LCCA models can be mechanically converted into mathematical programming (MP) constraints. Currently, chemical process systems are usually modeled directly with MP constraints. We show with an example that it is much easier to model hybrid systems in our LCCA framework.

We then pursue a more rigorous approach for the MP part of our work, for the purposes of providing a computer implementation of an MP framework. There are two main results: a rich computer language for declaring MPs and automation of certain model transformations. Mathematically, these correspond to defining a set p of MPs and defining a binary relation on p.

The set p contains programs as one would want to write in practice, not just canonical matrix forms. Complex index sets can be defined in intuitive ways, and they are first-class entities in our theory, not mere notational conveniences eliminated at parse time. This has many benefits: it retains knowledge of the problem structure, keeps the program size to a minimum, and speeds up certain operations. Our definition of the semantics elucidates the nature of MP algorithms and explains the information sought from a solution.

The binary relation on programs p can be defined because a logical formulation allows treating constraints and programs as mathematical objects. Principally, our definition includes: a procedure for putting Boolean expressions into conjunctive normal form, and a procedure for converting disjunctive constraints into mixed-integer inequalities. Neither has been defined previously for a language as expressive as ours, and the latter has not been defined as a formal mapping on constraint spaces. Overall this leads to a procedure for converting general MPs to pure mixed-integer programs (MIPs).

Sets and set relations are defined using the methods of type theory, which espouses a close relation between mathematics and computation. As a result, the set p can be viewed simultaneously as a novel definition of MP and as the software architecture for implementing an MP language. Similarly, the binary relation on p can be directly implemented on a computer. Several examples of our software's input and output are provided.

Acknowledgments

Ignacio E. Grossmann has been my co-advisor throughout the Ph.D. program. His work ethic is well known and appreciated. He has a genuine concern for his students, which I have seen through the many difficulties he has helped me overcome. His knowledge is broad and he has led me to study subjects in diverse areas.

My committee members have provided much guidance over the years. Bruce H. Krogh's excellent course on hybrid systems originally got me interested in the area. John N. Hooker has taught me much about the use of logic in mathematical programming. Gary J. Powers has always supported my efforts to employ Computer Science methods in Chemical Engineering, something he has been doing for years. Lorenz T. Biegler provided a fresh perspective on the value of my work.

I have been fortunate to have had great group members. I mention especially Vikas Goel and Nicolas W. Sawaya. Vikas and I studied for many courses together our first year and he was always someone I could discuss research problems with. My long discussions with Nick are the reason I understand many of the concepts in disjunctive programming.

David Swasey has been immensely kind with his time. With no particular benefit to him, he volunteered to meet me regularly to assist with my ML programming efforts. Without his guidance, my Ph.D. would certainly have taken longer and the software would be in poorer shape.

My dad always told me I could do anything, and my mom always made me think about the value of each thing I chose to do. Both have always been completely supportive and given me the freedom to pursue my interests. My sisters Monika and Nisha raised me alongside my parents. Their optimism in my abilities is certainly more than I deserve, but it has driven me all along.

The Ph.D. program brought me to Pittsburgh, a city to which I had no prior connections. Now I have so many wonderful friends here; it will be difficult to leave. I won't try to name them all, but I want them to know how important they are. I am certain these friendships will last a lifetime.

Finally, I will attempt to explain the influence my co-advisor Robert Harper has had on me. Bob is a remarkable researcher and an inspiring teacher. In the last few years, he has taught me, through individual sessions, more mathematics than I ever imagined I would know. I rarely managed to pose a question he could not answer. On those few occasions that he could not derive a result on the spot, he committed himself to finding the answer and emailing me a thorough explanation. I am privileged to have worked with Bob and honored that he chose to advise me.

Brief Contents

Detailed Contents

List of Figures

Chapter 1

Introduction

Modeling refers to the process of converting a conceptual understanding of a system into a mathematical representation, or model. Often, we understand a system well but still have difficulty declaring a model for it. This is because there is a large discrepancy in the expressive power of the natural language, e.g. English, we think in and the mathematical language, e.g. linear algebra or the infinitesimal calculus, in which the model must be declared. Although natural languages are highly expressive, they are not formal. One approach to closing this gap would be to attempt a formal interpretation of natural languages, but this has little chance of success. A more modest goal is to enhance formal languages with additional forms of expression.

In particular, we are concerned with modeling dynamical systems with mixed discrete-continuous phenomena, called hybrid systems. (Systems considered in this work are restricted to piecewise-linear dynamics in the continuous regime.) These are increasingly important in the chemical process industry, where continuous dynamics occur in the form of reaction rates and material flows, and discrete dynamics arise from control decisions that for example call for one or another reaction to be run at various times. Such a system cannot be modeled as a set of differential equations alone, the theory within which most engineering models are presently defined. In the process industry, these systems have usually been modeled with mathematical programming (MP) constraints.

More recently, frameworks based on hybrid automata (HA) are being increasingly used. Several variations of this framework exist, usually with an emphasis on their differing levels of theoretical expressivity. We introduce another HA style framework that we call linear coupled component automata (LCCA). Our emphasis is on providing features that ease modeling in practice. We show through examples that LCCA models are significantly easier to formulate than MP models.

There is, however, a mature theory for optimizing systems represented in the MP framework. So it would be valuable to derive the equivalent MP model from a given LCCA model. We provide a transformation procedure for doing this, thereby allowing us to use LCCA for its modeling benefits and MP for its algorithmic ones. However, neither the definition of LCCA nor its transformation to MP are provided in a manner leading to computer implementation. This is a shortcoming of much existing work also, as we will discuss in

detail.

We address this by providing a type theoretic (logical) formulation of mathematical programming. This firstly allows inclusion of programs as written in practice, not just canonical forms. It thus leads immediately to a rich computer language for expressing mathematical programs. Of principal importance is our support of index sets, formalized as a logic of finitary types.

Secondly, a logical formulation defines programs and constraints as mathematical objects on which operations can be defined, as where previous definitions of MP only support numeric operations. This allows us to provide Boolean and disjunctive constraints, which are often intuitive declarations, and automatically transform them to pure mixed-integer programming (MIP) constraints, the form required by most algorithms.

In summary, we claim that

> It is possible to design more expressive mathematical languages to facilitate modeling, and to compile these to prior languages, allowing use of existing algorithms.

In the rest of this chapter, we describe the modeling challenges and the proposed solution methodology in more depth.

1.1 Modeling Challenges

Consider the following conceptual description of a common process in the chemicals industry:

> There are two streams flowing into a reactor, one at rate 6.3 and the other at rate 5.0. What is the total flow into the reactor?

We would like to declare a mathematical model of this system, which is

$$f_1, f_2, f_{\text{tot}} \in \mathbb{R} \tag{1.1a}$$

$$f_1 = 6.3 \tag{1.1b}$$

$$f_2 = 5.0 \tag{1.1c}$$

$$f_{\text{tot}} = f_1 + f_2 \tag{1.1d}$$

This small example explains what we mean by modeling. First, variables representing the appropriate quantities must be declared. Secondly, statements on those variables must be made. We have converted the conceptual explanation of a system into mathematical statements.

Variable declarations are often omitted in engineering models because it is implicit that all variables are of type real. But providing multiple data types is a principal technique we use to enhance a framework's expressivity, so we will have to state variables' types.

Now, let us consider a slightly more complicated system:

> A reactor can run one of three reactions. If rxnX is run, then the temperature must stay below 400. If rxnZ is run, the temperature must be below 600, and the maximum temperature the reactor can handle is 1000.

Again, we must think of a formal model. Here is one option:

$$x, y, z \in \{0, 1\} \tag{1.2a}$$

$$\Theta_x, \Theta_y, \Theta_z \in \mathbb{R} \tag{1.2b}$$

$$\Theta \in \mathbb{R} \tag{1.2c}$$

$$x + y + z = 1 \tag{1.2d}$$

$$\Theta = \Theta_x + \Theta_y + \Theta_z \tag{1.2e}$$

$$0 \leq \Theta_x \leq 400x \tag{1.2f}$$

$$0 \leq \Theta_y \leq 1000y \tag{1.2g}$$

$$0 \leq \Theta_z \leq 600z \tag{1.2h}$$

Variables x, y, and z must take either the value 0 or 1, and the others are of type real. The first equation assures that exactly one of the reactions is run. The second appears odd; it says that the overall temperature is a sum of three others. In the last three constraints, we have assured that exactly one of Θ_x, Θ_y, or Θ_z will be non-zero, and it will be bounded between a minimum and maximum value. Now we see how the temperature equation works. It is effectively setting the actual temperature variable of concern Θ to one of the Θ_i's.

Formulating model (1.2) was somehow more complicated than formulating (1.1). Model (1.1) is self-explanatory. All that was required was to convert English words in the conceptual description into mathematical symbols. In contrast, constraints in model (1.2) rely on various tricks. Adding three temperatures is an awkward statement of physics and was certainly not implied by the conceptual description.

Conversely, the conceptual description does require the temperature Θ to stay below 400 if rxnX is run, but none of the constraints in the formal model states this directly. It is only by understanding all statements simultaneously that we infer this condition is being enforced. Thus, a statement that stands alone in our conceptual understanding has somehow become coupled with all other statements in the formal model. Modularity is broken. Removing this condition or adding a similar condition for a fourth reaction would require modifying multiple constraints in the model.

Both conceptual systems were equally easy to understand. So what made modeling the second one more difficult? The first one is a purely continuous system, as where the second combines discrete and continuous phenomena. Perhaps discrete-continuous systems are just fundamentally harder to model. We argue that this need not be the case. Rather, the difficulty arose from selecting an inappropriate modeling framework.

The framework or language employed to model the first system is linear algebra. The second employs mixed-integer programming (MIP) constraints[1], discussed in Nemhauser and Wolsey (1999). MIPs extend linear algebraic style constraints by allowing select variables to be restricted to integer values. This single extension allows modeling a much broader class of systems, and changes the required algorithms significantly.

MIPs allow modeling a broad class of systems in theory, but general mathematical programs frequently employ other constructs that significantly ease model formulation in practice. Balas (1974) discusses the use of disjunctive constraints, which require only one of several inequalities to hold, and Raman and Grossmann (1994) introduced the use of Booleans within the disjunctive constraints. With these additional language features, we can provide the following MP model for the above three reaction system,

$$x, y, z \in \{\texttt{true}, \texttt{false}\} \tag{1.3a}$$

$$\Theta \in \mathbb{R} \tag{1.3b}$$

$$x \veebar y \veebar z \tag{1.3c}$$

$$0 \leq \Theta \leq 1000 \tag{1.3d}$$

$$\begin{bmatrix} x \\ \Theta \leq 400 \end{bmatrix} \vee [y] \vee \begin{bmatrix} z \\ \Theta \leq 600 \end{bmatrix} \tag{1.3e}$$

where $\veebar$ stands for exclusive or and $\vee$ for normal or. All variables are physically reasonable, i.e. part of the conceptualization of the system, and the constraints are virtually self-explanatory. It is clear that 1000 is an upper limit on the temperature irrespective of what reaction is run, as where in the previous model it appeared to be associated with rxnY. The MP model (1.3) is easier to declare, understand, and modify, as compared to the MIP model (1.2). It also has fewer variables and fewer constraints.

Although the general MP model serves as a better modeling framework, most algorithms require the model in the pure MIP format. Ideally then, one could provide the MP model and have it converted automatically to the MIP model. The operative word here is automatically. It is well known how to do this manually. Model (1.2) is known as the convex-hull reformulation of model (1.3). This method is discussed in Raman and Grossmann (1994) and follows from Balas (1985). The automation challenge is discussed soon, but for now we proceed with a broader discussion.

The previous example shows that two frameworks, MIP and MP, can be used to model the identical physical system. Other frameworks also exist (e.g. Hooker and Osorio, 1999), and, most importantly, can be invented. A language supporting finite domain variables would allow replacing the three variables x, y, and z with a single variable

$$q \in \{\texttt{rxnX}, \texttt{rxnY}, \texttt{rxnZ}\}.$$

[1]MILP models are always declared for optimization purposes. So an objective is always included also. We disregard this for now as we are focussed on the modeling aspects.

This would eliminate the need for the mutual exclusivity constraint, equation (1.2d) in the MIP model and (1.3c) in the MP model.

MP is a richer modeling framework than pure MIP because it provides Boolean operators and disjunctive constraints. More sophisticated types and operators will be required to model real industrial systems. We discuss such systems in the next section and review the previous efforts made to model them.

1.2 Chemical Process Systems

Much of the early work on chemical process systems had to do with the design of chemical plants. The mathematical model overall can be viewed as a system of differential algebraic equations (DAEs). However, one cannot simply begin writing DAEs for such a large system. It is necessary to consider smaller parts individually and then couple those parts. A conceptual guide for doing this is the flowsheet. Although not discussed as a modeling framework in the literature (e.g. Westerberg et al., 1979), we can see that a flowsheet is essential to organizing one's understanding about a large-scale system. A chemical plant is viewed as consisting of separate unit operations connected by pipes. Each unit operation can be defined separately and the equations for each coupled afterwards. Flowsheets are so standard that they are hardly recognized as an innovation, but imagine trying to declare all the equations describing a chemical plant without envisioning a flowsheet.

Flowsheets were used to model only the continuous aspects of a chemical plant. Kondili et al. (1993) introduced the state-task network (STN) to allow modeling systems with discrete and continuous aspects. In a flowsheet, each node represents a specific volume of space, e.g. a reactor or heat exchanger. A task node, on the other hand, represents an abstract process, which might occur in any one of several locations. The discrete choice of which unit will run a process can thus be represented.

Discussions about STN models often regard computational efficiency (e.g. Maravelias and Grossmann, 2003; Ierapetritou and Floudas, 1998). When perceived as a modeling framework, their use can be described as follows. A modeler draws a diagram of the network, specifies the available equipment, states which tasks can run on which equipment, and declares other numerical parameters. All the information required to understand the system is contained in such a description, but this knowledge is not encoded in a formal theory. The formal theory is mathematical programming, and ultimately the MP constraints must be provided. The cited works explain how such models can be obtained from the conceptual description.

Flowsheets, STNs, and other similar frameworks are purely conceptual. They aid the modeler in formulating a model, but they themselves cannot be considered formal models. Our aim is to provide novel formal modeling frameworks.

In summary, mathematical programming models have been a mainstay in the process industry (Kallrath, 2000) when the concern is to optimize mixed discrete-continuous systems. Unfortunately, MP models can require significant expertise to formulate. Conceptual frameworks such as the STN reduce this burden, but the MP model must eventually be provided. It would be useful to have alternative frameworks in which formal models could

be expressed more easily than in MP. We discuss such alternatives in the next section.

1.3 Hybrid Systems

Mathematical programs allow representing systems with mixed discrete-continuous dynamics. Dynamics are not however explicitly supported. Time has no special status; it is just another variable. As a result there is no support for making statements such as something happens after something else and the conditions under which this is so. Such a concept can be expressed in theory, but much ingenuity might be required to think of the appropriate constraints. Several frameworks have been proposed recently to represent dynamical discrete-continuous systems, called hybrid systems.

Barton and Pantelides (1994) provided one of the earliest in the process literature (but see references therein for earlier efforts). They augmented the theory of differential algebraic equations (DAEs) with an index set, which serves as the possible discrete modes of a system. Each index was associated with a different DAE. Their focus was on the numerical complications that arise as a consequence of switching between DAEs. For instance, how to accurately detect the time at which a switching condition is satisfied, called event detection. The language for expressing discrete conditions is still not very flexible; only a single discrete variable is allowed for example.

The mixed logical dynamical system defined in Bemporad and Morari (1999) incorporates difference equations, where some of the variables can be binary 0-1. They discuss the use of Boolean logic, but mostly from the angle of showing that such statements can be represented as integer inequalities. The system they formally introduce does not allow Boolean propositions. Their emphasis is on analyzing a system which is theoretically expressive, but declaring models directly in this framework would still be difficult.

The frameworks coming into most common use are based on a combination of discrete automata and differential equations. Automata are an elegant method for describing discrete dynamics, and differential equations are of course the standard for continuous dynamics. Their combination creates a system more expressive than either alone. Variations, usually in the generality of continuous dynamics allowed, lead to specific frameworks.

Alur and Dill (1994) introduced timed automata, which allow variables measuring elapsed time. Within each discrete mode, the differential equations are of the form $\mathrm{d}x/\mathrm{d}t = 1$. Interesting systems can be represented when such equations are combined with discrete conditions. The variable can have its value reset to 0 when the system transitions to a new discrete mode. Also, switching out of a mode can be restricted based on the value of this variable, requiring the system to remain in a certain mode for a specified amount of time. The timed automata is somewhat the opposite extreme of the efforts of Barton and Pantelides (1994). It enriches a discrete dynamical system with a minimal form of continuous dynamics, while the latter enriches DAEs with a single set of discrete modes.

Cassez and Larsen (2000) consider an extension allowing the derivative of a variable to be 0 or 1, and call this a stopwatch automaton. Alur et al. (1995) define the linear hybrid system, which allows the rate of change to be any constant. These seemingly minor variations can significantly affect some kinds of analysis. For example, the reachability problem for

linear hybrid systems is undecidable but can be solved for timed automata. Finally, these fall under the class of systems allowing general continuous dynamics, generically called hybrid automata. The linear hybrid system in Alur et al. (1995) is actually defined as a restriction of this more general system. See also Henzinger (1996) and Nicollin et al. (1991).

With respect to theoretical expressivity, the LCCA framework we will define is closest to the linear hybrid system. However, our aim is to simplify modeling in practice, and we provide additional features to this end. For example, the discrete modes of two automata might both represent the use of a single resource. We allow finite domain constraints, which could be used to disallow the two automata from being in those modes simultaneously. We also provide modular modeling capabilities by allowing multiple automata. Variable scoping rules assure that modules' meanings are independent of each other. Modularity eases modifications because changes are localized.

Given that a system has been modeled, various questions about that system might be asked. Chutinan and Krogh (2003) discuss techniques for the verification of hybrid systems, Barton (1992) began work on simulation of hybrid systems with fairly general continuous dynamics, reachability for systems with piecewise-constant derivatives is investigated in Asarin et al. (1995), and stability via a generalization of the Lyapunov method is presented in DeCarlo et al. (2000). Krogh (2000) provides a brief survey and references for further investigation.

Possible approaches for the optimization of hybrid systems can be divided into those that employ MP and those that define algorithms directly on the hybrid system model. There is relatively little work in both categories.

Asarin and Maler (1999) show how to design an optimal controller for timed automata. Abdeddaim and Maler (2001) model job-shop scheduling problems with acyclic timed automata. These have been traditionally modeled in the MP framework, but these works define optimization methods independent of MP algorithms. As we do, they also make the case that hybrid automata models allow modeling these problems more naturally than MP. In Abdeddaim and Maler (2002), they extend their work to preemptive job-shop scheduling, which require modeling with the richer stopwatch automata.

Algorithms developed directly on timed or stopwatch automata can be efficient because the structures of these special classes of systems can be exploited. On the other hand, extensions to the modeling class require new algorithms to be developed. A broader class of systems can be modeled in the MP framework, and there is an extensive body of literature and commercial software for optimizing problems posed this way. Bixby (2002) reviews the impressive advances in this area.

So the second approach to optimizing hybrid systems, the one we take, is to transform the hybrid systems models into MP models. Usually these are mixed-integer programs, but Raghunathan and Biegler (2003) show that integer variables can often be eliminated, but at the expense of introducing nonlinearities.

Stursberg et al. (2002) show how an optimal control problem on their hybrid automata formulation can be posed as an MP. Our transformation procedure differs firstly because it operates on the alternative LCCA framework we provide. Additional methods must be considered to transform the additional features, such as finite domain constraints. Secondly,

we focus more on the mechanics of performing the transformation. This makes it more applicable to models as written in practice, not just canonical forms.

Lee et al. (2004) have also employed MP to optimize hybrid systems. Their focus is largely on numerical matters. For example, results, such as sensitivity analysis, on purely continuous systems are complicated by the addition of switching. They show how to compute sensitivities for hybrid systems, allowing efficient application of gradient based algorithms. In contrast, we are focused on the transformation as a symbolic procedure, as opposed to the numerical properties of the resulting formulation.

A somewhat similar distinction exists with the work of Heemels et al. (2001). They show that several hybrid systems modeling frameworks are equivalent. However, this does not provide a systematic procedure for translating models in one of those frameworks to another. Also, they consider only the discrete time case. We provide an exact reformulation for continuous time models at the expense of restricting continuous dynamics to be piecewise linear. Torrisi et al. (2000) describe a software implementation for transforming hybrid systems models into mixed-integer constraints, but this is also for discrete time models.

1.4 Mathematical Programs

Thus far, we have said that we will provide a formal hybrid automata (HA) framework and a systematic procedure for converting these models into mathematical programming (MP) models. But what exactly does it mean to have a formal modeling framework and a systematic procedure? The HA framework we define is formal in the sense that a human reader (with appropriate mathematical proficiency) of the definition would understand it unambiguously. Similarly, the transformation procedure is defined formally enough to assure that a human could know exactly what the steps are.

An improvement would be to make the models comprehensible to a computer and to automate the transformation. This final step of computer implementation is paid less attention in the literature. In fact, the software challenge needs to be considered carefully because it requires an even greater degree of mathematical formality. The definitions of the modeling framework and the transformation must be precise enough to be comprehensible not just by humans but by a machine. Our demand for computable definitions is fulfilled on the MP part of our overall goal.

Type theory is the methodology we employ to resolve this software challenge. In fact, this leads to a more fundamental accomplishment—we provide a logical definition of mathematical programs. The software design specification is merely a by-product of this result. Aside from immediately providing a natural input language, this formulation could also aid in algorithms development. More information about the problem structure is retained, it is possible to treat constraints as objects in a precise way, and different kinds of algorithms can be easily integrated.

In the rest of this section, we review current formulations of mathematical programming both from a mathematical and software perspective. Emphasis is placed on how our use of type theory provides alternatives for both.

1.4.1 Previous Definitions of Mathematical Programs

A mixed-integer linear program (MILP), according to the standard reference text Nemhauser and Wolsey (1999, p. 3), is

$$\max \left\{ cx + hy : Ax + Gy \leq b, x \in \mathbb{Z}^n_+, y \in \mathbb{R}^p_+ \right\} \tag{1.4}$$

This definition characterizes an MILP by the vectors and matrices c, h, A, G, and b. It is compact, and methods from numeric mathematics can be readily applied to it. Virtually all the existing results for MILP were made possible by this definition.

Nonetheless, this matrix form has disadvantages. In the core theory, according to this definition, there are only inequalities and no equations. This is considered adequate because the equation $x = y$ can be written as the two inequalities $x \leq y$ and $x \geq y$. However, rearranging constraints into a canonical form discards valuable information provided by the modeler. The equation $x = y$ immediately requires these two variables to take on the same value, as where, depending on the algorithm used, this inference could take longer from the inequality form. Our formulation will provide several additional mechanisms to retain knowledge about the problem structure. A critical one is provided by indexed constraints, which we discuss shortly.

The general issue is that the matrix-based definition only treats the coefficient matrices c, h, A, G, and b as mathematical constructs. These matrices characterize the objective, constraints, and overall program, but these items cannot themselves be referred to in a formal theory. For example, consider a function f applied as $f(A, G, b)$. Formally this function operates on a numeric space. We interpret the numeric space as characterizing a constraint space, but that interpretation is external to the theory within which f is defined. Thus, the definition of f could not take account of things such as the variable names or the fact that the constraint is an inequality.

Such information is required to define the program transformations we aim for. We need functions that can be truly applied to constraints, for example as $g(Ax + Gy \leq b)$, and this requires defining the constraint spaces that are the domain and codomain of the function.

In the case of MILPs, coefficient matrices could be thought of as characterizing the program to some extent, although we state this is not sufficient to define program transformations. However, a matrix definition cannot work, even for canonical forms, for other constraints commonly used in MP. Raman and Grossmann's (1994) definition of what they

call a generalized disjunctive program (GDP) is

$$\begin{aligned}
&\min Z = \sum_i \sum_k c_{ik} + f(x) \\
&\text{s.t. } g(x) \leq 0 \\
&\bigvee_{i \in D_k} \begin{bmatrix} Y_{ik} \\ h_{ik}(x, c_{ik}) \leq 0 \end{bmatrix}, \quad k \in SD \\
&\Omega(Y) = true \\
&x \in \mathbb{R}^n, \quad c \in \mathbb{R}^m, \quad Y \in \{\text{true}, \text{false}\}^m
\end{aligned} \tag{1.5}$$

followed by the statement that f, g, and h are nonlinear functions and Ω is a Boolean proposition. An external understanding of the terms "nonlinear function" and "Boolean proposition" is assumed.

A complete definition requires stating what class of statements these refer to. Interestingly, it is precisely these two things that cannot be defined with matrices. A matrix form might be provided for polynomial functions, but not for more general nonlinear forms. More obviously, Ω is a Boolean proposition. It contains no numbers whatsoever, so certainly cannot be characterized by a coefficient matrix. Completing the above definition necessarily requires forgoing the matrix perspective.

Furthermore, in practice it is not sufficient to define only canonical forms. MILP models cannot be defined by providing the coefficient matrices of (1.4). A more natural syntax is required, of which the most important feature is indexing. Of course, the above definitions were never meant to provide a syntax. It is considered obvious that for example the constraint

$$x_i = x_{i-1} + y_i \quad \forall i \in \{1, \ldots, N\} \tag{1.6}$$

can be rearranged into the form required by (1.4). A significant flaw lies in this reasoning. Such a rearrangement is a function and defining it first requires that we define its domain, which is this more natural syntax. In other words, the need to provide a formal definition of indexed constraints has not been avoided.

This seemingly innocuous feature introduces several challenges. Exactly what kind of construct is $\{1, \ldots, N\}$? Allowing modelers to employ such an index set implies that we have defined the set of all index sets. In practice, rather sophisticated sets are needed. For example, we might want to write the quantifiers $\forall i \in I, \forall j \in J_i$. Here, the second set J_i depends on the value of i, and J by itself is a function which, when applied to i, returns an index set. Also, index variables i and j are distinct from the MP variables x and y. The two can be used to create a composite term such as x_i. This evaluates to a real number, but what is x by itself? It seems to be a function; it evaluates to a real number when applied to i. Already, we see that MP constraints as written in practice employ mathematical constructs—index sets, functions returning index sets, universal quantifiers, and function variables—not supported by the current definitions.

Perhaps our discussion is making things unnecessarily complicated. The prevailing view

is that indexing has to do with nothing more than a simple substitution procedure. In constraint (1.6), we could simply substitute in each value of i to create N unindexed constraints. This is firstly inefficient because it increases the program size dramatically. It also discards knowledge provided by the modeler. Now, there are simply N unrelated constraints, but the indexed constraint states that there are N constraints of an identical form.

Secondly, this does not avoid most of the issues. Indices are still present after substitution. Constraint (1.6) for $i = 1$ becomes $x_1 = x_{1-1} + p_1$. Now, x_1 can be treated as a single symbol, a name for one variable, but how about x_{1-1}? Clearly, this is supposed to be reduced to x_0, but we have just evaluated $1 - 1$ to 0. So handling indices requires doing mathematics, but it is not clear what system of mathematics this is. It is not plain integer arithmetic. If x is supposed to be indexed by the set $\{1, \ldots, N\}$ or $\{A, B, C\}$, then $1 - 1$ should not evaluate to 0. It should be recognized as an erroneous statement.

In summary, the current matrix based definitions of mathematical programs have several drawbacks. Their numerical focus does not allow treating constraints and whole programs as mathematical objects, which is required to automate program transformations. They provide only canonical forms, which does not allow modeling in practice nor retaining knowledge provided by the modeler. It is difficult or impossible to extend these definitions to nonlinear programs or additional data types such as Boolean.

Some of these concerns, especially the need for a more natural syntax, have been addressed by previous works. Their goal has been to provide computer languages serving as interfaces to the matrix definition of an MP. We review these in the next section but argue that all lack a formal specification, limiting the features and trustworthiness of their software. Following that, we explain how a more formal approach to language design is possible. This simultaneously produces better software and provides an alternative to matrix-based definitions.

1.4.2 Previous Mathematical Programming Languages

The book by Kallrath (2004) provides a comprehensive overview of modeling languages for mathematical programming. Most of the ones we discuss are parts of commercial products with a wide breadth of features, but our focus is only on the modeling language. The language we will provide has several features not available in any of these, but also the previous languages have features not available in ours. The main distinction is that our software results from a mathematical theory. This provides trustworthy software because properties about its execution behavior can be proven.

One of the earliest languages is GAMS by Bisschop and Meeraus (1982). These authors clearly made the case that modeling is a significant challenge and addressed many fundamental concerns. This language is still widely used, attesting to their far reaching vision. For its time, GAMS introduced many elegant features, most notably the use of index sets. Modern demands are being addressed by more recent languages such as AMPL (Fourer et al., 1990) and Mosel (Colombani and Heipcke, 2002). However, although these are the standard references, none defines the syntax of the language being introduced. Their explanations are only through examples.

Syntax definitions that are available are presented for end users, e.g. in the manuals

for GAMS (Brooke et al., 1998), AMPL (Fourer et al., 2003), and OPL (van Hentenryck and Lustig, 1999). Also, Bisschop and Meeraus (1979) provides a syntax for GAMS, but it is included without any discussion in the body of the work. Overall, little effort has been made to provide formal definitions for MP languages.

In the few cases where the syntax is defined, it is of the *concrete syntax*, which is concerned literally with the characters that can be typed into an input file. Actually, it is known in the programming languages literature (Pierce, 2002, p. 53) that a better focus is on the *abstract syntax*, which is concerned with the essential features of a language.

For example, we declare $e_1 + e_2$ to be an expression, given that e_1 and e_2 are. The important point here is not that the + symbol comes in between the two expressions, but rather that there is such a thing as addition and that addition involves two other expressions. We might have notated it as plus (e_1, e_2), which is an identical definition from the perspective of abstract syntax. Another example is $\sum_{i \in S} e$. This is declared to be an expression given that i is an identifier, S is an index set, and e is itself an expression. In the concrete syntax, we in fact resort to ASCII text. The $\sum$ symbol is written SUM and the $i \in S$ cannot be subscripted. These are relatively trivial matters compared to the abstract declaration: there is such a thing as indexed summation.

Concrete syntax is burdened with details that are not essential to a language's definition. This complicates the theory without benefits. The GAMS syntax provided in Bisschop and Meeraus (1979) requires thirty-four syntactic categories, and that was apparently for a simplified version of the language. In contrast, the theory we will provide requires fewer than ten, and produces a language with significantly more advanced features. Of course, eventually we do have to provide a concrete syntax, but this is relatively trivial to do given the abstract syntax. The concrete syntax of our language is discussed in Appendix C.

In all but the simplest languages, syntactical definitions (concrete or abstract) will include nonsensical terms, such as $2 + \texttt{true}$. All software look for such errors in some way. Fourer and Gay (2002) acknowledge this need. They define a syntax for constraints which itself involves expressions, and say that any expression used must be a "valid expression" (p. 332). However, neither they nor any of the works we have referenced provide a method for determining the valid expressions.

We can only surmise that developers have implemented some ad hoc procedures for validating syntax. What is actually needed is an exposition of the language's type system. The types of the language must be declared and terms of the language categorized into those types. A language definition cannot be considered complete without knowledge of its type system. Without one, we do not know its admissible syntax nor the nature of the objects that comprise it. With one, we have a precise definition of the set of programs.

Various operations are of importance on such a set. We explained above that program transformations are not supported by the theory of MP, but given their importance, several software have attempted to implement them. Unfortunately, the results are often erroneous due to the lack of a supporting theory.

OPL, a leading MP software, for example allows expressing the disjunctive constraint

$$\left(x_1 + x_2 \leq 9 \times 10^5\right) \vee \left(x_1 + x_2 \geq 10^6\right) \tag{1.7}$$

where $x_1, x_2 \in \mathbb{R}$, and no bounds on x_1 nor x_2 are provided. It erroneously transforms this into the MILP constraints

$$x_1 + x_2 + 10^5 y \leq 10^6 \tag{1.8a}$$

$$x_1 + x_2 + \left(2 \times 10^6\right) y \geq 10^6 \tag{1.8b}$$

where $y \in \{0, 1\}$. The exact OPL input and output files are given on page 207. By setting $y = 0$ and $y = 1$, it is easy to see that these constraints do not represent the same region as the disjunctive constraint, not even approximately. The reason is a bound is assumed when not provided. Had smaller constants been chosen, the transformation would still be incorrect but the error smaller. LogMIP makes a similar error in the example provided on page 208. It converts an unbounded optimization problem into a bounded one.

These could be attributed to minor programming errors, but we believe the issue is more significant. The literature lacks a clear explanation of what variable bounds are and how this information can be used to test whether certain transformations are applicable to a given constraint. Our work fills in this gap, and our software clearly informs a user that bounds must be provided prior to converting the above disjunctive constraint.

Aside from lacking a type system and errors in program transformations, there is also little attention paid to more elementary, but crucial, operations. As an example, we discuss the issue of variable capture.

Vecchietti and Grossmann (2000) describe a procedure for putting constraints into conjunctive normal form (CNF) in preparation for the transformations they implement in LogMIP. Equation (9) of their work is

$$\left(\bigvee_{i \in I} g_i\right) \vee \left(\bigwedge_{j \in J} f_j\right) \tag{1.9}$$

where g_i and f_j are constraints[2] indexed by i and j, respectively. They then state that this can be converted into[3]

$$\bigwedge_{j \in J} \left(\bigvee_{i \in I} g_i \vee f_j\right) \tag{1.10}$$

By subscripting g and f, it is implied that these are the only subscripts involved in the constraint, and the above conversion is correct in this case.

However, the subscripting restrictions are excessive. In practice, it would be reasonable for g to also involve the subscript j. Unfortunately, the above transformation is then not applicable. The j in g would be unrelated to the j in f because the latter is locally scoped, but the transformation would result in both j's coinciding. This problem is called variable capture because the j in g has erroneously become captured in the wrong scope.

Such an error probably does not occur in the LogMIP software because it is an extension of GAMS, and, in GAMS, index variables can only be introduced in a way that makes them globally visible. But this means the conversion above is defined on a syntax different from the

[2] We have slightly simplified the notation from their original.
[3] We have corrected a typographical error. The I and J are mistakenly accented in their publication.

one actually implemented because the indexed operators above do provide local scoping. We cannot thus be confident of what the implemented transformation is and whether it handles critical details such as variable capture.

The above discussion makes clear that existing MP modeling languages lack a formal foundation. Often, even the syntax for a language is not provided. When it is, it is always the relatively unimportant concrete syntax. Never has an MP language's type system been defined in the literature. It is easy to find examples of program transformations giving incorrect answers. Other operations are either not defined at all or are discussed by example only. The examples are often stated for a syntax different than the one the operations are implemented for.

On the other hand, most of the languages mentioned are successful commercial products. Our work can be viewed as supporting these efforts by providing a theory for developing such software. This is firstly necessary because the informal design methods currently in use are unable to provide the more advanced features being continuously demanded. Furthermore, formally defined software can be analyzed. We can prove that our software correctly performs the computations we claim it to.

We will provide a formal theory for a mathematical programming language. The syntax will be defined precisely, and its type system declared. We will then go one step further; a semantic interpretation of the syntax is provided. But a formal syntax and semantics provides a mathematical system which can be understood in its own right. In other words, designing a computer language, when defined rigorously, is equivalent to inventing a mathematical system[4]. Our definition of a language will provide not only a theoretical foundation for the efforts of software developers but also serve as an alternative to current matrix-based definitions of MP. This close connection between computation and mathematics is espoused by type theory, the theory employed to provide our language's definition.

1.5 Type Theory

The distinction between a language easy to use in practice and the formal definition of a mathematical program—sometimes called the modeler's and algorithm's form—is eliminated by providing a formal definition of an elegant language. Henceforth, language design is to be understood as equivalent to invention of a novel mathematical system, not merely specification of a computer file format. To avoid confusion, we will often say we are defining a logic. A logic, used as a noun, is a mathematical system with a well defined syntax and semantics.

Several methodologies for inventing logics exist. Mathematical logic is the general discipline of concern and can be thought of as a discipline for studying reasoning in a formal way (Curry, 1963, p. 2). Specific systems for carrying out such a formalization include propositional and predicate logic, also called zeroth- and first-order logic, respectively. These systems can be studied abstractly but are instantiated to provide specific theories. For instance, $\exists x . A \vee B$, read "there exists x such that A or B is true", is a statement in first-order

[4]However, the popular programming languages, such as C/C++, Java, and Fortran, do not adhere to this standard. Amongst widely used languages, ML, the language in which this work is implemented, comes closest. Its definition is provided in Harper, Milner and Tofte (1989).

logic. A specific first-order logic is obtained if we let x range over the reals, and let A and B be equations. In that case, the statement is a constraint in a disjunctive program (the existential quantifier is not written in current practice, but it is implicitly present).

Approaches along this line have been taken to define languages for constraint programming (CP), a discipline increasingly associated with mathematical programming (MP). For example, the constraint logic programming scheme of Jaffar and Lassez (1987) provides a system which can be instantiated with a specific domain of discourse. Saraswat (1989) extended this into a class of languages for concurrent constraint programming. CP has its roots in logic, and a more formal approach to language design is clearly evident in these works as compared to MP languages. As a result, they are able to treat constraints as objects, a critical need for CP algorithms. We too wish to treat constraints as objects for the purpose of formalizing program transformations. Both areas require this to legitimately claim a language definition.

Richer logics can be invented with axiomatic set theory. Axiomatic set theory is very powerful because it allows defining constructs of a very general nature, but this is not necessarily an advantage. Famously, Frege's (1893) original formulation of set theory turned out to be inconsistent. Roughly, this was because sets could be too large; they could contain themselves leading to a "vicious circularity". Certain restrictions to the axioms of set theory do provide a consistent theory. However, Bertrand Russell circumvented the problem by another strategy. In Russell (1903), he explains how requiring set definitions to be stratified avoids the circularity causing trouble in naive set theory. This is called type theory, and is the subject of Whitehead and Russell's (1910) extraordinarily influential *Principia Mathematica*.

Type theory restricts the contents of sets by distinguishing between different types of objects (e.g. numbers, functions from numbers to numbers, sets of numbers). The set of natural numbers for instance could not contain itself because it can only contain numbers, not sets. Categorizing objects in this way is a very natural restriction that coincides with the nature of human thought, both in everyday life and in mathematics. Even prior to the development of type theory, mathematicians kept in mind that there are different types of objects. A function applied to a number would not be applied to a set because numbers and sets are different types of things. Type theory makes this categorization explicit and demands it.

Type theory is a higher-order logic much more powerful than first-order logic. Strictly, axiomatic set theory is even more powerful. That is important for certain studies, but type theory is often preferred for its elegance. According to Andrews (2002)—which provides an excellent introduction to mathematical logic—it is "an accident of intellectual history that at present most logicians and mathematicians are more familiar with axiomatic set theory than with type theory" (p. xii).

Several improved versions of type theory have been presented since Russell's original formulation. Stratifying set definitions to categorize objects into their types is always the basic theme. The theory of mathematical programming we provide incorporates this technique to define a language with richer data types. The definition of MILPs given in (1.4) acknowledges only the real and integer data types. MP languages however speak of indexed variables. It is said that x_i is a real variable, but we must also understand what x is by

itself. We formalize such a variable by defining it to be of function type, a mapping from an index set to the reals. An enriched type system is the essence of a more elegant language.

The MP language will rely on a separately defined indexing language. From the perspective of language design, indexing is far more complex than mathematical programming. MP types are fixed objects—the elements of the real, integer, and Boolean types are set. The indexing language requires *dependent types*. For example, the type $[1, i]$ represents the integers from 1 through i. In contrast to the MP types, the elements of $[1, i]$ depend on the value of a variable.

The indexing language is an instantiation (of most) of a particular type theory called *intuitionistic type theory*, invented by Martin-Löf (1984). Amongst other things, intuitionistic type theory enriched previous versions of type theory by introducing dependent types. Dependent function types allow the codomain of a function to depend on the particular argument the function is applied to. For example, a function f could be defined such that $f(1)$ returns a value in the set $\{A, B, C\}$, but $f(2)$ returns a value in $\{D, E\}$. Product types are similarly generalized. The type $I \times J$, which contains pairs (i, j), could be defined such that the allowed value of j depends on what i is. Often, our conceptualization of a system incorporates exactly such types, as some examples will demonstrate. Indeed, dependent types are commonly used in models defined on paper, where imprecise mathematical notation is accepted. Our dependent type theory makes such models comprehensible to a computer.

The indexing language is an especially interesting application of dependent type theory. It is a rare example of a useful language that contains only finitary types. Most languages require infinite types such as real and integer. However, it is the very essence of an index set that it contain a finite number of elements. As a result it is possible to define an ontological type checker. This defines a construct to be well formed precisely when the meaning of that construct can be understood. This is in stark contrast to the syntactically driven type checker of the MP language. This is the usual kind and type checking is prior to semantics. Further discussion on this matter is provided at the beginning of Chapter 6.

As the name suggests, intuitionistic type theory follows the program of intuitionism instigated in the mathematician Brouwer's (1907) thesis. Intuitionism is a form of constructivism. Dummett (1977) provides an authoritative account of intuitionism, and the first chapter of Troelstra and van Dalen (1988) provides an overview of constructivism in general. There is significant debate between followers of this view of mathematics and the classical or platonic view. Moderates accept that both views can coexist. Our work attempts only to reflect the current practice of mathematical programming. Clearly, infinite objects, i.e. the reals, are currently treated classically, but interestingly, existing algorithms seem to take a constructive view of solving an MP. We expand on this after explaining how solving an MP corresponds to executing a program.

1.6 Programming Languages

The constructivist approach is especially relevant for providing computational theories of mathematics. Thus, it is particularly important for the design of programming languages,

which are meant to be executed on a computer. Thus far, we have spoken of MP as a modeling language and as a logic. Ultimately, we would like to view it also as a programming language. Designing programming languages has been one of the primary applications of type theory. See for example Pierce (2002), Harper (2005), and Harper et al. (1989).

Treating mathematical programs as computer programs appears to be in direct opposition to the prevailing view. The word "programming" in mathematical programming results from its original use in linear programming (LP). LPs were originally used to optimize logistics schedules, which were called programs. That the word programming in this context is unrelated to computer programming is an often mentioned point. We do not mean to contradict this distinction. Our explanation of MP as a programming language is unrelated to the occurrence of the word programming in mathematical programming.

MPs have not been considered programming languages also because constraints are thought of as declarative statements while programming languages as procedural. This view is manifested in current software designs, e.g. OPL provides a modeling language for declaring an MP and a separate scripting language for specifying algorithms to solve it. This dichotomy is somewhat obviated by the functional programming paradigm, which has its roots in the λ-calculus (Barendregt, 1984). The language Mosel (Colombani and Heipcke, 2002) purportedly follows this view and correspondingly allows intermixing model and solve statements.

We explicate the relationship further. It is possible to view a declarative statement as a programming language statement that must be executed. This clarifies the intention of the so-called procedural aspects of current MP software. They are statements in the overall language directing the execution of the mathematical programming sub-language.

Executing a mathematical program means solving the optimization problem it expresses. We define the meaning of solving an MP in a logical manner; a semantic interpretation of the syntax of an MP is provided. For example, after stating that $x = y + 1.0$ is a constraint, we will state what it means for the constraint to be true or false. Following the constructive view, truth is understood through proof. It is only when we have defined what a proof of truth is that we understand what it means for a constraint to be true.

Our treatment of semantics is less formal than of syntax. The required proofs are explained only conceptually (we do not provide a language of proof terms). This will be sufficient to gain some interesting insights. We will see that current algorithms often take a constructive view of proof. This is most evident for disjunctive constraints $A \vee B$, one of the main points of contention between classicists and constructivists. Constructively, the constraint $A \vee B$ is considered to be satisfied only when we can prove that either A or B is true. Classical mathematics accepts proof by contradiction, which could allow concluding $A \vee B$ to be true without in fact knowing either that A is true or that B is true. All MP algorithms we know of interpret disjunction constructively.

Making this point explicit clarifies what the output of MP software should be. Solution reporting is currently treated in completely ad hoc manners. In fact, what the user is seeking is a proof of why their particular mathematical program has the solution it does. The output should state not only what the optimal solution is, but why it is so. This is precisely the constructive view of mathematics. Our definition of the semantics of an MP

is a first step towards formalizing the algorithmic aspects of MP.

Note however that we will not provide an executable MP language. Firstly, this is because solving an MP is the purview of virtually all the rest of the MP literature. Our goal is only to fit that work into a logical formulation of MP. Secondly, doing so would require us to address the issue of computation on the reals, which is well beyond the scope of this work. Even in future, the best we can hope to do is imagine executing a mathematical program approximately, in such a way that it gives a solution reliably close to the true solution.

1.7 Dissertation Overview

Hybrid Systems

- Chapter 2 defines our LCCA modeling framework. We show that systems with mixed discrete-continuous dynamics can be elegantly modeled in this framework.
- Chapter 3 presents a method for transforming LCCA models to mathematical programs, thereby enabling the application of existing algorithms to our novel modeling framework.

Logical Mathematical Programs

- Chapter 4 provides a type theoretic (logical) formulation of mathematical programs. An introduction to the basic concepts of type theory is provided.
- Chapter 5 defines the subset of MP that we consider to be MIP. Then, a compiler from MP to MIP is defined. Not all MPs can be expressed as an equivalent MIP. We present a sufficient, but not necessary, precondition that the compiler must satisfy.

Indexed Mathematical Programs

- Chapter 6 defines a language for declaring complex index sets in an intuitive manner. This is a logic of finitary types, separate from the MP language.
- Chapter 7 combines this indexing language with the unindexed MP language to define a logic of indexed programs.
- Chapter 8 defines a compiler on indexed programs, analogous to that of Chapter 5.

Chapters 7 and 8 subsume the theories of Chapters 4 and 5, respectively. They extend the former theories with indexing. There is thus some repetition in the definitions but the discussion differs. The former chapters emphasize how type theory can provide a novel formulation of mathematical programs and how this allows automating tasks currently performed manually. The latter focus on the additional theory required due to indexing.

An incremental presentation was considered necessary because indexing complicates the theory extensively. It is by far the greatest challenge addressed in this work. Chapters 4 and 5 are intended for an audience unfamiliar with type theory, while Chapters 7 and 8 assume prior experience in the area.

Closing Chapters

- Chapter 9 presents an application tying together the results of the dissertation.
- Chapter 10 concludes with an assessment of our contributions and some ideas for future work.

Appendices

- Appendix A reviews basic concepts from mathematical programming, especially the various methods for transforming constraints.
- Appendix B presents a meta-logic with built-in support for variable scoping structure. This saves us the trouble of defining numerous operations related to variables.
- Appendix C discusses the concrete syntax of our language. The chapters present all theory with respect to the abstract syntax, but examples show our software's actual input and output, which is in concrete syntax.
- Appendix D includes data and input and output files referenced in the main body.

Finally, a summary of notation is provided beginning on page 214 and a list of acronyms on page 220.

Chapter 2

Modeling Hybrid Systems

Many processes in the chemical industry exhibit mixed discrete-continuous dynamics, called hybrid dynamics. An example is shown in Figure 2.1, which depicts a tank with two inlet streams and one outlet. The inlets are each governed by a hybrid process. Process α depicts a pump which can be in one of two discrete modes: on or off. Similarly, process β can be pumping at either a high or low setting.

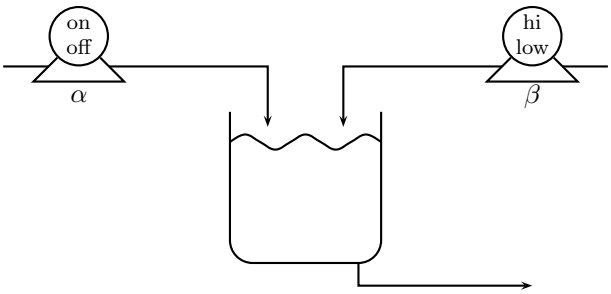


Figure 2.1: Schematic of switched flow process.

In a large system, e.g. a full chemical plant, it would be necessary to break down the modeling task into smaller pieces. The sub-model defining process α should be unaffected by changes to the sub-model for process β. The two processes are coupled however, through their mutual contribution to the material level in the tank. A modeling framework should require this coupling to be expressed in a way that maintains a separation between the sub-models.

The linear coupled component automata (LCCA) framework we introduce does this. Each linear automaton represents a single hybrid process. Certain variables, e.g. the flow input by process α, are local to each automaton, meaning they cannot be used in any other part of the model. Other variables, e.g. the material level in the tank, are global, meaning they are visible to all processes. However, there are restrictions on how the global variables can be employed. In particular, a component automaton cannot directly specify the value of a global variable; it can only specify its own contribution to that value.

In the presence of hybrid dynamics, modularity introduces a complication relating to the

timing of discrete events. Discrete events can occur only at certain points in the timeline, called event points. However, naturally, at an event point, not all automata should have to make a discrete transition. Thus, time must be able to progress through an event point in such a way that a discrete event is in fact not required. This matter is resolved with what we call a dummy transition.

The LCCA framework we define follows the style of much recent work in the area of hybrid systems, reviewed in the Introduction chapter. In terms of theoretical expressivity, it is similar to the linear hybrid automata of Alur et al. (1995). In the next chapter we will show that in fact systems expressed in LCCA can be expressed as a mixed-integer linear program (MILP). Our aim thus is not to provide a framework more expressive in theory. Rather it is to facilitate modeling in practice.

2.1 Preliminaries

The LCCA framework makes use of a hybrid timeline and various constraint forms. We define these in the current section.

2.1.1 Hybrid Timeline

For continuous systems, the timeline is simply an interval of the real numbers, but a different model of time is needed for hybrid systems. Discrete dynamics occur instantaneously and the timeline must allow specification of two values at certain time points, called event points.

Let $\mathbb{N} = \{1, \ldots, n\}$ for some constant n. Lygeros et al. (1999) define a hybrid timeline[1], depicted in Figure 2.2, as an ordered sequence of intervals $\mathcal{T} = \{[t_i^s, t_i^e]\}_{i \in \mathbb{N}}$ such that

- $t_i^s \leq t_i^e$ for $i \in \mathbb{N}$, and
- $t_i^e = t_{i+1}^s$ for $i \in \mathbb{N} \backslash \{n\}$.

The interpretation is that all discrete variables are constant during each interval. Only continuous variables evolve within intervals. Discrete variables change their values at the boundaries between intervals, the event points. Let $\Delta t_i = t_i^e - t_i^s$ denote the length of interval i.

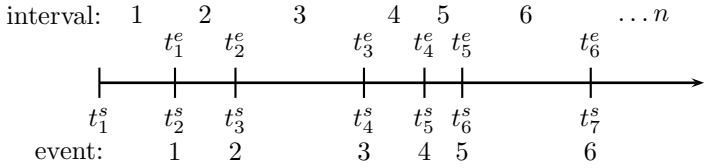


Figure 2.2: Hybrid timeline.

Despite this alternative timeline for hybrid systems, the definition of dynamic variables in the existing literature (e.g. Aubin et al., 2002) has followed that of purely continuous systems, i.e. a dynamic variable is defined on the domain $\mathbb{R}$. This complicates the mathematics

[1] We have restricted their definition to the $n < \infty$ case.

because a variable can take two values at the event points, requiring them to be treated as more general mappings than functions. However, a functional form can be maintained if the above timeline definition is slightly rearranged. We define the hybrid timeline

$$\mathbb{T} = \{(i,t) : t \in [t_i^s, t_i^e] \in \mathcal{T}\}.$$

With this definition, a single time point is a pair (i,t). We strictly maintain this interpretation; any reference to t by itself is considered incomplete.

There is a total order relation $\preceq$ on $\mathbb{T}$, as expected of a timeline. The time point (i,t) precedes or is equal to (i',t'), denoted $(i,t) \preceq (i',t')$, if and only if $i \leq i'$ and $t \leq t'$. If $(i,t) \not\preceq (i',t')$, then it must be that $(i',t') \preceq (i,t)$. If both $(i,t) \preceq (i',t')$ and $(i',t') \preceq (i,t)$, then $(i,t) = (i',t')$.

The set $\mathbb{N}\backslash\{n\} = \{1,2,\ldots,n-1\}$ can be interpreted as the set of event points. Our convention is that the i^{th} event point occurs at the *end* of interval i. Event point i coincides with two time points: (i, t_i^e) and $\left(i+1, t_{i+1}^s\right)$. Only the integer component of time changes when an event occurs. The real component at both times is $t = t_i^e = t_{i+1}^s$.

Events are said to occur instantaneously, but this is only with respect to the real component of time. The integer component of time does progress. The concept of instantaneous has been captured by a finite time change in our definition. Precisely, an event occurs over zero units of real time and one unit of integer time. As a consequence, events do not occur at the initial $(1, t_1^s)$ and final (n, t_n^e) time points.

Time is usually considered continuous or discrete, but additional possibilities exist in a hybrid timeline. The formulation provided is a continuous timeline with event points. If $n = 1$, then we have a continuous time formulation without events, which is the usual model of time. If $n > 1$ and interval lengths are fixed, the system restricts the timing of event points. However, there could still be continuous evolution within an interval. Finally, time could be discretized within each interval, whether or not interval lengths are fixed, to accommodate the numerical solution of differential equations. Discretization will however not affect us because we consider only differential equations simple enough to be symbolically integrated. In summary, a hybrid timeline can be discrete or continuous and can either restrict event times to fixed points or not.

2.1.2 Constraints

Constraints on real and discrete variables will be needed in the overall modeling framework. Here, we discuss the meaning of continuous and discrete dynamic variables and define some notation needed subsequently.

A real valued dynamic variable is a function from the timeline to the reals, $X : \mathbb{T} \to \mathbb{R}$. Our modeling framework allows use of equations and inequalities: $=$, $\leq$, and $\geq$. Strict inequalities are not allowed because they are also not allowed in MIPs, to which we wish to convert our models.

Continuous variables change infinitesimally in an infinitesimal amount of time, but their values can also jump instantaneously at an event point. Consider an event occurring from time (i,t) to $(i+1,t)$. The instantaneous change can be defined by an algebraic equation,

e.g. $X(i+1,t) = X(i,t) + 1$ would increment the value of X by 1 at event i.

In purely continuous systems, it is customary to refer to "a set of constraints". Really, what is meant by this is a conjunction of constraints. For example,

$$X(i+1,t) = X(i,t) + 1 \quad \wedge$$

$$Y(i,t) = X(i,t)$$

is usually considered two constraints and the conjunction symbol $\wedge$ is not shown. We often call this a single constraint (which happens to be a conjunction of two other constraints).

Discrete variables come in various forms. MIP allows integer variables, and GDP includes Boolean logic. We consider variables taking values from a finite set, e.g. $\mathbb{Q} = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$, which are often called set valued or finite domain variables. `A`, `B`, and `C` are finite domain constants, just as 1, 2, and 3 are integer constants.

Given a finite set $\mathbb{Q}$, we can now define a dynamic finite domain variable $Q : \mathbb{T} \to \mathbb{Q}$. However, by definition, discrete variables do not evolve within an interval. The value of $Q(i,t)$ depends only on the interval number i, making the t superfluous. It suffices to define dynamic discrete variables as functions on the set of intervals, $Q : \mathbb{N} \to \mathbb{Q}$. Equations can be used to restrict the values taken by Q over time, e.g. $Q(2) = \mathtt{B}$ sets Q in the second interval to the value `B`. Also, the values between intervals can be related, e.g. $Q(i) = Q(i+1)$ forces Q's value to remain unchanged from interval i to $i+1$.

Of course, multiple discrete variables might be needed. Superscripts are used to refer to different finite sets, e.g. $\mathbb{Q}^\alpha = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $\mathbb{Q}^\beta = \{\mathtt{D}, \mathtt{E}\}$. The corresponding dynamic variables are named with the same superscript. So Q^α is understood to be a function from $\mathbb{N} \to \mathbb{Q}^\alpha$. The equation $Q^\alpha(i) = \mathtt{D}$ would be erroneous.

Constraints on reals can only be connected by conjunction, but equations on finite domain variables can be connected with the logical operators negation $\neg$, disjunction $\vee$, and conjunction $\wedge$. For example, perhaps Q^α taking the value `B` requires the use of some resource, and that same resource is required if Q^β is equal to `D`. The constraint

$$\neg\left(\left(Q^\alpha(i) = \mathtt{B}\right) \wedge \left(Q^\beta(i) = \mathtt{D}\right)\right)$$

assures that these values are not taken at the same time. Another example is

$$\left(Q^\alpha(i) = \mathtt{B}\right) \vee \left(Q^\alpha(i) = \mathtt{C}\right)$$

which requires Q^α to take either the value `B` or `C` in interval i.

The modeling framework defined in the next section allows constraints in the above forms but requires various restrictions on which variables can be used and the time points at which the variables can be evaluated. Some notation will allow stating these restrictions compactly. Let $\mathcal{L}$ denote the set of all constraints in the forms discussed above, and let $\mathbf{X}$ be a set of dynamic real variables and $\mathbf{Q}$ a set of dynamic finite domain variables. Also, let $\mathbf{X}(i,t)$ and $\mathbf{Q}(i)$ mean each of the variables in the sets are evaluated at time point (i,t). Finally, $\mathcal{L}(\mathbf{X}(i,t))$ is the set of constraints involving only variables in $\mathbf{X}$ evaluated at time (i,t), and similarly for $\mathcal{L}(\mathbf{Q}(i))$.

The following font conventions have been used:

- plain font represents a single variable, e.g. i, t, X, Q
- bold font represents a set of variables, e.g. $\mathbf{X}$, $\mathbf{Q}$
- blackboard bold font represents a space of values, e.g. $\mathbb{N}$, $\mathbb{R}$, $\mathbb{T}$, $\mathbb{Q}$
- calligraphic font represents more complex mathematical objects, e.g. $\mathcal{L}$.

2.2 Linear Coupled Component Automata

With these preliminary concepts in place, we can now define the LCCA framework. Conceptually, the system consists of a set of component automata, so called because they are components of an overall system. Each automaton has a set of discrete modes associated with it, and a dynamic finite domain variable specifies the mode the automaton is in over time. There are overall real valued system variables, and each automaton specifies its contribution to how these variables evolve. Automata cannot directly specify the rate of change of a system variable, but the values of system variables can prohibit or require discrete transitions in the automata. These are natural restrictions that support modular modeling.

A dynamical system consists of a timeline, variables, and a specification of how these variables evolve. Precisely, we define a linear coupled component automata model as the 5-tuple

$$(n, G_t, \mathbf{X}, Aut, G_V) \tag{LCCA}$$

where n is an integer, G_t is a constraint on the timeline variables, $\mathbf{X}$ is a set of dynamic real variables, Aut is a set of component automata, and G_V is a constraint coupling the component automata.

The integer n specifies the number of intervals in the timeline. Given n we take the timeline to be $\mathcal{T} = \{[t_i^s, t_i^e]\}_{i\in\mathbb{N}}$, where $\mathbb{N} = \{1, \ldots, n\}$. We can also refer to the timeline in the form $\mathbb{T}$, which is defined in terms of $\mathcal{T}$ in section 2.1.1. Let $\mathbf{t} = \cup_{i\in\mathbb{N}} \{t_i^s, t_i^e\}$ be the set of timeline variables.

G_t is an element of $\mathcal{L}(\mathbf{t})$, allowing constraints on the timeline variables. For example, interval lengths could be fixed to a constant by requiring $t_i^e - t_i^s = k$ for all i. For optimization purposes, an upper bound on the time horizon will be required. This can be given by the constraint $t_n^e \leq T^{\max}$.

Each automaton $a \in Aut$ is itself a 5-tuple of the form

$$(\mathbb{Q}, \bar{\mathbf{x}}, \hat{\mathbf{x}}, F, Arc) \tag{component automaton}$$

where:

- $\mathbb{Q}$ gives the discrete modes of the automaton, and we let $Q : \mathbb{N} \to \mathbb{Q}$ give the discrete mode in each interval.
- $\bar{\mathbf{x}}$ is a set of given rates, the values of which can vary by mode. Each $\bar{x} \in \bar{\mathbf{x}}$ is a function from $\mathbb{Q} \to \mathbb{R}$. These will be used in differential equations governing the continuous evolution of $\mathbf{X}$.

- $\hat{\mathbf{x}}$ is a set of jump variables, whose values are dependent on the event number. Each $\hat{x} \in \hat{\mathbf{x}}$ is a function from $\mathbb{N}\backslash\{n\} \rightarrow \mathbb{R}$. These will be used in algebraic equations governing the discrete evolution of $\mathbf{X}$.

- $F : \mathbb{Q} \rightarrow \mathcal{L}(\mathbf{X}(i,t))$ is an invariant, a constraint that must be satisfied during continuous evolution. The constraint can depend on the mode of the automaton. Formally, the condition is enforced as $F(Q(i))$ for all $(i,t) \in \mathbb{T}$. At time (i,t), the automaton's mode is $Q(i)$. Thus, $F(Q(i))$ gives the desired constraint for the active mode.

- $Arc \subseteq \mathbb{Q} \times \mathbb{Q}$ is a set of transitions. Associated with each transition $(q, q') \in Arc$ is a *guard* $\gamma_{(q,q')} \in \mathcal{L}(\mathbf{X}(i,t))$ and a *reset* $\rho_{(q,q')} \in \mathcal{L}(\mathbf{X}(i,t) \cup \hat{\mathbf{x}}(i))$. The transition can be made at event point i only if the guard holds at time (i,t), and, if the transition is made, the reset is also enforced. There exists at most one transition from any q to q'.

  It is required that a dummy transition (q,q) exists from every $q \in \mathbb{Q}$ to itself. On this transition, the guard $\gamma_{(q,q)}$ is set to a trivially satisfied constraint such as $1 = 1$, and the reset $\rho_{(q,q)}$ is $\wedge_{\hat{x} \in \hat{\mathbf{x}}}(\hat{x}(i) = 0)$. Dummy transitions are required because the mathematical model forces all automata to transition at an event point, but this is a superficial physical requirement. The dummy transition enables an automaton to progress through an event point in a manner that has no effect.

In the above, an automaton has been generically denoted by the tuple $(\mathbb{Q}, \bar{\mathbf{x}}, \hat{\mathbf{x}}, F, Arc)$. Superscripts are used to refer to multiple automata, e.g. automaton a consists of the elements $(\mathbb{Q}^a, \bar{\mathbf{x}}^a, \hat{\mathbf{x}}^a, F^a, Arc^a)$. The discrete variable associated with this automaton is Q^a. Let $\mathbf{Q} = \cup_{a \in Aut} Q^a$ be the set of discrete variables in an LCCA model, just as $\mathbf{X}$ is the set of continuous variables.

Variables $\bar{\mathbf{x}}^\alpha$ and $\hat{\mathbf{x}}^\alpha$ are local to automaton α. Another automaton β cannot make any reference to these variables, only to its own $\bar{\mathbf{x}}^\beta$ and $\hat{\mathbf{x}}^\beta$. Separating the variable name space requires modeling to be done in a modular fashion. In contrast to existing hybrid automata modeling frameworks, differential equations are not associated with the discrete modes of our component automata. They specify only rates $\bar{\mathbf{x}}$. Similarly, resets do not specify discontinuous evolution; they dictate only the values of $\hat{\mathbf{x}}$. Component automata are so named because they do not represent a dynamical system on their own; they are useful only as components of the overall LCCA system.

The last element of the modeling framework is G_V. This is where the dynamical equations are specified. All local and system variables can be used. G_V can include several types of constraints:

- differential equations of the form

$$\frac{\mathrm{d}X(i,t)}{\mathrm{d}t} = \sum_{a \in Aut} \bar{x}^a(Q^a(i)) + k \qquad \forall (i,t) \in \mathbb{T} \tag{2.1}$$

  where k is some constant. Thus, the overall rate of change of X is potentially dependent on the active modes of all automata,

- discontinuity equations of the form

$$X\left(i+1, t_{i+1}^{s}\right)=X\left(i, t_{i}^{e}\right)+\sum_{a \in Aut} \hat{x}^{a}(i)+k \qquad \forall i \in \mathbb{N} \backslash\{n\} \tag{2.2}$$

  where k is some constant. Discontinuity equations are the discrete analog of differential equations. A flow rate $\bar{x}$ specifies an infinitesimal change in an infinitesimal amount of time. Similarly, a jump variable $\hat{x}$ specifies a finite change over an instantaneous step in time, i.e. over an event occurring from time $\left(i, t_{i}^{e}\right)$ to $\left(i+1, t_{i+1}^{s}\right)$,

- finite domain constraints in $\mathcal{L}(\mathbf{Q}(i))$ or real constraints in $\mathcal{L}(\mathbf{X}(i, t))$. These are constraints that must always hold, independent of the active mode of automata, and

- initial conditions from $\mathcal{L}(\mathbf{Q}(i))$ or $\mathcal{L}(\mathbf{X}(i, t))$, where (i, t) and i are specific time values. For example, $X\left(1, t_{1}^{s}\right)=0.0$.

Mathematical notation has been used thus far, but automata are customarily depicted graphically, e.g. see equation (2.3b). A box is drawn for each discrete mode, and the name of the mode is written within the box. This is followed by the values of variables $\bar{\mathbf{x}}$ and invariant F for that mode. Next, for each transition $(q, q') \in Arc$, an arrow is drawn from box q to q'. The guard $\gamma_{(q,q')}$ is written near the tail of the arrow and reset $\rho_{(q,q')}$ near the head. Dummy transitions are not shown because their definition is fixed. LCCA models are thus provided with a mixture of graphical and textual declarations.

2.3 Hybrid Trajectories

In the previous section we defined the LCCA framework, but it is not yet clear what it means to solve a problem posed in this framework. Conceptually, the solution of a dynamical system is a trajectory. A trajectory is a mapping from a timeline to the set of possible values for all state variables, which are $\mathbf{X}$ and $\mathbf{Q}$ in our modeling framework. Also in our system, the timeline is itself variable because event times are not fixed. A trajectory is feasible for a given model if it obeys the constraints of that model.

For a hybrid system, a trajectory consists of alternating sequences of continuous and discrete evolution. Let $\chi(i, t)=(i, t), \mathbf{X}(i, t), \mathbf{Q}(i, t))$ be a triplet consisting of a time point and values of all continuous and discrete variables at that point. A *continuous trajectory* is an interval $\left[\chi\left(i, t_{i}^{s}\right) \longrightarrow \chi\left(i, t_{i}^{e}\right)\right]$, where the integer component i of time does not change. A *discrete step* is a pair $\langle\chi(i, t) \mapsto \chi(i+1, t)\rangle$, where we have used an arrow $\mapsto$ to separate the two elements of the pair. During a discrete step, the real component of time t does not change.

Finally, a *hybrid trajectory* is an ordered set of continuous trajectories

$$\xi=\left\{\left[\chi\left(i, t_{i}^{s}\right) \longrightarrow \chi\left(i, t_{i}^{e}\right)\right]\right\}_{i \in \mathbb{N}}.$$

The discrete steps can be formed from the given continuous trajectories. They are the pairs $\left\langle\chi\left(i, t_{i}^{e}\right) \mapsto \chi\left(i+1, t_{i+1}^{s}\right)\right\rangle$ for all $i \in \mathbb{N} \backslash\{n\}$. Also, a timeline is implied by ξ. It is simply $\mathcal{T}=\left\{\left[t_{i}^{s}, t_{i}^{e}\right]\right\}_{i \in \mathbb{N}}$, and this can be reformulated into $\mathbb{T}$ as discussed in Section 2.1.1.

Given an LCCA model $(n, G_t, \mathbf{X}, Aut, G_V)$, a trajectory ξ is said to be feasible for that model if

- the timeline $\mathcal{T}$, or equivalently $\mathbb{T}$, implied by ξ is feasible,
- all continuous trajectories $[\chi(i, t_i^s) \to \chi(i, t_i^e)]$ are feasible, and
- all discrete steps $\langle \chi(i, t_i^e) \mapsto \chi(i+1, t_{i+1}^s) \rangle$ are feasible.

The timeline $\mathcal{T}$ is feasible if

- $t_i^s \leq t_i^e$ for all $i \in \mathbb{N}$,
- $t_i^e = t_{i+1}^s$ for all $i \in \mathbb{N} \setminus \{n\}$, and
- G_t is satisfied.

If $\mathcal{T}$ is feasible, its corresponding $\mathbb{T}$ is feasible. Each continuous trajectory $[\chi(i, t_i^s) \to \chi(i, t_i^e)]$ is feasible if

- $\mathbf{Q}(i, t_i^s) = \mathbf{Q}(i, t') = \mathbf{Q}(i, t_i^e)$ for all (i, t') such that $(i, t_i^s) \preceq (i, t') \preceq (i, t_i^e)$. Discrete variables do not change during continuous evolution.
- $\mathbf{X}(i, t_i^s)$, $\mathbf{X}(i, t')$ and $\mathbf{X}(i, t_i^e)$ satisfy G_V and all $F^a(Q^a(i))$ at all times (i, t') such that $(i, t_i^s) \preceq (i, t') \preceq (i, t_i^e)$. Differential equations in G_V and the invariants in each mode dictate the continuous evolution of continuous variables.

Each discrete step $\langle \chi(i, t_i^e) \mapsto \chi(i+1, t_{i+1}^s) \rangle$ is feasible if

- $\left(Q^a(i, t_i^e), Q^a(i+1, t_{i+1}^s)\right) \in Arc^a$ for all automata $a \in Aut$. A transition is only possible between modes for which a transition has been declared.
- $\mathbf{X}(i, t_i^e)$ satisfies $\gamma^a_{\left(Q^a(i,t_i^e), Q^a(i+1,t_{i+1}^s)\right)}$ for all automata $a \in Aut$. A transition is only allowed if the guard for that transition is satisfied.
- $\mathbf{X}(i+1, t_{i+1}^s)$ satisfies G_V and $\rho^a_{\left(Q^a(i,t_i^e), Q^a(i+1,t_{i+1}^s)\right)}$ for all automata $a \in Aut$. Resets, along with any discontinuity equations in G_V, govern how the values of continuous variables jump during an event.

Let $\Xi_{(n, G_t, \mathbf{X}, Aut, G_V)}$ denote the set of all trajectories feasible for the given model.

2.4 Results

We now present a small example to clarify the technical definitions. A larger example in Chapter 9 will be used to demonstrate the benefits of our framework.

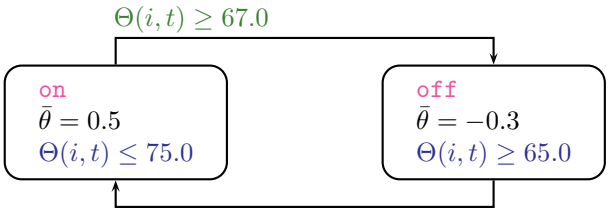
Example 2.1 Consider that we have the following conceptual description of a system:

A thermostat can either be on or off. When it is on, temperature increases at a rate of $0.5\,^\circ\mathrm{F/min}$, and when it is off temperature decreases at rate $0.3\,^\circ\mathrm{F/min}$. The thermostat should not be on if the temperature exceeds $75.0\,^\circ\mathrm{F}$, and when turned on, the temperature

should be allowed to rise to at least $67\,^{\circ}\mathrm{F}$. Finally, it should not be off if the temperature is below $65\,^{\circ}\mathrm{F}$.

Now, our goal is to produce a formal representation of this system, and we accomplish this with the LCCA model

$$n = 10 \tag{2.3a}$$



(2.3b)

$$\frac{\mathrm{d}\Theta\left(i,t\right)}{\mathrm{d}t} = \bar{\theta}\left(Q\left(i\right)\right) \qquad \forall\left(i,t\right) \in \mathbb{T} \tag{2.3c}$$

We have chosen to allow 10 intervals in the timeline. The component automata represents all the constraints on the thermostat's operation. The temperature Θ is governed by the differential equation. In this simple example, it is affected by only a single hybrid process, but in general there might be other terms on the right-hand-side. There are also no resets in this example. Figure 2.3 shows a feasible trajectory for the temperature.

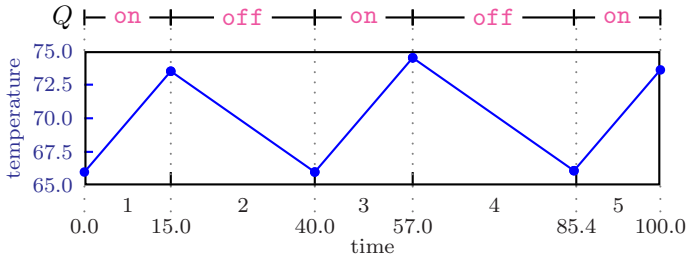


Figure 2.3: Feasible trajectory for thermostat example.

The LCCA framework allows formulating models for discrete-continuous dynamical systems. Next, we consider optimizing such systems.

Chapter 3

Optimizing Hybrid Systems

The previous chapter introduced the linear coupled component automata (LCCA) framework. Dynamic systems represented in the LCCA framework can have multiple feasible trajectories. Optimization would allow the system to be operated along its most profitable path. In this section, we first state what an optimization problem on an LCCA model is, and then define a procedure for transforming this problem to a mixed-integer linear program (MILP).

The transformation will serve two purposes. It is a systematic procedure for generating an MILP, which connects our modeling framework to existing algorithms. But also, it demonstrates how the framework we propose facilitates modeling. As we proceed through the transformation from LCCA to MILP, it will be clear that equivalent declarations become more cumbersome.

3.1 Optimization Problems

The notion of a feasible trajectory ξ for an LCCA model was defined in Section 2.3. Many feasible trajectories are possible for a given model. There is freedom to choose the length of time spent in each discrete mode, various discrete transitions are possible, and values of jump variables are flexible. We used $\Xi_{(n,G_t,\mathbf{X},Aut,G_V)}$ to denote the set of all trajectories feasible for the given model. It is understood that we are working with a specific LCCA model with elements $(n, G_t, \mathbf{X}, Aut, G_V)$, and the subscript will usually be omitted.

An objective Ω is a metric on the space Ξ. An optimization problem seeks the trajectory $\xi \in \Xi$ such that Ω is minimized (or maximized) and is denoted

$$\min_{\xi \in \Xi} \Omega.$$

The objective function can involve any of the real valued variables in the LCCA model, which are the timeline variables $\mathbf{t}$ and the dynamic continuous variables $\mathbf{X}$. For example, if X represents cost, one may wish to minimize its final value. The objective is

$$\Omega = X\left(n, t_n^e\right). \tag{3.1}$$

Or often one is concerned with a time average performance criterion,

$$\begin{aligned}\Omega &= \frac{1}{(t_n^e - t_1^s)} \int_{t_1^s}^{t_n^e} X(i,t)\mathrm{d}t \\ &= \frac{1}{2\,(t_n^e - t_1^s)} \sum_{i\in\mathbb{N}} \left(X\left(t_i^s\right) + X\left(t_i^e\right)\right) \Delta t_i \end{aligned} \tag{3.2}$$

where the integral calculates trapezoidal areas because continuous variables evolve piecewise linearly. Finally, in makespan minimization problems, the value of time is itself the objective,

$$\Omega = t_n^e. \tag{3.3}$$

Such a problem would normally define the notion of a completed job, e.g. producing a certain amount of material. The problem then is to complete the job as fast as possible.

Our general definition of a hybrid timeline allows systems to evolve indefinitely. For optimization purposes, it is necessary to bound the timeline with respect to both the integer and real components of time. The definition of LCCA requires specification of the number of intervals n; so the number of intervals is fixed by definition. Actually, this is less restrictive than it appears. If all transitions into a certain interval are dummy transitions, there has been effectively no change in the physical behavior of the system. Fixing n to a constant only provides an upper bound on the *effective* number of intervals. Second, the final time t_n^e, called the time horizon, must be bounded by or set equal to a time horizon $T^{\max}$ by adding the appropriate constraint in G_t.

The optimization problem we wish to solve is $\min_{\xi\in\Xi} \Omega$. An MILP is of the form $\min_{x\in P} \Omega'$, where x is a vector of real and integer variables and P is a region defined by a system of mixed-integer linear inequalities. Our goal is to convert the former into the latter. The objective function Ω must be converted into a form Ω' allowed in MILP, and an LCCA model must be converted into linear inequalities.

3.2 Constraint Conversions

Constraints in the LCCA framework contain several features not allowed in MP constraints: they are quantified over infinite sets, they employ variable arguments, and finite domain variables are used. In this section, methods for eliminating these features without altering the meaning of the constraints are presented. These will be used in the next section, which discusses a transformation for the full modeling framework.

First, we state a theorem needed a few times in the subsequent discussion.

Theorem 3.1. *Let $g\,(i)$ and $f\,(i)$ be two constraints involving an index i ranging over a set $\mathbb{S}$. Assume $g\,(i)$ holds for exactly one i, i.e. $\veebar_{i\in\mathbb{S}} g\,(i)$ is true. Then, the conjunction of implications*

$$\bigwedge_{i\in\mathbb{S}} \left[g\,(i) \Rightarrow f\,(i)\right] \tag{conj-impl}$$

is equivalent to the disjunction of conjunctions

$$\bigvee_{i \in \mathbb{S}} \left[g\left(i\right) \wedge f\left(i\right) \right]. \tag{disj-conj}$$

The proof is provided in Appendix 3.A on page 40.

3.2.1 Eliminating Infinite Quantifiers

Several constraints are required to hold for all time points $(i,t) \in \mathbb{T}$. All constraints in LCCA are linear or piecewise linear. It is thus possible to consider a finite quantification such that, if a constraint holds over it, it must hold over the infinite set.

First, consider the differential equations, which must be of the form (2.1). The derivative is not continuous over changes in i, but integration over each interval can be considered. In general, we have

$$\mathrm{d}X\left(i,t\right) = \int_{t_i^s}^{t} \left[\sum_{a \in Aut} \bar{x}^a \left(Q^a\left(i\right)\right) + k \right] \mathrm{d}t \qquad \forall \left(i,t\right) \in \mathbb{T}, \tag{3.4}$$

which gives

$$X\left(i,t\right) = X\left(i,t_i^s\right) + \left[\sum_{a \in Aut} \bar{x}^a \left(Q^a\left(i\right)\right) + k \right] \left(t - t_i^s\right) \qquad \forall \left(i,t\right) \in \mathbb{T}. \tag{3.5}$$

The quantification $\forall \left(i,t\right) \in \mathbb{T}$ can be rewritten as $\forall i \in \mathbb{N}, \forall t \in [t_i^s, t_i^e]$. For a fixed i, the above equation is linear in t. Thus, it suffices to consider only $t = t_i^e$, giving

$$X\left(i,t_i^e\right) = X\left(i,t_i^s\right) + \left[\sum_{a \in Aut} \bar{x}^a \left(Q^a\left(i\right)\right) + k \right] \Delta t_i \qquad \forall i \in \mathbb{N}, \tag{3.6}$$

where $\Delta t_i = t_i^e - t_i^s$. In other words, the value of X is determined only at time points t_i^s and t_i^e. Given these, $X\left(i,t\right)$ for any $t \in [t_i^s, t_i^e]$ can be determined because X evolves linearly.

Infinite quantifications also occur in invariants F and certain constraints of G_V. In both cases, the constraints are required to be from $\mathcal{L}\left(\mathbf{X}\left(i,t\right)\right)$. By definition of $\mathcal{L}$, these are linear and the situation is similar to equation 3.5. Each constraint involving $X\left(i,t\right)$ can be written for $X\left(i,t_i^s\right)$ and $X\left(i,t_i^e\right)$ and then enforced over the finite set of intervals. For example, if X represents mass, we might have the constraint

$$X\left(i,t\right) \geq 0 \qquad \forall \left(i,t\right) \in \mathbb{T}, \tag{3.7}$$

which requires mass to be non-negative at all times. This can be replaced with

$$X\left(i,t_i^s\right) \geq 0 \qquad \forall i \in \mathbb{N} \tag{3.8a}$$

$$X\left(i,t_i^e\right) \geq 0 \qquad \forall i \in \mathbb{N}, \tag{3.8b}$$

which requires mass to be non-negative only at the beginning and end of every interval. But

since it varies linearly within an interval, it is guaranteed to be non-negative at all points in between.

3.2.2 Eliminating Variable Arguments

Both dynamic variables and parameters are, in various places, evaluated with variable arguments. For example, $X\left(i, t_i^s\right)$ is evaluated at the two arguments i and t_i^s. Argument i can be interpreted as an index; it is not an unknown. However, the second argument t_i^s is a variable, of unknown value, because event points are not fixed. Continuous dynamic variables arise in this way in several locations: differential equations after integration (3.6), discontinuity equations (2.2), and the initial conditions that are allowed in G_V.

Automata specify a set of flow rates $\bar{\mathbf{x}}$. Each flow rate $\bar{x} \in \bar{\mathbf{x}}$ is a parameter, i.e. $\bar{x}\left(q\right)$ is a known constant given as part of the automaton's declaration. However, in equation (3.6), flow rates are used in the form $\bar{x}\left(Q\left(i\right)\right)$, where the argument $Q\left(i\right)$ is an unknown. Again, there is a term with a variable argument.

Mathematical programs do not allow variable arguments of any form. We present methods for eliminating them in both forms encountered: $X\left(i, t_i^s\right)$ or $X\left(i, t_i^e\right)$, and $\bar{x}\left(Q\left(i\right)\right)$.

$X\left(i, t_i^s\right)$ can be replaced with $X^s\left(i\right)$ wherever it occurs. This is possible with the recognition that $X\left(i, t_i^s\right)$ depends ultimately on just i because its second argument t_i^s is itself fully determined by i. Identically, all occurrences of $X\left(i, t_i^e\right)$ are replaced with $X^e\left(i\right)$. Instead of a single variable X evaluated at two time points (for every i), we have two variables X^s and X^e (for every i).

Replacing $\bar{x}\left(Q\left(i\right)\right)$ is motivated by the same insight. The value of $\bar{x}\left(Q\left(i\right)\right)$ is ultimately dependent on just i. Let us imagine another variable $\bar{w}\left(i\right)$ such that $\bar{w}\left(i\right) = \bar{x}\left(Q\left(i\right)\right)$ for all i. The goal now is to satisfy this equation without resorting to any use of variable arguments. One way to accomplish this is to require

$$\bigwedge_{q \in \mathbb{Q}} \left[Q\left(i\right) = q \Rightarrow \bar{w}\left(i\right) = \bar{x}\left(q\right)\right] \qquad \forall i \in \mathbb{N}. \tag{3.9}$$

This constraint considers every mode. If the automaton is in mode q, then $\bar{w}\left(i\right)$ is set equal to the parameter $\bar{x}\left(q\right)$.

This is a conjunction of implications as in Theorem 3.1 if we recognize q as i, $\mathbb{Q}$ as $\mathbb{S}$, $Q\left(i\right) = q$ as g_i, and $\bar{w}\left(i\right) = \bar{x}\left(q\right)$ as f_i. The theorem can be applied if $Q\left(i\right) = q$ holds for exactly one q (for each i). This of course is true because a variable can only take a single value. So the above can be reformulated into the disjunctive constraint

$$\bigvee_{q \in \mathbb{Q}} \begin{bmatrix} Q\left(i\right) = q \\ \bar{w}\left(i\right) = \bar{x}\left(q\right) \end{bmatrix} \qquad \forall i \in \mathbb{N}. \tag{3.10}$$

In general, a new variable $\bar{w}^a$ will be needed for each $\bar{x}^a \in \bar{\mathbf{x}}^a$ of every automaton.

Let us implement these replacements into equation (3.6), which involves both trouble-

some forms. It becomes the two constraints

$$X^e(i) = X^s(i) + \sum_{a \in Aut} (\bar{w}^a(i) + k)\,\Delta t_i \qquad \forall i \in \mathbb{N} \tag{3.11}$$

$$\bigvee_{q \in \mathbb{Q}^a} \left[\begin{array}{c} Q^a(i) = q \\ \bar{w}^a(i) = \bar{x}^a(q) \end{array} \right] \qquad \forall a \in Aut, \forall i \in \mathbb{N}. \tag{3.12}$$

Unfortunately, the first equation contains a bilinearity $\bar{w}^a(i)\,\Delta t_i$.

A slight modification to the procedure allows producing a linear equation. Instead of letting $\bar{w}(i) = \bar{x}(Q(i))$, require $\bar{w}(i) = \bar{x}(Q(i))\,\Delta t_i$. Now, we can write

$$X^e(i) = X^s(i) + \sum_{a \in Aut} (\bar{w}^a(i) + k\Delta t_i) \qquad \forall i \in \mathbb{N} \tag{3.13}$$

$$\bigvee_{q \in \mathbb{Q}^a} \left[\begin{array}{c} Q^a(i) = q \\ \bar{w}^a(i) = \bar{x}^a(q)\,\Delta t_i \end{array} \right] \qquad \forall a \in Aut, \forall i \in \mathbb{N}. \tag{3.14}$$

Instead of having to multiply Δt_i with $\bar{w}^a(i)$, it is multiplied by $\bar{x}^a(q)$ within the disjunction. The latter is a parameter, and so the equation is still linear.

3.2.3 Converting Finite Domains to Booleans

By definition of $\mathcal{L}$, finite domain constraints are of the form $Q(i) = q$ or $Q(i) = Q(i+1)$, and these can be connected by the logical operators $\neg$, $\vee$, and $\wedge$. Both equations can be converted into Boolean propositions. For each dynamic finite domain variable Q of type $\mathbb{N} \to \mathbb{Q}$, introduce a Boolean variable Y of type $\mathbb{Q} \times \mathbb{N} \to \{\texttt{true}, \texttt{false}\}$. This allows associating a Boolean with each possible value of the finite domain variable.

The equation $Q(i) = q$ is simply replaced by the Boolean $Y(q, i)$. This substitution alone is not sufficient however. It would be possible for $Y(q, i)$ and $Y(q', i)$ to both be true for distinct q and q'. Translating this solution back into the original model would imply that $Q(i)$ takes two values. Clearly, that cannot be allowed. For every Boolean Y introduced, it is necessary to include the constraint

$$\bigvee_{q \in \mathbb{Q}} Y(q, i) \qquad \forall i \in \mathbb{N} \tag{3.15}$$

which guarantees that $Y(q, i)$ will be true for exactly one q (for each i).

The equation $Q(i) = Q(i+1)$ effectively says that there exists some q such that the automaton is in mode q for both intervals i and $i+1$. Stated as a formula, we have

$$\exists q \in \mathbb{Q} \text{ s.t. } [Q(i) = q \wedge Q(i+1) = q], \tag{3.16}$$

and now the finite domain equations are in the simpler form. The existential quantifier, when quantified over a finite set, is merely an alternative notation for indexed disjunction.

Simply changing this notation and replacing the equations with Booleans gives

$$\bigvee_{q\in\mathbb{Q}} \left[Y\left(q,i\right) \wedge Y\left(q,i+1\right)\right]. \tag{3.17}$$

3.3 Symmetry Breaking

In the course of generating an MILP, we also add some constraints for efficiency purposes; these do not affect the model. The source of the inefficiency is dummy transitions. They are a mathematical artifact allowing an automaton to transition but in a way that has no physical consequence. Unfortunately, these introduce a redundancy in the set of feasible trajectories because, at some event point, all automata might make a dummy transition, meaning the system has not actually evolved. Also, this could occur at a continuum of time values. An infinite number of mathematically distinct trajectories represent an identical physical solution. Avraam et al. (1998, p. S225) recognized this problem in a related system and proposed a solution which we accommodate to our framework.

A *dummy event point* is one at which all automata make dummy transitions. We require all dummy event points to occur at the end of a trajectory, i.e. if $i \in \mathbb{N}\backslash\{n\}$ is a dummy event point, then j is a dummy event point for all $j > i$. Also, the interval length after each dummy event point should be zero, i.e. if $i \in \mathbb{N}\backslash\{n\}$ is a dummy event point, then $\Delta t_{i+1} = 0.0$. Occurrence of a dummy event point means the effective number of intervals is less than n. The trajectory with dummy event points squeezed at the end of the timeline has many others equivalent to it. Adding the stated constraints makes all but this one infeasible.

From the previous section, we let the Boolean $Y^a\left(q,i\right)$ substitute for the constraint $Q^a\left(i\right) = q$. For convenience, we let $YY^a\left(i\right)$ mean automaton a makes a dummy transition at the i^{th} event point and $YYY\left(i\right)$ mean the i^{th} event point is dummy. These are defined in terms of Y's with the constraints

$$YY^a\left(i\right) \Leftrightarrow \bigvee_{q\in\mathbb{Q}^a} \left[Y^a\left(q,i\right) \wedge Y^a\left(q,i+1\right)\right] \tag{3.18}$$

$$YYY\left(i\right) \Leftrightarrow \bigwedge_{a\in Aut} YY^a\left(i\right). \tag{3.19}$$

Now, the two symmetry breaking constraints are

$$YYY\left(i\right) \Rightarrow YYY\left(i+1\right) \qquad \forall i \in \mathbb{N}\backslash\{n-1,n\} \tag{3.20a}$$

$$YYY\left(i\right) \Rightarrow \left(\Delta t_{i+1} = 0.0\right) \qquad \forall i \in \mathbb{N}\backslash\{n\}. \tag{3.20b}$$

The antecedent of both determines if i is a dummy event point by checking if all automata's discrete modes have remained unchanged. If so, the first requires the next event point to also be dummy and the second sets the next interval length to zero. If the antecedent is satisfied for i, the conclusion of the first constraint is such that its antecedent will be true for $i+1$. This causes the constraint to be iteratively enforced for all $j > i$. The antecedent

might not be satisfied for any i—there might not be any dummy event points—in which case these constraints have no effect.

3.4 Model Transformation

Converting the optimization problem $\min_{\xi \in \Xi} \Omega$ into the MILP $\min_{x \in P} \Omega'$ requires converting the objective and the model. Some objectives, e.g. minimize makespan, are already in an MILP form. Others, e.g. time averaged cost, involve terms with variable arguments. These are easily transformed by substituting variables X^s and X^e as discussed in Section 3.2.2. It remains to convert an LCCA model into MILP constraints.

A few steps are required to transform each of the elements $(n, G_t, \mathbf{X}, Aut, G_V)$ of an LCCA model.

- The timeline is represented with MILP constraints.
- Component automaton are reformulated into two disjunctive constraints: one over the discrete modes and the other over the transitions.
- These constraints along with G_V are converted using the methods of the previous section to eliminate infinite quantifiers, remove variable arguments, and transform finite domain logic into Boolean propositions. The result is a generalized disjunctive program (GDP).
- Finally, the GDP model is converted into an MILP using a technique provided by Raman and Grossmann (1994).

The first element of an LCCA model n is used simply to define the index set $\mathbb{N} = \{1, \ldots, n\}$ in the MILP model. According to the definition of LCCA, n is used to construct a hybrid timeline. The timeline is dictated by variables $\mathbf{t}$, and these are included in the MILP model unaltered. The constraints implicit in the definition of a hybrid timeline

$$t_i^s \leq t_i^e \qquad \forall i \in \mathbb{N} \tag{3.21a}$$

$$t_i^e = t_{i+1}^s \qquad \forall i \in \mathbb{N} \backslash \{n\} \tag{3.21b}$$

are included explicitly in the MILP model. Constraint G_t is already in an MILP form and is included unaltered.

Each variable $X \in \mathbf{X}$ is of type $\mathbb{T} \to \mathbb{R}$. Functions whose domains are infinite spaces are not allowed in MILP. Following the methods of Section 3.2.2, each X will be replaced with two variables X^s and X^e of type $\mathbb{N} \to \mathbb{R}$. Such functions can be interpreted simply as indexed variables and are allowed in MP models. Let $\mathbf{X}^s$ and $\mathbf{X}^e$ denote these new sets of variables.

The component automata Aut are the most involved constructs of an LCCA model. They can be restated as disjunctive constraints on real and finite domain variables. Recall each automaton in Aut is of the form $(\mathbb{Q}, \bar{\mathbf{x}}, \hat{\mathbf{x}}, F, Arc)$. (In this section, we speak generically of any automaton, and so the superscript a is omitted.) The finite domain space is left

unaltered but will now be interpreted as an index set. The parameters $\bar{\mathbf{x}}$ and variables $\hat{\mathbf{x}}$ also remain unchanged; they are of a type allowed in MILP. It is F and Arc that must be transformed.

The invariant constraint F for each automaton is expressed as

$$F\left(Q\left(i\right)\right) \qquad \forall\left(i,t\right)\in\mathbb{T}, \tag{3.22}$$

which says there is some constraint associated with each mode, and we apply the constraint for the mode the system is currently in. Instead, consider each possible value of $Q\left(i\right)$ separately. Then, the above can be equivalently stated as

$$Q\left(i\right)=q\Rightarrow F\left(q\right) \qquad \forall q\in\mathbb{Q},\forall\left(i,t\right)\in\mathbb{T}. \tag{3.23}$$

If the automaton is in mode q, then the invariant for that mode must be applied. The variable argument has been removed at the expense of introducing a quantifier. A universal quantifier over a finite set can be viewed as a notational variation for indexed conjunction. The above can be rewritten as

$$\bigwedge_{q\in\mathbb{Q}}\left[Q\left(i\right)=q\Rightarrow F\left(q\right)\right] \qquad \forall\left(i,t\right)\in\mathbb{T}. \tag{3.24}$$

This is a conjunction of implications as in Theorem 3.1 if we recognize q as i, $\mathbb{Q}$ as $\mathbb{S}$, $Q\left(i\right)=q$ as g_i, and $F\left(q\right)$ as f_i. The theorem allows reformulating the constraint into a *disjunction over modes*

$$\bigvee_{q\in\mathbb{Q}}\begin{bmatrix} Q\left(i\right)=q \\ F\left(q\right)\end{bmatrix} \qquad \forall\left(i,t\right)\in\mathbb{T}. \tag{3.25}$$

The constraint still involves an infinite quantifier and the disjunct involves a finite domain variable. Application of the constraint conversions discussed in Section 3.2 will produce a disjunction in a GDP form.

At event i, a transition can occur from mode q to q' if $(q,q')\in Arc$. In addition, it is required that both the guard $\gamma_{(q,q')}$ and reset $\rho_{(q,q')}$ are enforced. Written as a formula, we can say

$$\bigwedge_{(q,q')\in Arc}\left[\left(Q\left(i\right)=q\wedge Q\left(i+1\right)=q'\right)\Rightarrow\gamma_{(q,q')}\wedge\rho_{(q,q')}\right] \qquad \forall i\in\mathbb{N}\backslash\left\{n\right\}. \tag{3.26}$$

Every transition is considered. If the transition taken from interval i to $i+1$ is from mode q to q', then the guard and reset along that transition are enforced.

Again, we have a conjunction of implications as in Theorem 3.1 if we recognize (q,q') as i, Arc as $\mathbb{S}$, $Q\left(i\right)=q\wedge Q\left(i+1\right)=q'$ as g_i, and $\gamma_{(q,q')}\wedge\rho_{(q,q')}$ as f_i. The theorem is applicable if $Q\left(i\right)=q\wedge Q\left(i+1\right)=q'$ is valid only for a unique pair (q,q'). It must hold for at least one pair because a transition must be made at an event. It does not hold for more than one because component automaton, by definition, allow only a single transition between any two modes (with possibly both modes the same). Thus, we can transform the

above constraint into a *disjunction over transitions*

$$\bigvee_{(q,q')\in Arc} \begin{bmatrix} Q(i) = q \wedge Q(i+1) = q' \\ \gamma_{(q,q')} \wedge \rho_{(q,q')} \end{bmatrix} \qquad \forall i \in \mathbb{N}\backslash\{n\}. \tag{3.27}$$

The number of disjuncts can be reduced by recognizing that $\gamma_{(q,q)}$ and $\rho_{(q,q)}$ are identical for all dummy transitions (q,q). First, let the finite domain constraint be replaced with the Boolean proposition $Y(q,i) \wedge Y(q',i+1)$, and recall equation (3.18) defined $YY(i)$—the superscript a omitted for now—to mean an automaton makes a dummy transition at event i. Now, the disjuncts can be partitioned to give

$$\left(\bigvee_{\substack{(q,q')\in Arc \\ q\neq q'}} \begin{bmatrix} Y(q,i) \wedge Y(q',i+1) \\ \gamma_{(q,q')} \wedge \rho_{(q,q')} \end{bmatrix} \right) \vee \begin{bmatrix} YY(i) \\ \gamma_{(q,q)} \wedge \rho_{(q,q)} \end{bmatrix} \qquad \forall i \in \mathbb{N}\backslash\{n\}. \tag{3.28}$$

There is now one disjunct instead of $|\mathbb{Q}|$ for the dummy transitions.

The disjunction over modes (3.25) and the disjunction over transitions (3.28) can be written for all component automaton $a \in Aut$, replacing F^a and Arc^a for each.

These disjunctions along with G_V still involve infinite quantifiers, variable arguments, and finite domain constraints. Let F', γ', ρ', and G'_V refer to the respective constraints after applying the conversions of Section 3.2. This introduces a new set of variables $\bar{\mathbf{w}}$, when eliminating variable arguments in terms of the form $\bar{x}(Q(i))$. Let $\mathbf{Y}$ be the set of Boolean variables introduced to replace the finite domain variables. In summary, the resulting GDP model is

$$\min_{\mathbf{t},\mathbf{X}^s,\mathbf{X}^e,\hat{\mathbf{x}},\bar{\mathbf{w}},\mathbf{Y}} \Omega' \tag{GDP(LCCA))}$$

$$\text{s.t.} \qquad t_i^s \leq t_i^e \qquad \forall i \in \mathbb{N}$$

$$t_i^e = t_{i+1}^s \qquad \forall i \in \mathbb{N}\backslash\{n\}$$

$$G_t$$

$$\bigvee_{q\in\mathbb{Q}^a} \begin{bmatrix} Y^a(q,i) \\ F'(q) \end{bmatrix} \qquad \forall i \in \mathbb{N}, \forall a \in Aut$$

$$\left(\bigvee_{\substack{(q,q')\in Arc \\ q\neq q'}} \begin{bmatrix} Y(q,i) \wedge Y(q',i+1) \\ \gamma'_{(q,q')} \wedge \rho'_{(q,q')} \end{bmatrix} \right) \vee \begin{bmatrix} YY(i) \\ \gamma'_{(q,q)} \wedge \rho'_{(q,q)} \end{bmatrix} \qquad \begin{matrix} \forall i \in \mathbb{N}\backslash\{n\}, \\ \forall a \in Aut \end{matrix}$$

$$G'_V$$

$$YYY(i) \Rightarrow YYY(i+1) \qquad \forall i \in \mathbb{N}\backslash\{n-1,n\}$$

$$YYY(i) \Rightarrow (\Delta t_{i+1} = 0.0) \qquad \forall i \in \mathbb{N}\backslash\{n\}.$$

Each of the elements $(n, G_t, \mathbf{X}, Aut, G_V)$ of an LCCA model are represented in this GDP model; so we call it GDP(LCCA). Element n is used to define the index set $\mathbb{N} = \{1, \ldots, n\}$, the first two constraints define a timeline, G_t is included unaltered, the two main disjunctions represent the component automata, and G'_V is the result of converting the final component G_V. The symmetry breaking constraints are added for efficiency reasons. Items $\mathbb{Q}^a$, Aut, and Arc are still present but they are now used only as index sets.

Model GDP(LCCA) is nearly in the form defined by Raman and Grossmann (1994), and so their transformation can be mostly applied to produce an MILP. Two main steps are required: the Boolean propositions within the disjunctions, G'_V, and the symmetry breaking constraints are converted into integer constraints, and the disjunctive constraints are transformed using the convex hull method.

An implicit requirement of Raman and Grossmann's (1994) method is that exactly one Boolean variable amongst all in the disjuncts of a disjunction must be true. For example, their method can be applied to convert the disjunction over modes only if $\veebar_{q \in \mathbb{Q}^a} Y^a(q, i)$ holds for all i and a. This is satisfied because the disjunction over modes was obtained by application of Theorem 3.1 for which the exclusivity constraint is a precondition. Similarly, $\veebar_{(q,q') \in Arc^a} Z^a(q, q', i)$ is guaranteed; so their method can be applied to the disjunction over transitions also. Finally, G'_V will contain disjunctions of the form (3.14), and their requirement is satisfied here also.

Model GDP(LCCA) differs from Raman and Grossmann's form in two minor ways. They allow only a single Boolean variable in disjunctive constraints, but the disjunction over transitions includes the Boolean expression $Y^a(q, i) \wedge Y^a(q', i+1)$. This is easily rectified by adding the Boolean constraint

$$Z^a(q, q', i) \Leftrightarrow (Y^a(q, i) \wedge Y^a(q', i+1)) \qquad \forall i \in \mathbb{N} \setminus \{n\}, \forall a \in Aut, \forall q, q' \in \mathbb{Q}^a \tag{3.29}$$

and then replacing the Boolean expression in the disjunct with $Z^a(q, q', i)$. (We can also use Z^a to simplify equation (3.18) to $YY^a(i) \Leftrightarrow \vee_{q \in \mathbb{Q}^a} Z^a(q, q, i)$.)

The second difference arises in the symmetry breaking constraint

$$YYY(i) \Rightarrow (\Delta t_{i+1} = 0.0).$$

This is equivalent to the disjunction $[\neg YYY(i)] \vee [\Delta t_{i+1} = 0]$, which is not in the form required for Raman and Grossmann's method. However, this constraint is easily seen to be equivalent to the mixed-integer inequalities

$$0.0 \leq \Delta t_{i+1} \leq T^{\max}(1 - yyy(i)). \tag{3.30}$$

With this final conversion, model GDP(LCCA) is converted into an MILP.

3.5 Conclusions

Example 3.1 An LCCA model of a thermostat was presented in Example 2.1 on page 27. Using the techniques provided in this chapter, we generate the following equivalent GDP model.

$$t_i^s \leq t_i^e \qquad \forall i \in \mathbb{N} \tag{3.31a}$$

$$t_i^e = t_{i+1}^s \qquad \forall i \in \mathbb{N} \backslash \{n\} \tag{3.31b}$$

$$\Delta t_i = t_i^e - t_i^s \qquad \forall i \in \mathbb{N} \tag{3.31c}$$

$$\begin{bmatrix} Y(\texttt{on}, i) \\ \Theta^s(i) \leq 75.0 \\ \Theta^e(i) \leq 75.0 \end{bmatrix} \vee \begin{bmatrix} Y(\texttt{off}, i) \\ \Theta^s(i) \geq 65.0 \\ \Theta^e(i) \geq 65.0 \end{bmatrix} \qquad \forall i \in \mathbb{N} \tag{3.32a}$$

$$\begin{bmatrix} Z(\texttt{on}, \texttt{off}, i) \\ \Theta^e(i) \geq 67.0 \end{bmatrix} \vee \begin{bmatrix} Z(\texttt{off}, \texttt{on}, i) \end{bmatrix} \vee \begin{bmatrix} YY(i) \end{bmatrix} \qquad \forall i \in \mathbb{N} \backslash \{n\} \tag{3.32b}$$

$$\Theta^e(i) = \Theta^s(i) + \bar{w}(i) \qquad \forall i \in \mathbb{N} \tag{3.33a}$$

$$\Theta^s(i+1) = \Theta^e(i) \qquad \forall i \in \mathbb{N} \backslash \{n\} \tag{3.33b}$$

$$\bigvee_{q \in \mathbb{Q}} \begin{bmatrix} Y(q, i) \\ \bar{w}(i) = \bar{\theta}(q) \Delta t_i \end{bmatrix} \qquad \forall i \in \mathbb{N} \tag{3.33c}$$

$$\bigvee_{q \in \mathbb{Q}} Y(q, i) \qquad \forall i \in \mathbb{N} \tag{3.34}$$

$$YYY(i) \Rightarrow YYY(i+1) \qquad \forall i \in \mathbb{N} \backslash \{n-1, n\} \tag{3.35a}$$

$$YYY(i) \Rightarrow (\Delta t_{i+1} = 0.0) \qquad \forall i \in \mathbb{N} \backslash \{n\} \tag{3.35b}$$

$$YY(i) \Leftrightarrow \bigvee_{q \in \mathbb{Q}} Z(q, q, i) \qquad \forall i \in \mathbb{N} \backslash \{n\} \tag{3.36a}$$

$$YYY(i) \Leftrightarrow YY(i) \qquad \forall i \in \mathbb{N} \backslash \{n\} \tag{3.36b}$$

$$Z(q, q', i) \Leftrightarrow (Y(q, i) \wedge Y(q', i+1)) \qquad \forall i \in \mathbb{N} \backslash \{n\}, \forall q, q' \in \mathbb{Q} \tag{3.36c}$$

Since there is only one component automaton in this small example, YYY is identical to YY.

We can see that the LCCA model is easier to formulate, and our transformation proce-

dure allows mechanically generating the more complex GDP constraints.

Through the previous chapter and this one, we have presented the overall goals of this dissertation. The novel LCCA framework facilitates modeling be providing forms of expression more natural for hybrid systems. In this chapter, we demonstrated how models in the LCCA framework can be systematically transformed into MILP models. This provides us the ability to model in a more elegant system, while still benefiting from existing algorithms.

Further improvements to this work are needed: the models should be made comprehensible to a computer and the transformations should be automated. Doing so requires more rigorous definitions than provided so far.

Consider our definition of $\gamma_{(q,q')}$, stated to be a constraint associated with each $(q, q') \in Arc$. It is quite clear to us what is meant by this definition. However, expressing $\gamma_{(q,q')}$ in a computer language requires the constructs γ and (q, q') to be defined more precisely. The latter is an element of an index set that contains pairs. The modeler should be able to define any such set; so we must define what the set of all index sets is. This will be done in Chapter 6, where we provide a theory of indexing. Next, the modeler should be able to define a construct called γ, which when applied to the pair (q, q') returns a constraint. In other words γ is a function that maps an index set to the space of constraints. The modeler should be able to declare any γ of their choice; so we must define the set of all such functions. This will be done in Chapter 7.

Only with these definitions in place will we have a definition of the set of all programs we are considering. And only then can we define a mapping on this set, which is what a model transformation is. The remainder of this dissertation regards employing type theory to provide such definitions for the mathematical programming parts of our overall goal.

Appendix 3.A Proof of Theorem 3.1

Let us first define some terminology. Given an implication $a \Rightarrow b$, a is called the *antecedent* and b the *consequent*. In a conjunctive constraint $a \wedge b$, each of a and b are called *conjuncts*. For indexed conjunction $\wedge_{i \in \mathbb{S}} g(i)$, each $g(i)$ is called a conjunct. Similarly the terms comprising a disjunctive constraint are called *disjuncts*.

The equivalence is proven by demonstrating implication in both directions.

First, we prove the forward implication: conj-impl $\Rightarrow$ disj-conj, i.e. assume conj-impl is true and show disj-conj must be true. An assumption of the theorem is that $g(i)$ holds for exactly one i. Without loss of generality, assume this value is i'. Rewrite conj-impl as

$$\underbrace{[g(i') \Rightarrow f(i')]}_{\text{lconj}} \wedge \underbrace{\bigwedge_{i \in \mathbb{S} \setminus \{i'\}} [g(i) \Rightarrow f(i)]}_{\text{rconj}}.$$

For conj-impl to be true, as is being assumed, both lconj and rconj must be true. First, we state the conditions under which these both hold:

- lconj is true if $f(i')$ is satisfied. This is because, by assumption, the antecedent $g(i')$ of lconj is true, and for the whole implication to be true, the consequent must be true.

- rconj is always true. This follows because we are assuming $g(i')$, and thus $g(i)$ is false for all $i \neq i'$. This means the antecedent $g(i)$ of every conjunct of rconj is false, making every conjunct true irrespective of the consequent $f(i)$ because falsehood can imply anything.

The net result is that $f(i')$ must be true. With this observation, it remains to show that disj-conj must be true. Divide up disj-conj similarly as

$$\underbrace{[g(i') \wedge f(i')]}_{\text{ldisj}} \vee \underbrace{\bigvee_{i \in \mathbb{S} \setminus \{i'\}} [g(i) \wedge f(i)]}_{\text{rdisj}}$$

For disj-conj to be true, either ldisj must be true or rdisj must be true. We show that ldisj is true. By assumption $g(i')$ is true, and we just argued that $f(i')$ must be true. So $[g(i') \wedge f(i')]$ is true.

Now, we prove the implication in the opposite direction: disj-conj $\Rightarrow$ conj-impl, i.e. show that conj-impl must be true under the assumption that disj-conj is true. disj-conj can be true if any one of its disjuncts is true. Let i' be the index of one of the true disjuncts, i.e. $[g(i') \wedge f(i')]$ is true. In fact, this is the only disjunct that can be true because all others require $g(i)$ to hold for some $i \neq i'$, which violates the assumption of the theorem. So we know that $f(i')$ must hold. Now, divide up conj-impl as above into lconj and rconj. It remains to show that both lconj and rconj are true. It was assumed that $g(i')$ holds and we just argued that $f(i')$ must hold; thus lconj follows immediately. rconj is vacuously true because $g(i)$ is false for all $i \neq i'$, making each conjunct of rconj true irrespective of $f(i)$.

Chapter 4

Logical Formulation of Mathematical Programs

Our purpose in the remainder of the dissertation is similar to that in the previous two chapters in the sense that we wish to define modeling frameworks and transformations between them. But it is also markedly different because we now make the stringent demand that these definitions be comprehensible to a computer; in other words, that they be completely formal.

A computer should be able to read in an arbitrary input file and determine definitively whether the contents of the file do or do not represent a model in the purported framework. It should accept models written in the most natural manner possible, not only those in a canonical form. Next, the computer should be able to apply the desired transformation and output the result. These demands are addressed for the framework of mathematical programming (MP). Meeting them is not merely a software development matter. Rather, we provide a novel logic-based definition of MP.

No knowledge of type theory is assumed in this chapter. We define MPs without indexing to keep the initial formulation as simple as possible. Indexing significantly complicates the theory and will be introduced in later chapters. Familiarity with induction and set relations will be helpful, although we provide a rudimentary introduction in the next section. These concepts are essential to the remainder of the dissertation.

The main content begins with a definition of the syntax of unindexed MPs, followed by a declaration of the language's type system, and finally a semantic interpretation of the syntax. Small examples at the end demonstrate the results.

4.1 Mathematical Preliminaries

4.1.1 Induction

Virtually every definition in the main paper employs induction. Frequently, induction is explained over the natural numbers, which is a special case of the version we use. Induction over numbers follows the steps:

1. Prove that a property P is true for $k = 0$.
2. Assume P is true for $k = n$, and under this assumption, prove that it holds for $k = n + 1$.
3. Invoke the principle of induction to conclude that P is true (for all k).

This version depends on the existence of the natural numbers, requires the property of interest to be parametric on numbers, allows only a single base case ($k = 0$), and allows the induction step to proceed in only one direction (from $k = n$ to $k = n + 1$). Type theory requires a generalized principle of induction, which is more flexible and does not depend on the prior existence of any mathematical construct.

The general idea is explained with an example; we provide an inductive definition of the set of natural numbers. The definition is

$$k ::= 0 \mid \mathit{succ}\,(k) \tag{4.1}$$

This notation is interpreted as follows. We are defining a set. An element of this set is generically referred to as k. The definition then says that an element k can take one of two syntactic forms; it can either be "0", or given some other k it can be the string "$\mathit{succ}\,(k)$". This is just a way notating natural numbers; $\mathit{succ}\,(k)$ does not represent an operation on k.

The first form is the base case; it is how we start producing elements of this set. The second form is what makes this an inductively defined set. It says that, given any element k in the set, $\mathit{succ}\,(k)$ is also an element of the set. For example, "0" is in the set and thus so is "$\mathit{succ}\,(0)$". But now we have that "$\mathit{succ}\,(0)$" is in the set, and thus so is "$\mathit{succ}\,(\mathit{succ}\,(0))$". An infinite set has been defined, but all its elements are of one of two syntactic forms.

Note that we never named the set being defined. We usually will not, but let us momentarily use $\mathbb{Z}_+$ to refer to the set defined above. It is understood that any use of "k" means for all $k \in \mathbb{Z}_+$. However, we will subsequently not write the "for all $k \in \mathbb{Z}_+$" part.

The above definition has defined a syntax for natural numbers. Syntax is not a mere notational matter. By defining the syntax of natural numbers, we have invented natural numbers. The syntax being defined is called the abstract syntax, which is meant to capture the core mathematical principles. It is usually necessary to introduce a concrete syntax to simplify the notation and to accommodate practical considerations such as the fact that most parsers only handle ASCII text. For example, we let "1" stand for "$\mathit{succ}\,(0)$" and "2" for "$\mathit{succ}\,(\mathit{succ}\,(0))$". Our focus is always on the abstract syntax. Providing concrete syntax is a relatively minor matter and is discussed in Appendix C for our language.

The above definition of natural numbers is a special case of a more general definition style, called an inductive definition. An inductive definition consists of a collection of *inference rules*, which together define a set. For example, we define x nat to mean x is an object in the set of natural numbers (customarily written $x \in \mathbb{Z}_+$). Statements such as x nat are called *judgements* because we are judging an object to have a certain property—in this case, that it lies in the set of natural numbers.

An inference rule takes the form

$$\frac{J_1 \quad \cdots \quad J_m}{J} \tag{4.2}$$

where each J_i is a judgement. The meaning of this notation is that J is a conclusion that holds if all of the preconditions J_1, ..., J_m hold. A collection of such rules defines a single set. The definition of natural numbers can now be provided in the alternate form

$$\frac{}{0 \text{ nat}} \tag{4.3a}$$

$$\frac{x \text{ nat}}{succ\,(x) \text{ nat}} \tag{4.3b}$$

There are two rules. Together, they define the set of natural numbers. The first says that the object "0" is a natural number, without condition. The second says that if x is a natural number then so is "$succ\,(x)$".

We now see that the definition (4.1) is just a compact notation for this more general definition style. We used the symbol k to refer to objects that satisfy the judgement k nat. This is useful because we may need objects with certain properties very often. Instead of constantly saying x such that x nat, we can just use k. The compact notation was possible only because of the simplicity of the preconditions. Usually, the general definition style will be required.

Given a set, it is possible to define relations on the set. A binary relation, for example, relates two elements to each other. Let R be a binary relation on the natural numbers. The notation $k_1\,R\,k_2$ means the pair (k_1, k_2) is an element of the relation R. The definition of R can be provided inductively on the construction of k_1 and k_2. "On the construction of" means the definition provides rules for each of the syntactic forms of k_1 and k_2. We say "construction" because each of the syntactic forms 0 and $succ\,(k)$ is a method for constructing a natural number.

Let us consider an example. Define R by the rules

$$\frac{}{0\,R\,0} \tag{4.4a}$$

$$\frac{k_1\,R\,k_2}{succ\,(k_1)\;R\,succ\,(k_2)} \tag{4.4b}$$

The first rule says that $(0, 0)$ is in R. The second says that if (k_1, k_2) is in R, then so is $(succ\,(k_1)\,, succ\,(k_2))$. Recall our convention; the second rule is implicitly written for all $k_1 \in \mathbb{Z}_+$, $k_2 \in \mathbb{Z}_+$. The judgement, or relation, R that has been defined is of course known as equality. Instead of R, let us name this judgement $=$. Then, instead of writing $k_1\,R\,k_2$, we write $k_1 = k_2$.

Thus, $=$ is a set. It is a set of pairs of natural numbers. Now, various questions about this set may be asked. For example, is $(0, succ\,(0))$ in the set? In other words, is $0 = succ\,(0)$ true? It is not, and we can prove this by looking at the rules defining $=$. The conclusion of neither rule matches the syntactic forms in question. Neither has a 0 as its left argument and $succ\,(0)$ as its right argument. Thus, $(0, succ\,(0))$ is not an element of $=$. We were able

to determine that none of the infinity of elements in $=$ matched $(0, succ(0))$ by looking at the syntactic forms of the elements in $=$, of which there are just two combinations.

4.1.2 Types

All elements k in the set of natural numbers defined above are well-formed. However, in more complex sets, it is usually not possible or convenient to provide a definition that immediately contains only well-formed elements. In other words, there exists a need for syntax that sometimes is well-formed but sometimes is not. A particularly important judgement, known as a typing judgement, becomes important in these cases.

An example is a set of expressions, which might be defined by

$$\begin{aligned} e ::= {} & k \mid \texttt{true} \mid \texttt{false} \\ & \mid e_1 + e_2 \mid e_1 - e_2 \\ & \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \end{aligned} \tag{4.5}$$

There are seven syntactic forms. We would like $2 + 5$ to be an expression, so we need the syntax for addition expressions. However, by providing this, $2 + \texttt{false}$ is also an expression, which we do not want. The resolution is to introduce types, which lie at the core of the eponymous theory.

Let us also define the types

$$\tau ::= \texttt{int} \mid \texttt{bool} \tag{4.6}$$

which happens not be an inductive set. There are exactly two types. Now, a typing judgement $e : \tau$ can be provided to associate expressions with types. The relation ":" defines which expressions are well-formed. An expression e is well-formed only if there exists a τ such that $e : \tau$ is satisfied. The definition of ":" should be such that neither $(2 + \texttt{false}, \texttt{int})$ nor $(2 + \texttt{false}, \texttt{bool})$ are an element of it.

We define $e : \tau$ with the rules

$$\frac{}{k : \texttt{int}} \tag{4.7a}$$

$$\frac{}{\texttt{true} : \texttt{bool}} \tag{4.7b}$$

$$\frac{}{\texttt{false} : \texttt{bool}} \tag{4.7c}$$

$$\frac{e_1 : \texttt{int} \quad e_2 : \texttt{int}}{e_1 + e_2 : \texttt{int}} \tag{4.7d}$$

$$\frac{e_1 : \texttt{int} \quad e_2 : \texttt{int}}{e_1 - e_2 : \texttt{int}} \tag{4.7e}$$

$$\frac{e_1 : \texttt{bool} \quad e_2 : \texttt{bool}}{e_1 \wedge e_2 : \texttt{bool}} \tag{4.7f}$$

$$\frac{e_1 : \texttt{bool} \quad e_2 : \texttt{bool}}{e_1 \vee e_2 : \texttt{bool}} \tag{4.7g}$$

The definition is said to be inductive on the form of e because there is one rule for each of the syntactic forms of e.

The first rule states that any number k is of type `int`, without condition. The second and third rules are similar for the Boolean constants. Next, the expression $e_1 + e_2$ is declared to be of type `int` if both $e_1 : \texttt{int}$ and $e_2 : \texttt{int}$ are true, and the remaining rules are similar.

We wish to check whether the expression $2 + (3 - 1)$ is of type `int`. This expression is in the form $e_1 + e_2$, where e_1 is 2 and e_2 is $3 - 1$. We look for a rule whose conclusion is in this form and find that the fourth rule matches. It says that $2 + (3 - 1) : \texttt{int}$ is true under the conditions $2 : \texttt{int}$ and $(3 - 1) : \texttt{int}$. We now must prove each of these. The judgement $2 : \texttt{int}$ matches the first rule; it is declared true without any conditions, so the recursion stops. We must also check $(3 - 1) : \texttt{int}$, and by the same procedure eventually find that it is true. So finally we conclude that the judgement $2 + (3 - 1) : \texttt{int}$ is satisfied.

Now, we try to check if $2 + \texttt{false} : \texttt{int}$ is satisfied. The fourth rule says it is under the conditions $2 : \texttt{int}$ and $\texttt{false} : \texttt{int}$. The first of these is satisfied, but the second is not. There is no rule whose conclusion has the expression `false` and type `int`. Thus, $2 + \texttt{false}$ is not of type `int`. Similarly, we could find that it is also not of type `bool`.

We have been using a top-down proof strategy for type checking expressions. (It is called top-down even though the notation makes it appear as if we are moving from bottom to top). In general, given an expression e and type τ, we look at the conclusions of the rules defining the judgement $e : \tau$ to find the one that matches the syntactic forms of the given e and τ. Checking that the conclusion is satisfied requires moving to its preconditions and checking them. The procedure continues until the recursion reaches base cases, which are rules without preconditions or failure to match any rule.

We see that types categorize expressions. Every expression that is considered well-formed must be categorized into some type, either `int` or `bool`. Later we will introduce the syntax for propositions (called constraints in the MP literature) and whole programs. These too must be type checked. Propositions and programs will however be categorized into only one type. For example, propositions will either belong to the propositional type PROP or not. The judgements defining the well-formed propositions and programs are also called typing judgements, but the term type by itself usually refers to the categories of expressions.

The examples given in this section were for tutorial purposes only. We will not actually define numbers for we can rely on existing definitions. Also, the syntax we gave for expressions is rather simple; it does not even include variables. Expressions with variables require a more involved typing judgement than given above, but the essential idea is always that mathematical constructs must belong to some category to be considered well-formed.

4.2 Syntax

The first step in defining a language is to declare its syntax. The language being defined is called the object language, mathematical programs in our case. The language used to define the object language is called the meta-language, English augmented with the notation of type theory in this case. There are operations *within* the object language, such as addition, but also we define operations *on* the syntax, called meta-operations.

In this section, we first define the full syntax of the object language, which consists of expressions, types, propositions, and whole programs, and then define two basic meta-operations.

4.2.1 Full Forms

4.2.1.1 Types

The types of the language are

$$\tau ::= \texttt{real} \mid \texttt{bool} \tag{4.8}$$

The set of types is actually finite in this logic. It contains exactly two elements. In subsequent chapters, we will introduce more sophisticated logics with an infinity of types. Mathematical programs also require integers, and we will subsequently introduce `int` as a refinement of `real`.

4.2.1.2 Expressions

Expressions are given by the syntax

$$\begin{aligned} e ::= {} & x \mid r \mid \texttt{true} \mid \texttt{false} \\ & \mid -e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \\ & \mid \texttt{not}\, e \mid e_1 \,\texttt{or}\, e_2 \mid e_1 \,\texttt{and}\, e_2 \end{aligned} \tag{4.9}$$

Any lone variable x by itself is an expression. We are not being too specific about what constitutes a variable name, but it is understood to be some alphanumeric string. This is all a variable is. Existing MP software provide constructs which make it appear as if bounds or the "current" value are properties intrinsic to a variable. Treating variables as anything more than symbols couples ideas that should be kept distinct.

After variables, we provide a constant in the form r, which stands for any real constant. Actually, in our computer implementation, r is either an integer, e.g. 1, 2, or a rational, e.g. 0.0, 3.1. There does not exist a method for expressing the irrationals. Nonetheless, variables in MP can take any real value. So in general we have to allow r to denote also the irrationals. This is an awkward situation. Variables must be allowed to take values that we cannot express. That is however the situation in classical mathematics, and we are not attempting a constructive reformulation of MP.

The standard numeric operators are also provided. Again, these are interpreted classically, as in current practice. These calculations cannot actually be carried out on a computer, or by hand. They can in the case that the arguments happen to be integer or rational. There are various ways for encoding the distinction between operations on integers, rationals, and reals, but we do not address this matter in the current work.

Division is not provided. It is difficult to provide a proper theory of division because the denominator has to be of type "`real` except zero", but we have no such type. Enriching the types to include this concept complicates the theory in other ways, which we hope to address in future.

Finally, we provide the two Boolean constants `true` and `false`, and the standard Boolean operators. These pose none of the complications of their numeric counterpart.

4.2.1.3 Propositions

From expressions and types, we build propositions. The propositions of the language are

$$\begin{aligned} c ::= & \ \texttt{T} \mid \texttt{F} \\ & \mid \texttt{isTrue}\ e \mid e_1 = e_2 \mid e_1 \leq e_2 \\ & \mid c_1 \vee c_2 \mid c_1 \wedge c_2 \\ & \mid \exists x : \tau \centerdot c \end{aligned} \tag{4.10}$$

Firstly, we provide the basic propositions truth `T` and falsehood `F`. Propositional truth is distinct from Boolean truth, which is given by the Boolean constants `true` and `false`. A Boolean expression e can be converted into a proposition by prefixing it with the keyword `isTrue`. Distinguishing between Boolean expressions and propositions is important because certain operations are valid only on one or the other.

Numerical expressions are formed into propositions by comparing their magnitudes, giving equations and inequalities. The proposition $e_1 \geq e_2$ is not included because it can be viewed as a notational synonym for $e_2 \leq e_1$. It could be included if any algorithm benefitted from this distinction.

A disjunction and conjunction operator on propositions is provided. Conjunction $\wedge$ simply allows building up a set of propositions. Disjunction $\vee$ is the key novelty of disjunctive programming over pure mixed-integer programming. Operators $\vee$ and $\wedge$ operate on propositions, and are distinct from operators `or` and `and` on expressions.

The "$\centerdot$" in $\exists x : \tau \centerdot c$ is called Peano's dot and serves as an alternative to parentheses. It stands for a left parenthesis, and the matching right parenthesis is implicitly as far to the right as possible (Andrews, 2002, p. 15). The proposition $\exists x : \tau \centerdot c$ is read "there exists x of type τ such that proposition c holds". Although the existential quantifier is not common in MP, it is a useful extension. Variables can be introduced locally because the x has scope only in the body c of the existential proposition where it is introduced. There is no universal quantifier. Adding one would extend the language to include semi-infinite programs but we intentionally disallow this.

Occasionally, vector notation will be convenient. Let $\vec{x} = x_1, \ldots, x_m$ be a vector of variables, and similarly $\vec{\tau} = \tau_1 \times \cdots \times \tau_m$ a Cartesian product of m types. Then, $\exists \vec{x} : \vec{\tau} \centerdot c$ is an abbreviation for $\exists x_1 : \tau_1 \centerdot \ldots \exists x_m : \tau_m \centerdot c$. This is an abbreviation in the meta-language. There are no vectors in the object language.

4.2.1.4 Programs

Finally, a mathematical program is

$$p ::= \delta_{x_1:\tau_1,\ldots,x_m:\tau_m} \{e \mid c\} \tag{4.11}$$

where $\delta ::= \texttt{min} \mid \texttt{max}$. The definition is not inductive; programs cannot be constructed from other programs. So minimax problems for example cannot be expressed in our language.

A program consists fundamentally of four things: a direction of optimization δ, a list of optimization variables x_j, an objective e, and a proposition c. The proposition can be viewed as the feasible space of the problem. The space is the set of all points satisfying the proposition. The value of the objective e is considered at all points in this space, and the program returns the value that is the `min` (or `max`) of all such points. We have just conceptually defined the meaning of a program. Section 4.5 defines the semantics more precisely.

4.2.2 Free Variables

Knowing the variables occurring in a syntactic construct is a basic need. Variables occur in expressions, propositions, and programs. The equation $x = 3$ has a single *free variable* x. In contrast, the proposition $\exists x : \texttt{int} \,.\, x = 3$ has no free variables. There are however two occurrences of x. The existential quantifier is known as a *variable binder* because it introduces a variable, the first x. The second x is a use of this variable, and is a *bound variable*.

Consider the proposition

$$(\exists x : \texttt{int} \,.\, x = 3) \wedge (x = 4)$$

in which x occurs three times. The third x is unrelated to the first two. The first x gives a name to a variable that will be used further in the proposition, but only the second x is a use of this variable. The third x is an entirely different variable that had to have been introduced elsewhere.

The above proposition has a single free variable, the third x. Say we wish to set x to the value 5 in this proposition. The result is

$$(\exists x : \texttt{int} \,.\, x = 3) \wedge (5 = 4)\,,$$

leaving the bound variable, the second x, untouched.

Knowing the free variables of a syntactic construct is crucial. Without this we cannot understand the meaning of a statement as simple as "let x equal 5". We define how to calculate the set of free variables in expressions, propositions, and programs. A *closed* construct is one that has no free variables.

4.2.2.1 Free Variables of Expression

Let $FV\,(e)$ denote the set of free variables in expression e. The definition of this function is inductive on the construction of e. The notation of inference rules (4.2) is not used. That notation is reserved for judgements, which are n-ary relations on sets and implies that a top-down proof strategy is applicable. In contrast, $FV\,(e)$ is simply a function operating on expressions and returning a set of variables.

The definition is

1. $FV\,(x) = \{x\}$

2. $FV\,(r) = \emptyset$
3. $FV\,(\texttt{true}) = \emptyset$
4. $FV\,(\texttt{false}) = \emptyset$
5. $FV\,(\texttt{op}\,e) = FV\,(e)$, where $\texttt{op} \in \{-, \texttt{not}\}$
6. $FV\,(e_1\,\texttt{op}\,e_2) = FV\,(e_1) \cup FV\,(e_2)$, where $\texttt{op} \in \{+, -, *, \texttt{or}, \texttt{and}\}$.

Consider computing $FV\,(x + y)$. Expression $x + y$ is of a form governed by rule 6. It calls $FV\,(x)$ and $FV\,(y)$ which return $\{x\}$ and $\{y\}$, respectively. The union of these is $\{x, y\}$, and so $FV\,(x + y) = \{x, y\}$.

Let e CLOSED mean $FV\,(e) = \emptyset$. This kind of notation is frequently used. CLOSED is a unary relation, consisting of all expressions which have no free variables.

4.2.2.2 Free Variables of Proposition

Let $FV\,(c)$ denote the free variables of a proposition. Its definition is by induction on the form of c,

1. $FV\,(\texttt{T}) = \emptyset$
2. $FV\,(\texttt{F}) = \emptyset$
3. $FV\,(\texttt{isTrue}\ e) = FV\,(e)$
4. $FV\,(e_1\,\texttt{op}\,e_2) = FV\,(e_1) \cup FV\,(e_2)$, where $\texttt{op} \in \{=, \leq\}$
5. $FV\,(c_1\,\texttt{op}\,c_2) = FV\,(c_1) \cup FV\,(c_2)$, where $\texttt{op} \in \{\vee, \wedge\}$
6. $FV\,(\exists x : \tau\,\texttt{.}\,c) = FV\,(c) \setminus \{x\}$.

Propositions involve expressions. Thus, $FV\,(c)$ calls $FV\,(e)$. The interesting case is the existentially quantified proposition. The free variables of $\exists x : \tau\,\texttt{.}\,c$ are determined by first calculating the free variables of c and then removing x from this set.

Consider the previous example, $(\exists x : \texttt{int}\,\texttt{.}\,x = 3) \wedge (x = 4)$. This proposition is in the form $c_1 \wedge c_2$, so rule 5 is applied. There are no free variables in c_1 and $FV\,(c_2) = \{x\}$. Taking their union gives

$$FV\,((\exists x : \texttt{int}\,\texttt{.}\,x = 3) \wedge (x = 4)) = \{x\}\,.$$

The free x arises from the right conjunct, not the left.

Let c CLOSED mean $FV\,(c) = \emptyset$.

4.2.2.3 Free Variables of Program

Let $FV\,(p)$ denote the free variables of a program. The definition is given by the rule

1. $FV\,(\delta_{x_1:\tau_1,\ldots,x_m:\tau_m}\,\{e \mid c\}) = (FV\,(e) \cup FV\,(c)) \setminus \{x_1, \ldots, x_m\}$.

The idea is the same as for the existentially quantified proposition. In a program

$$\delta_{x_1:\tau_1,\ldots,x_m:\tau_m}\{e \mid c\},$$

variables $x_1, \ldots, x_m$ are bound in both e and c.

Let p CLOSED mean $FV(p) = \emptyset$. A well-formed program must always be closed. Variables must be introduced somewhere. Since programs are the top level construct, a free variable is necessarily an error.

4.2.3 Substitution

In the previous section, we defined procedures for calculating the free variables of expressions, propositions, and programs. It is often necessary to replace the free variables of a construct with some other expression. We gave the example of the proposition

$$(\exists x : \texttt{int}.\, x = 3) \wedge (x = 4)$$

with x set to 5, and saw that only the free occurrences of x got replaced. In this section we define the procedure for substituting an expression for a variable into either another expression or a proposition. Substituting into a program will not be needed since programs are forbidden to have free variables.

Let $x = x'$ mean the variables x and x' are identical and $x \neq x'$ mean they are distinct. We do not define these binary relations because variables are taken to be alphanumeric strings, and it is reasonable to assume that such an equality test is available.

4.2.3.1 Substitution into Expression

Let $\{e/x\}\, e'$ denote the substitution of e for free occurrences of x in e'. The definition of this procedure is inductive on the form of e',

1. $\{e/x\}\, x' = \begin{cases} \text{if } x = x' \text{ then } e \\ \text{else } x' \end{cases}$
2. $\{e/x\}\, r = r$
3. $\{e/x\}\, \texttt{true} = \texttt{true}$
4. $\{e/x\}\, \texttt{false} = \texttt{false}$
5. $\{e/x\}\, (\texttt{op}\, e) = \texttt{op}\ \{e/x\}\, e$, where $\texttt{op} \in \{-, \texttt{not}\}$
6. $\{e/x\}\, (e_1\, \texttt{op}\, e_2) = (\{e/x\}\, e_1\ \texttt{op}\ \{e/x\}\, e_2)$, where $\texttt{op} \in \{+, -, *, \texttt{or}, \texttt{and}\}$.

The definition parallels the definition of $FV(e)$. Variable x is free in the expression x' only if $x' = x$. If not, there are no occurrences of x in x', and no substitution should take place. Constants have no variables at all, so substituting into them has no effect. Substitution into other expressions recurses into their nested expressions.

Let $\{e_1/x_1, \ldots, e_m/x_m\}\, e$ denote the simultaneous substitution of each e_j for each x_j into e. The order in which the substitutions are done will not matter as long as $FV(e_j) \cap$

$\{x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_m\} = \emptyset$ for all $j = 1, \ldots, m$. This is imposed as a precondition of simultaneous substitution.

Vector notation is occasionally used. Let $\vec{x} = x_1, \ldots, x_m$ be a vector of variables, and similarly $\vec{e} = e_1, \ldots, e_m$. Then, $\{\vec{e}/\vec{x}\}\, e'$ is an abbreviation for $\{e_1/x_1, \ldots, e_m/x_m\}\, e'$.

4.2.3.2 Substitution into Proposition

Let $\{e/x\}\, c$ denote the substitution of e for x in c. The definition is by induction on the form of c,

1. $\{e/x\}\,(\mathtt{T}) = \mathtt{T}$
2. $\{e/x\}\,(\mathtt{F}) = \mathtt{F}$
3. $\{e/x\}\,(\mathtt{isTrue}\ e') = (\mathtt{isTrue}\ \{e/x\}\, e')$
4. $\{e/x\}\,(e_1\ \mathtt{op}\ e_2) = (\{e/x\}\, e_1\ \mathtt{op}\ \{e/x\}\, e_2)$, where $\mathtt{op} \in \{=, \leq\}$
5. $\{e/x\}\,(c_1\ \mathtt{op}\ c_2) = (\{e/x\}\, c_1\ \mathtt{op}\ \{e/x\}\, c_2)$, where $\mathtt{op} \in \{\vee, \wedge\}$
6. $\{e/x\}\,(\exists x' : \tau \centerdot c) =$
$$\begin{cases} \text{if } x \notin FV\,(\exists x' : \tau \centerdot c) \text{ then} \\ \quad \exists x' : \tau \centerdot c \\ \text{else if } x' \notin FV\,(e) \text{ then} \\ \quad \exists x' : \tau \centerdot \{e/x\}\, c \\ \text{else} \\ \quad \{e/x\}\,(\exists x'' : \tau \centerdot \{x''/x'\}\, c) \end{cases}$$
where in the final else branch, x'' chosen such that $x'' \notin (FV\,(c) \cup FV\,(e))$.

Substitution into the existentially quantified proposition, rule 6, requires explanation. We first check if x is free in $\exists x' : \tau \centerdot c$. If it is not, no substitution is required, and the proposition is returned unaltered. If it is free, substitution is required but we must be sure to avoid variable capture, which would occur if x' is free in e. The "else if" branch performs the substitution in the nested proposition c only if this is not the case. If x' is free in e, the substitution method in the final "else" clause must be used. This uses *α-conversion* to rename the variable being introduced by the existential quantifier, and then performs the substitution on this converted proposition.

A concrete example will help. Say the substitution being considered is

$$\{(x' + 1)/x\}\,(\exists x' : \tau \centerdot x' = 1 \wedge (x = 2))\,.$$

The x' in $x' + 1$ is unrelated to the x' in $x' = 1$ because the latter occurs within the scope of the existential quantifier. If we mistakenly perform the substitution without checking for the possibility of variable capture, we get

$$\exists x' : \tau \centerdot (x' = 1) \wedge ((x' + 1) = 2)\,,$$

which is erroneous. Now the x' in $x' + 1$ is within the scope the quantifier. The problem occurred because the bound variable x' is free in the expression $x' + 1$ being substituted for

x. The correct thing to do is to first rename the bound variable. We change the proposition $\exists x' : \tau \centerdot (x' = 1) \wedge (x = 2)$ to $\exists x'' : \tau \centerdot (x'' = 1) \wedge (x = 2)$, which does not alter the meaning of the proposition in any way, and then perform the substitution.

Let $\{e_1/x_1, \ldots, e_m/x_m\}\, c$ denote simultaneous substitution into a proposition, and require the same precondition as for simultaneous substitution into an expression.

4.3 Type System

A mathematical program is a syntactic construct in the form p. However, the syntax includes ill-formed programs. We now define a typing judgement for each syntactic construct. This defines the subset of the syntax that we wish to consider well-formed. Also, it categorizes the language's objects so that meta-operations can know the nature of the object being operated on.

First, we define another necessary construct. Let Γ be a context of variables. Its definition is

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau \tag{4.12}$$

which essentially defines a list of elements of the form $x : \tau$. A context can either be empty, or given a context, an item $x : \tau$ can be added to it. A context maintains a list of variable names and the types of those variables. It is assumed that variable names are unique.

A context is required to make sense of expressions and propositions with free variables. Whether the expression $x + 2$ is well-formed depends on the type of x, which must be provided by the context within which $x + 2$ occurs. The syntactic constructs τ, e, c, and p comprise the object language. In contrast, Γ is used only in the meta-language.

4.3.1 Well-Formed Type

Strictly, we need to know whether a type is well-formed. We have only two types and both are well-formed in this simple language. For completeness, let τ TYPE mean that τ is a well-formed type. The definition of this judgement is simply

$$\frac{}{\tau \text{ TYPE}} \tag{4.13}$$

Recalling our convention, this judgement is implicitly written for all τ. So there are really two rules defined here, one where τ is `real` and the other where it is `bool`. In later chapters, we introduce more sophisticated type theories where this judgement will be less trivial.

4.3.2 Well-Formed Context

Let Γ CTXT mean the context Γ is well-formed. Its definition simply requires all types in it to be well-formed,

$$\frac{}{\emptyset \text{ CTXT}} \tag{4.14a}$$

$$\frac{\tau \text{ TYPE} \quad \Gamma \text{ CTXT}}{\Gamma, x : \tau \text{ CTXT}} \tag{4.14b}$$

There are two rules, one for each form of a context. The context $\emptyset$ is well-formed without condition. Any context of the form $\Gamma, x : \tau$ is well-formed if the type τ being added is a well-formed type, and if the rest of the context is well-formed.

4.3.3 Type of Expression

Assigning a type to an expression requires a context to obtain information about its free variables. Let the ternary relation $\Gamma \vdash e : \tau$ mean, in context Γ, e is of type τ. The $\vdash$ is used in logic to denote a statement that is true under a certain hypothesis. $\Gamma \vdash e : \tau$ can also be read "under the hypothesis Γ, it is possible to prove that e is of type τ". The hypothesis in this case is an assumption about the types of the free variables.

The definition of $\Gamma \vdash e : \tau$ is inductive on the form of e. There is one rule for each form of e in the following definition,

$$\frac{}{\Gamma \vdash x : \tau} \text{ where } (x : \tau) \in \Gamma \tag{4.15a}$$

$$\frac{}{\Gamma \vdash r : \texttt{real}} \tag{4.15b}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \tag{4.15c}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}} \tag{4.15d}$$

$$\frac{\Gamma \vdash e : \texttt{real}}{\Gamma \vdash -e : \texttt{real}} \tag{4.15e}$$

$$\frac{\{\Gamma \vdash e_j : \texttt{real}\}_{j=1}^{2}}{\Gamma \vdash e_1 + e_2 : \texttt{real}} \tag{4.15f}$$

$$\frac{\{\Gamma \vdash e_j : \texttt{real}\}_{j=1}^{2}}{\Gamma \vdash e_1 - e_2 : \texttt{real}} \tag{4.15g}$$

$$\frac{\{\Gamma \vdash e_j : \texttt{real}\}_{j=1}^{2}}{\Gamma \vdash e_1 * e_2 : \texttt{real}} \tag{4.15h}$$

$$\frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{not}\, e : \texttt{bool}} \tag{4.15i}$$

$$\frac{\{\Gamma \vdash e_j : \texttt{bool}\}_{j=1}^{2}}{\Gamma \vdash e_1 \,\texttt{or}\, e_2 : \texttt{bool}} \tag{4.15j}$$

$$\frac{\{\Gamma \vdash e_j : \texttt{bool}\}_{j=1}^{2}}{\Gamma \vdash e_1 \,\texttt{and}\, e_2 : \texttt{bool}} \tag{4.15k}$$

The first line defines not just a single rule, but as many rules as there are elements in Γ. If a variable x is known to be of type τ in the given context, then, within that context, it is possible to conclude that x is of type τ. Most expressions of the form x are ill formed. Consider an x' not in Γ. No rule's conclusion matches this expression. Thus, $\Gamma \vdash x' : \tau$ is not element of the relation being defined. Alternatively, taking the proof theoretic view, under the hypothesis Γ, it is not possible to prove that $x' : \tau$.

The second rule declares every r to be of type `real`. The context Γ is not used in any preconditions; there are no preconditions. The conclusion is satisfied irrespective of the context. Rules for the Boolean constants are similar.

The rule for $-e$ concludes that $-e$ is of type `real` in a context Γ if, in that same context,

e is of type `real`. Binary operations are similar but they have two preconditions. The unary and binary operations on Booleans are analogous.

Consider type checking the expression $(y - 1) * x$ in the context $x : \texttt{real}, y : \texttt{real}$. This expression is of the form $e_1 * e_2$ for which there is one rule. The rule's conclusion

$$x : \texttt{real}, y : \texttt{real} \vdash (y - 1) * x : \texttt{real}$$

requires checking the two preconditions

$$x : \texttt{real}, y : \texttt{real} \vdash (y - 1) : \texttt{real}$$

$$x : \texttt{real}, y : \texttt{real} \vdash x : \texttt{real}$$

The first rule regards an expression of the form $e_1 - e_2$, and the second is of the variable form. By continuing the procedure, we eventually determine that these are both satisfied. Finally, the recursion returns to the original question, and $(y - 1) * x$ is determined to be of type `real`.

4.3.4 Well-Formed Proposition

Expressions are categorized into types. Propositions are also categorized but there is only one category, which we call PROP. Since there is only one propositional type, we did not bother to define its syntax. Nonetheless, the typing judgement for propositions is fundamentally similar to that of expressions. In both, the relevant construct must belong in some category.

Let $\Gamma \vdash c$ PROP mean, in context Γ, c is a well-formed proposition. The definition is inductive on the form of c,

$$\frac{}{\Gamma \vdash \texttt{T} \text{ PROP}} \quad (4.16a)$$

$$\frac{}{\Gamma \vdash \texttt{F} \text{ PROP}} \quad (4.16b)$$

$$\frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{isTrue}\ e \text{ PROP}} \quad (4.16c)$$

$$\frac{\Gamma \vdash e_1 : \texttt{real} \quad \Gamma \vdash e_2 : \texttt{real}}{\Gamma \vdash e_1 = e_2 \text{ PROP}} \quad (4.16d)$$

$$\frac{\Gamma \vdash e_1 : \texttt{real} \quad \Gamma \vdash e_2 : \texttt{real}}{\Gamma \vdash e_1 \leq e_2 \text{ PROP}} \quad (4.16e)$$

$$\frac{\Gamma \vdash c_1 \text{ PROP} \quad \Gamma \vdash c_2 \text{ PROP}}{\Gamma \vdash c_1 \vee c_2 \text{ PROP}} \quad (4.16f)$$

$$\frac{\Gamma \vdash c_1 \text{ PROP} \quad \Gamma \vdash c_2 \text{ PROP}}{\Gamma \vdash c_1 \wedge c_2 \text{ PROP}} \quad (4.16g)$$

$$\frac{\tau \text{ TYPE} \quad \Gamma, x : \tau \vdash c \text{ PROP}}{\Gamma \vdash \exists x : \tau \,.\, c \text{ PROP}} \quad (4.16h)$$

The propositions `T` and `F` are well-formed without condition. The first proposition `isTrue` e is well-formed if e is of type `bool`. A Boolean expression is a Boolean proposition. The keyword `isTrue` is provided to make the notation clear. Without it, it would not be clear if e refers to an expression or a proposition.

Equations and inequalities are well-formed if their left- and right-hand-sides are both of type real. Next, a disjunction or conjunction of well-formed propositions produces a well-formed proposition.

Finally, consider the rule for propositions of the form $\exists x : \tau \,.\, c$. This is the first rule demonstrating how variables get added to the context. In context Γ, this proposition is

a PROP if c is a PROP in the context Γ augmented with the declaration $x : \tau$. Since the existential quantifier introduces the variable x for use within its body, checking the body requires adding x to the context.

4.3.5 Well-Formed Program

Let p MP mean p is a well-formed mathematical program. No context is required because a program cannot have any free variables in it. There is just one rule because programs are of only one syntactic form

$$\frac{\begin{array}{c}\{\tau_j \text{ TYPE}\}_{j=1}^{m} \\ x_1 : \tau_1, \ldots, x_m : \tau_m \vdash e : \texttt{real} \\ x_1 : \tau_1, \ldots, x_m : \tau_m \vdash c \text{ PROP}\end{array}}{\delta_{x_1:\tau_1,\ldots,x_m:\tau_m} \{e \mid c\} \text{ MP}} \tag{4.17}$$

First, we check that the types of all the optimization variables are well-formed. The objective function and proposition can employ any of the declared optimization variables. Within the context $x_1 : \tau_1, \ldots, x_m : \tau_m$, the objective must be of type `real`, and the proposition must be a PROP.

Now, given any program within the syntax p, the top-down proof strategy can be used to check that it is well-formed. All nested propositions and expressions will get checked as the prover checks preconditions. Before showing some examples, we discuss type refinements and define the semantics of a program.

4.4 Refined Types

Thus far, variables can only be ascribed the type `real` or `bool`. In mathematical programs, however, we often wish to restrict the domain of the variable further. For example, numeric variables are sometimes restricted to integer values. Also, the compiler defined in the next chapter requires knowing whether certain variables are bounded, and if so, what those bounds are. We call such information a variable's domain. A domain can also be viewed as a refinement of a type τ. We now introduce a mechanism for declaring not only a variable's type, but a more restricted domain.

Firstly, the syntax for domains, or refined types, is

$$\begin{aligned}\rho ::= {} & \langle r_L, r_U \rangle \mid \langle r_L, \infty) \mid (-\infty, r_U \rangle \mid \texttt{real} \\ & \mid [r_L, r_U] \mid [r_L, \infty) \mid (-\infty, r_U] \mid \texttt{int} \\ & \mid \{\texttt{true}\} \mid \{\texttt{false}\} \mid \texttt{bool}\end{aligned} \tag{4.18}$$

Domain $\langle r_L, r_U \rangle$ denotes a bounded interval of reals, $\langle r_L, \infty)$ an interval bounded only from below, $(-\infty, r_U \rangle$ an interval with only an upper bound, and `real` an unbounded interval. `real` could be written as $(-\infty, \infty)$ to be consistent with the interval notation but we choose `real` to coincide with the type symbol. Integer intervals are defined similarly but are

denoted with square brackets. `int` is the unbounded interval of integers, i.e. the set of integers. {`true`} and {`false`} are singleton sets, refinements of `bool`.

A domain can be thought of as a subset of a type. For example, $\langle 1.0, 9.0\rangle$ is a subset of `real`. The relationship between domains and types is made precise with the judgement $\rho \subseteq \tau$, defined by the rules

$$\frac{}{\langle r_L, r_U\rangle \subseteq \texttt{real}} \quad (4.19a) \qquad \frac{}{(-\infty, r_U] \subseteq \texttt{real}} \quad (4.19g)$$

$$\frac{}{\langle r_L, \infty) \subseteq \texttt{real}} \quad (4.19b) \qquad \frac{}{\texttt{int} \subseteq \texttt{real}} \quad (4.19h)$$

$$\frac{}{(-\infty, r_U\rangle \subseteq \texttt{real}} \quad (4.19c) \qquad \frac{}{\{\texttt{true}\} \subseteq \texttt{bool}} \quad (4.19i)$$

$$\frac{}{\texttt{real} \subseteq \texttt{real}} \quad (4.19d) \qquad \frac{}{\{\texttt{false}\} \subseteq \texttt{bool}} \quad (4.19j)$$

$$\frac{}{[r_L, r_U] \subseteq \texttt{real}} \quad (4.19e) \qquad \frac{}{\texttt{bool} \subseteq \texttt{bool}} \quad (4.19k)$$

$$\frac{}{[r_L, \infty) \subseteq \texttt{real}} \quad (4.19f)$$

The syntax of the language can now be modified to allow specification of a variable's domain, instead of just its type. A variable's type could be declared in two places, existential quantifiers and the top level of a program. Instead of $\exists x : \tau \,\texttt{.}\, c$, we now allow the syntax $\exists x : \rho \,\texttt{.}\, c$, and instead of $\delta_{x_1:\tau_1,\ldots,x_m:\tau_m}\,\{e \mid c\}$, we allow $\delta_{x_1:\rho_1,\ldots,x_m:\rho_m}\,\{e \mid c\}$.

Domains do not affect our notion of a well-formed program. Given a program in this new syntax, we transform it into the original syntax for type checking. The proposition $\exists x : \rho \,\texttt{.}\, c$ is converted to $\exists x : \tau \,\texttt{.}\, c$, where $\rho \subseteq \tau$. Similarly, $\delta_{x_1:\rho_1,\ldots,x_m:\rho_m}\,\{e \mid c\}$ is converted to $\delta_{x_1:\tau_1,\ldots,x_m:\tau_m}\,\{e \mid c\}$, where $\rho_j \subseteq \tau_j$ for each j. There is thus no operator which could be defined to work only on the reals between 0.0 and 100.0, but not on other reals. Any operator is either valid on all reals or none.

What a domain declaration does do is restrict the values a variable can take. The declaration $x : \rho$ is like specifying a constraint on x. The judgement $x : \rho \backsimeq c$ associates a declaration $x : \rho$ with a proposition c. The definition of $\backsimeq$ is inductive on ρ,

$$\frac{}{x : \langle r_L, r_U\rangle \backsimeq r_L \le x \wedge x \le r_U} \quad (4.20a) \qquad \frac{}{x : (-\infty, r_U] \backsimeq x \le r_U} \quad (4.20g)$$

$$\frac{}{x : \langle r_L, \infty) \backsimeq r_L \le x} \quad (4.20b) \qquad \frac{}{x : \texttt{int} \backsimeq \texttt{T}} \quad (4.20h)$$

$$\frac{}{x : (-\infty, r_U\rangle \backsimeq x \le r_U} \quad (4.20c) \qquad \frac{}{x : \{\texttt{true}\} \backsimeq \texttt{isTrue}\ x} \quad (4.20i)$$

$$\frac{}{x : \texttt{real} \backsimeq \texttt{T}} \quad (4.20d) \qquad \frac{}{x : \{\texttt{false}\} \backsimeq \texttt{isTrue}\ (\texttt{not}\ x)} \quad (4.20j)$$

$$\frac{}{x : [r_L, r_U] \backsimeq r_L \le x \wedge x \le r_U} \quad (4.20e) \qquad \frac{}{x : \texttt{bool} \backsimeq \texttt{T}} \quad (4.20k)$$

$$\frac{}{x : [r_L, \infty) \backsimeq r_L \le x} \quad (4.20f)$$

In the special case that a domain is a type, no bounding information is provided. For more refined domains, a lower and/or upper bound are stated.

Despite the $x : \rho \trianglelefteq c$ judgement, domain declarations are not constraints. The declaration $\langle r_L, \infty)$ states a known fact, x is a value larger than r_L. In contrast, the proposition $r_L \leq x$ states no such thing—the value of x might be less than r_L. The proposition will be false in this case, but that might be okay, depending on the situation in which it occurs.

Retaining knowledge of bounds requires a context of variable domains. Similar to Γ, we define the refined context

$$\Upsilon ::= \emptyset \mid \Upsilon, x : \rho \tag{4.21}$$

In truth, both Γ and Υ must be maintained. However, Υ suffices in practice because it contains the information of Γ. The judgement $\rho \subseteq \tau$ allows synthesizing the coarser Γ from the more informative Υ. Given a refined context Υ, let $\Gamma(\Upsilon)$ be its corresponding coarse context. $\Gamma(\Upsilon)$ is defined inductively on the construction of Υ,

1. $\Gamma(\emptyset) = \emptyset$
2. $\Gamma(\Upsilon, x : \rho) = \Gamma(\Upsilon), x : \tau$ if

   $\rho \subseteq \tau$

Now, it is clear that refined contexts Υ can be maintained instead of coarse contexts Γ.

Nonetheless, the differing purposes of the two should be kept in mind. Γ provides information on the nature, type, of a variable. Υ provides some possible bounds, given that the nature of the variable is known. It so happens that the form of bounding information is dependent on the type being bounded, and so the type can be extracted from the bound, via $\rho \subseteq \tau$.

The compiler defined in the next chapter will need to know if certain variables have bounded domains. Let ρ BOUNDED mean ρ represents a bounded region. Its definition is given by the rules

$$\frac{}{\langle r_L, r_U \rangle \text{ BOUNDED}} \tag{4.22a}$$

$$\frac{}{[r_L, r_U] \text{ BOUNDED}} \tag{4.22b}$$

$$\frac{}{\{\texttt{true}\} \text{ BOUNDED}} \tag{4.22c}$$

$$\frac{}{\{\texttt{false}\} \text{ BOUNDED}} \tag{4.22d}$$

$$\frac{}{\texttt{bool BOUNDED}} \tag{4.22e}$$

For numeric domains, both an upper and lower bound is required. The domain `bool` and its refinements are all bounded. Actually, this judgement is needed only on numeric types; the Boolean domains are included only for completeness.

4.5 Semantics

Thus far we have defined the syntax of mathematical programs and a method for determining which programs are well-formed. However, we do not know what a program means. The language's type system declares the expression $2 + 3$ to exist. However, that this expression

equals 5 has never been stated. In this section, we define how to evaluate an expression, and then what it means for a proposition to be true and for a mathematical program to be solved.

We only define the meaning of a program, but do not provide an implementation of this meaning. In other words, our definition does not provide an algorithm for solving a mathematical program, but it does state exactly what the input-output relation of such an algorithm must be.

Only the semantics of well-formed constructs need to be defined. It is assumed in this section that all constructs satisfy their respective typing judgements.

4.5.1 Evaluation of Expression

It would not help much to say that $2 + 3$ evaluates to $1 + 4$. There is something special about the expression 5; we consider it fully evaluated while expressions $2 + 3$ and $1 + 4$ are reducible. Expressions that cannot be further reduced are called canonical expressions. Let e CANONICAL be a judgement defined by the rules

$$\frac{}{r \text{ CANONICAL}} \tag{4.23a}$$

$$\frac{}{\texttt{true} \text{ CANONICAL}} \tag{4.23b}$$

$$\frac{}{\texttt{false} \text{ CANONICAL}} \tag{4.23c}$$

Canonical expressions are also called constants or values, but these terms can be somewhat misleading in the richer languages we introduce in later chapters. They are important enough to justify a special notation. Let v denote an expression satisfying v CANONICAL.

We now define the evaluation of any closed expression to a canonical form. Open expressions must first be assigned values for their free variables, as will be discussed subsequently. Given e CLOSED, $e \searrow v$ means e evaluates to v. Its definition is inductive on the form of e,

1. $r \searrow r$
2. $\texttt{true} \searrow \texttt{true}$
3. $\texttt{false} \searrow \texttt{false}$
4. $-e \searrow \begin{cases} r \text{ if } e \searrow -r \\ -r \text{ if } e \searrow r \end{cases}$
5. $e_1 \,\texttt{op}\, e_2 \searrow r$ if
   $e_1 \searrow r_1$ and $e_2 \searrow r_2$, where $r_1 \,\mathrm{op}\, r_2 = r$, for $\texttt{op} \in \{+, -, *\}$
6. $\texttt{not}\, e \searrow \begin{cases} \texttt{true} \text{ if } e \searrow \texttt{false} \\ \texttt{false} \text{ if } e \searrow \texttt{true} \end{cases}$
7. $e_1 \,\texttt{or}\, e_2 \searrow \begin{cases} \texttt{true} \text{ if either } e_1 \searrow \texttt{true} \text{ or } e_2 \searrow \texttt{true} \\ \texttt{false} \text{ otherwise} \end{cases}$

8. $e_1 \,\texttt{and}\, e_2 \searrow \begin{cases} \texttt{true} \text{ if } e_1 \searrow \texttt{true} \text{ and } e_2 \searrow \texttt{true} \\ \texttt{false} \text{ otherwise} \end{cases}$.

There is no rule for expressions of the form x, which violate the precondition that the expression be closed. Constants evaluate to themselves. Next, consider rule 5, for the form $e_1 + e_2$ for concreteness. According to the rule, this expression evaluates to r if e_1 evaluates to r_1, e_2 evaluates to r_2, and $r_1 + r_2 = r$. The last requirement can be confusing if the distinction between object language and meta-language is not kept in mind. The $+$ in $e_1 + e_2$ is an operator in the object language, the language we are defining. The $+$ in $r_1 + r_2 = r$ is an operation in the meta-language. We take as a given in the meta-language some method for adding numeric constants. This method is employed in defining the operations within the object language. The identical distinction must be kept in mind for the other operators.

Our definitions for evaluating numerical expressions take a classical approach to mathematics. There is in fact no computational method for executing these operations. This however coincides with current practice and is sufficient for our goals. Providing a computational theory for the reals is a significant challenge being addressed by others.

4.5.2 Truth of Proposition

Let c TRUE mean proposition c is true, assuming c CLOSED. The truth of an open proposition requires first assigning values to its free variables and is discussed subsequently. The definition of c TRUE is inductive on the form of c,

1. $\texttt{T}$ TRUE
2. $(\texttt{isTrue}\ e)$ TRUE if

   $e \searrow \texttt{true}$
3. $(e_1 \,\texttt{op}\, e_2)$ TRUE if

   $e_1 \searrow r_1$ and $e_2 \searrow r_2$ and $r_1 \,\texttt{op}\, r_2$, for $\texttt{op} \in \{=, \leq\}$
4. $(c_1 \vee c_2)$ TRUE if

   either c_1 TRUE or c_2 TRUE
5. $(c_1 \wedge c_2)$ TRUE if

   both c_1 TRUE and c_2 TRUE
6. $(\exists x : \rho \,\texttt{.}\, c)$ TRUE if

   $\{v/x\}\, c$ TRUE for some $v \in \rho$

We now have at least three notions of truth: the Boolean constant `true`, the proposition `T`, and the truth of a proposition given by c TRUE. The symbol `T` must not be confused with the interpretation of it as true. It is interpreted as true only because we have defined it so in rule 1. These distinctions and their importance are discussed by Martin-Löf (1987, 1996). There is no rule for the proposition `F`, meaning `F` TRUE is never true.

By rule 2, Boolean propositions are interpreted as true if the corresponding expression evaluates to `true`. By rule 3, an equation is satisfied if its left- and right-hand-sides evaluate to constants that are equal. Analogously for inequalities. Again, we take a classical approach. There is in fact no computer implementation for comparing two real numbers.

In rule 4, we define $c_1 \vee c_2$ to be true if either c_1 is true or c_2 is true. This follows the constructivists' view of disjunction. Classically, one might prove the truth of $c_1 \vee c_2$ in various other ways, with proof by contradiction for example. One would prove that not $(c_1 \vee c_2)$ TRUE implies something absurd, such as `F` TRUE. Our constructive definition of disjunction does not accept this on the basis that, in practice, if we declare $c_1 \vee c_2$ to be true, we probably have in mind that either c_1 is true or that c_2 is true. Proof by contradiction does not provide any such information. It allows concluding that $c_1 \vee c_2$ is true without actually knowing either that c_1 is true or that c_2 is true.

We also require a constructive proof for the truth of a conjunction $c_1 \wedge c_2$. When we declare $c_1 \wedge c_2$ to be true, we are normally thinking that in fact we know that both c_1 is true and that c_2 is true. Our definition demands this. Proof by contradiction does not.

Finally, when we declare $\exists x : \rho \,\texttt{.}\, c$ to be true, we normally think that we now know that c is true for some particular value of x. This value v is called the witness because it allows us to witness the truth of c. Proof by contradiction would not provide any such witness.

Existential quantification is a generalization of disjunction, and the definitions of their truth are analogous. The proposition $\exists x : \rho \,\texttt{.}\, c$ effectively declares an infinity of disjuncts, c for each choice of x. By requiring that c be true for a particular choice of x, we are requiring knowledge of which one of these disjuncts is true. This is the same requirement as in binary disjunction, where we required that either c_1 be true or that c_2 be true.

We have, at various points now, provided classical and constructive definitions. So clearly, our aim is not to take one view or the other. We are attempting only to provide a logical formulation of mathematical programming as it is currently practiced. In current practice, real numbers are treated classically. However, it seems that the truth of a proposition is treated constructively. For example, consider the proposition

$$\exists x_1 : \texttt{real} \,\texttt{.}\, \exists x_2 : \texttt{real} \,\texttt{.}\, (x_1 + x_2 = 3.0) \wedge (x_1 + 2.0 * x_2 = 6.0)\,,$$

which is just a system of linear equations. In present algorithms, this system of equations is certainly satisfied by finding witnesses for x_1 and x_2 such that the equations are satisfied. Furthermore, the conjunction of the two equations is certainly shown to be true by showing that each of the equations is true. Witnesses are also demanded by users of MP software. One wants to know the values of the variables for which an equation is satisfied. We thus believe that our definitions coincide closely to current practice.

4.5.3 Solution of Mathematical Program

Finally, we state what it means to solve a mathematical program. There are a few possibilities: the program can have an optimum, it can be infeasible, it can be unbounded, or, for nonlinear programs, it could be bounded but have no optimum. The final possibility also exists for linear programs with irrational constants. We simplify these possibilities into two,

either the program has an optimum or not.

Let

$$r_{\text{option}} ::= \text{NONE} \mid \text{SOME}(r) \tag{4.24}$$

be an optional real number. This is basically the set of reals augmented with the value NONE. It allows us to correctly state that a function might return a real number or perhaps fail to do so. Now, let $p \twoheadrightarrow r_{\text{option}}$ mean the solution to program p is r_{option}. If $p \twoheadrightarrow \text{NONE}$, the judgement means the program does not have an optimum. If $p \twoheadrightarrow \text{SOME}(r)$, then there is an optimal solution r. Instead of r_{option}, we could have defined a set with additional elements, such as UNBOUNDED and INFEASIBLE. This would provide more specific information about why there is no optimum, but the simpler form suffices for our purposes.

Prior to defining $\twoheadrightarrow$, we introduce some notation that will ease its definition. Given a program $\delta_{x_1:\rho_1,\ldots,x_m:\rho_m}\{e \mid c\}$, let $\vec{\rho} = \rho_1 \times \cdots \times \rho_m$ be the Cartesian product of the program variables' types. Then $\vec{v} \in \vec{\rho}$ is an m-tuple, and v_j denotes its j^{th} component. Also, let F be the feasible space of the program. We have $F = \{\vec{v} \in \vec{\rho} \mid \{\vec{v}/\vec{x}\}\, c \text{ TRUE}\}$, i.e. all points in the space $\vec{\rho}$ for which proposition c is satisfied.

Then, the definition of $\twoheadrightarrow$ is given by the rules

1. $\texttt{min}_{x_1:\rho_1,\ldots,x_m:\rho_m}\{e \mid c\} \twoheadrightarrow$
$$\begin{cases} \text{if } F = \emptyset \text{ then} \\ \quad \text{NONE} \\ \text{else if } \exists \vec{v} \in F \,.\, \forall \vec{v}' \in F \,.\, r \leq r' \text{ then} \\ \quad \text{SOME}(r) \\ \text{else} \\ \quad \text{NONE} \end{cases}$$

2. $\texttt{max}_{x_1:\rho_1,\ldots,x_m:\rho_m}\{e \mid c\} \twoheadrightarrow$
$$\begin{cases} \text{if } F = \emptyset \text{ then} \\ \quad \text{NONE} \\ \text{else if } \exists \vec{v} \in F \,.\, \forall \vec{v}' \in F \,.\, r \geq r' \text{ then} \\ \quad \text{SOME}(r) \\ \text{else} \\ \quad \text{NONE} \end{cases}$$

where $\{\vec{v}/\vec{x}\}\, e \searrow r$ and $\{\vec{v}'/\vec{x}\}\, e \searrow r'$.

The two rules are similar. Consider minimization for concreteness. First we check if the feasible space is empty. If so, we can immediately conclude that there is no optimum. In the "else if" branch, we seek a feasible point $\vec{v}$ such that the value of the objective function r at this point is less than or equal to the value r' at all other feasible points. There might not be such a point—the feasible space might be unbounded or bounded but with only an infimum—in which case the "else" branch returns NONE. The rule for maximization is identical except that r must be greater than or equal to all r'.

As with existential propositions, we have required that a solution of a mathematical program provide a witness $\vec{v}$, when there is an optimum. This clearly coincides with how existing algorithms work and with the demands we make of MP software. We would like to know not only the optimum solution but also the point at which it occurs.

An algorithm f for solving a mathematical program operates on the space of well-formed programs and returns a solution, either the optimum or a statement that there is no opti-

mum. Any algorithm should be sound and complete with respect to the semantics we have defined. Soundness means it is correct; if $f(p) = r_{\text{option}}$, then $p \twoheadrightarrow r_{\text{option}}$ should be true. Completeness means the algorithm works for all mathematical programs; if $p \twoheadrightarrow r_{\text{option}}$, then $f(p) = r_{\text{option}}$.

4.5.4 Open Forms

So far we have defined the meaning of closed forms. Programs must be closed, so this suffices. However, expressions and propositions can have free variables, and it is useful to understand their meaning with free variables. This requires stating values for the variables. We cannot ask what $x + 3$ evaluates to or whether the equation $x = 3$ is satisfied, absent a value for x. A *valuation* is a list of variables with assigned values. Precisely, it is defined by

$$\Psi ::= \emptyset \mid \Psi, x = v \tag{4.25}$$

where $x = v$ means x is assigned the value v.

Now, let $\Psi \vdash e \searrow v$ mean, under the valuation Ψ, e evaluates to v. It is required that Ψ include a value for all the free variables in e. This judgement is defined by induction on the form of Ψ,

1. $\emptyset \vdash e \searrow v$ if

   $e \searrow v$

2. $\Psi, x = v' \vdash e \searrow v$ if

   $\Psi \vdash \{v'/x\}\, e \searrow v$

If Ψ is empty, then e must be closed, and evaluation on closed expressions, defined in the previous section, is employed. If Ψ contains an assignment $x = v$, then this value is substituted into e, and we recurse.

Similarly, let $\Psi \vdash c$ TRUE mean proposition c, not necessarily closed, is true under the valuation Ψ. Its definition is

1. $\emptyset \vdash c$ TRUE if

   c TRUE

2. $\Psi, x = v \vdash c$ TRUE if

   $\Psi \vdash \{v/x\}\, c$ TRUE

4.6 Results

An alternative logical definition of mathematical programs has been provided in this chapter. The linguistic emphasis of logic leads to a definition that is simultaneously a convenient computer language. Also, an expression, proposition, and whole program are now as formal a mathematical object as is an integer. Just as we define functions on integers, we can now define operations on proposition and program spaces. Converting a general mathematical

program to a pure mixed-integer program is exactly such an operation, and we will define it in the next chapter. Furthermore, the rigor demanded by type theory exposes subtle details not previously considered. For example, there is a difference between Boolean conjunction and propositional conjunction. The importance of this distinction will be seen when we convert Boolean expressions into integer propositions.

We also defined the semantics of a mathematical program. This specifies what the input-output relation of an algorithm for solving an MP should be. It exposes the challenges of implementing algorithms on a computer, namely real computation, but also, it elucidates the information we want from algorithms. Current software provide the feasible point and the optimum, but often one wants to know why the feasible point is feasible. There is little to no support for this kind of analysis. The information sought is the proofs we discuss in our interpretation of an MP. They allow understanding why a program has the optimum it does or why the problem is infeasible.

A benefit to developers of MP software is that the definitions are thorough and serve as a specification of the software architecture. The software implementation of the judgement p MP is nearly identical to its mathematical definition, provided in this chapter. As software gets more sophisticated, the formal approach of type theory will be required to design it.

The implementation does differ from the theory in minor ways. We did not include the printing of error messages in the theory, but the following example will show that error messages arise directly from the judgements we did define. Also, our parser only supports ASCII text. So, as discussed in Appendix C, the concrete syntax differs from the abstract syntax used in the theory, but this is a minor detail.

A user can now write a program in the syntax p, which is both similar to how mathematics is written on paper and formal enough to be understood by a computer. Then, the judgement p MP can be tested automatically. We show some examples of our software's input and output. The input is a program in the syntax p, and the output provides error messages in case p MP fails to hold.

Example 4.1 The following program involves a simple disjunctive constraint.

```
1 var x:real
2 var y:real
3
4 min x subject_to
5 (isTrue y, x <= 3.0) disj (isTrue (not y), x >= 4.0)
```

In the concrete syntax, the keyword `disj` represents propositional disjunction $\vee$, and a simple comma denotes propositional conjunction $\wedge$.

We check the judgement p MP on the above program and get the following error messages,

```
ERROR: type analysis failed
  context: x:real, y:real
  expr at 5.9: y
  type: bool

MSG: ill-formed prop, previous messages should explain why
```

```
context: x:real, y:real
prop at 5.2-5.9: isTrue y
```

The first error message states that type analysis, the judgement $\Gamma \vdash e : \tau$, failed. The particular Γ, e, and τ that were being checked are printed. The expression is stated to be on line 5, column 9. One can refer back to the program to see which expression this is, but also it is printed within the message. (Longer expressions are not re-printed in their entirety to avoid clutter.) The message tells us that y must be of type `bool` for the program to be well-formed.

In a more complex program, this message by itself may not help. We may wonder why this needs to be true. Subsequent messages show why this judgement was called. The next message says the judgement

$$x : \texttt{real}, y : \texttt{real} \vdash \texttt{isTrue}\ y\ \text{PROP}$$

is failing. Now we know why the previous judgement was being checked. For `isTrue` y to be a well-formed proposition, y must be of type `bool`.

This message is labeled `MSG` to indicate that it was not the end source of an error. Rather this judgement failed due to its precondition failing, which was the previous message labeled `ERROR`. Additional messages are also printed, but we have shown only the relevant ones. The entire call stack leading to the check p MP is generally printed.

Our error is now evident. We accidentally declared y to be of type `real` but meant it to be of type `bool`. We fix the program,

```
1 var x:real
2 var y:bool
3
4 min x subject_to
5 (isTrue y, x <= 3.0) disj (isTrue (not y), x >= 4.0)
```

Again, we check the judgement p MP, and this time there are no errors.

Example 4.2 In this example, we show that propositional and Boolean operations are distinct. Consider the program

```
1 var x:real
2 var y1:bool
3 var y2:bool
4
5 min x subject_to
6 isTrue y1 and y2 and
7 x = 0
```

Here, the software does not even get to the type checking phase. It prints the message

```
ERROR at 6.1-6.6: syntax error
```

which says that the given program is not even of the syntax p. The reason is we have accidentally used Boolean conjunction `and` when we were trying to denote propositional conjunction $\wedge$.

The program is corrected to

```
var x:real
var y1:bool
var y2:bool

min x subject_to
isTrue y1 and y2,
x = 0
```

We have replaced the second `and` with propositional conjunction, which in the concrete syntax can be denoted with a comma. The program now is of a valid syntax (and also type checks).

These examples demonstrate that we have a rigorous method for checking that a program falls within the set of mathematical programs according to our definition. In the next chapter, we define functions operating on this set.

Chapter 5

Compiling Mathematical Programs

One benefit of our definition of MP is that solution methods can be developed on the full structure, taking advantage of Boolean solvers and direct disjunctive techniques. Nonetheless, most existing algorithms operate only on the sub-language known as mixed-integer programming (MIP), which allows only a conjunction of (in)equations on reals and integers. See Figure A.1 on page 182 for a description of MP and its sub-languages.

It is possible to convert a general MP to a pure MIP (under certain conditions). The two main tasks are to convert Boolean expressions into integer constraints and disjunctive constraints into mixed-integer constraints. Currently, these are manual tasks, which has several drawbacks: one must be familiar with the transformation methods, the manual labor is tedious and error-prone, and it is difficult to analyze the procedure. Automation would enable wider application of MIP solvers, produce trustworthy models, greatly speedup the modeling process, and allow improvements to be investigated within a formal theory.

In this chapter, we define a mapping from MP to MIP. The basic ideas follow from work by others and are reviewed in Appendix A. Our contribution is defining these transformations on the richer syntax provided in the previous chapter and defining them precisely enough for implementation on a computer. We will see that these definitions depend crucially on the type system of MP defined in the previous chapter.

Converting from one language into another is called compiling in programming language terminology. Although compilers are usually thought of as generating machine executable code, the fundamental idea is the same here. The compiler definition elucidates one benefit of formalizing mathematical programs as a logic. It enables treating programs as formal objects. Program spaces can then be used as the domain and codomain of a function, the compiler.

5.1 Sub-Languages

A compiler transforms a source language into a target language. In our case, the target language is MIP and happens to be a sub-language of the source language MP. Defining an

MP was the purpose of the previous chapter. We now restrict that definition to the class of MIPs, which is easy to do.

Strictly, the compiler we will define is valid for nonlinear MPs, but it is a poor transformation in this case. Our main applications are for programs that are linear, and we also define this class. Finally, we will define the subset of programs whose disjuncts are bounded, which is a precondition of compilation.

5.1.1 Mixed-Integer Programs

The full syntax of MP involves types τ, refined types ρ, expressions e, propositions c, and programs p. A subset of that syntax comprises what we consider a MIP. Let τ^{MIP}, ρ^{MIP}, e^{MIP}, c^{MIP}, and p^{MIP} refer to the restricted syntax. These are defined by

$$\tau^{\text{MIP}} ::= \texttt{real}$$

$$\rho^{\text{MIP}} ::= \langle r_L, r_U \rangle \mid \langle r_L, \infty) \mid (-\infty, r_U \rangle \mid \texttt{real} \tag{5.1}$$

$$\mid [r_L, r_U] \mid [r_L, \infty) \mid (-\infty, r_U] \mid \texttt{int} \tag{5.2}$$

$$e^{\text{MIP}} ::= x \mid r$$

$$\mid -e^{\text{MIP}} \mid e_1^{\text{MIP}} + e_2^{\text{MIP}} \mid e_1^{\text{MIP}} - e_2^{\text{MIP}} \mid e_1^{\text{MIP}} * e_2^{\text{MIP}} \tag{5.3}$$

$$c^{\text{MIP}} ::= \texttt{T} \mid \texttt{F}$$

$$\mid e_1^{\text{MIP}} = e_2^{\text{MIP}} \mid e_1^{\text{MIP}} \leq e_2^{\text{MIP}}$$

$$\mid c_1^{\text{MIP}} \wedge c_2^{\text{MIP}}$$

$$\mid \exists x : \rho^{\text{MIP}} \centerdot c^{\text{MIP}} \tag{5.4}$$

$$p^{\text{MIP}} ::= \delta_{x_1:\rho_1^{\text{MIP}},\ldots,x_m:\rho_m^{\text{MIP}}} \{e^{\text{MIP}} \mid c^{\text{MIP}}\} \tag{5.5}$$

There are basically two restrictions: MIPs do not allow disjunctive propositions nor the type `bool`. Due to the latter, we exclude Boolean constants and operators from expressions because these could not possibly type check.

Superscripts allowed the definitions to be made in the compact form. We also use the judgement form; for example, p MIP means p is a program in the restricted syntax p^{MIP}.

The following definitions will also be needed,

$$\Upsilon^{\text{MIP}} ::= \emptyset \mid \Upsilon^{\text{MIP}}, x : \rho^{\text{MIP}} \tag{5.6}$$

$$\Psi^{\text{MIP}} ::= \emptyset \mid \Psi^{\text{MIP}}, x = v^{\text{MIP}} \tag{5.7}$$

where v^{MIP} stands for an expression satisfying e CANONICAL and e MIP. Only numeric constants r satisfy this form, i.e. MIP values are numbers.

5.1.2 Linearity

Programs that are linear are significantly easier to solve than nonlinear ones, and customized algorithms are applied to them. We define a method for determining the subset of programs that are linear. We avoid calling these linear programs (LPs) because that is a standard term referring to programs with further restrictions. Our definition checks that the numeric propositions in a program do not involve multiplication of two variables. It simply disregards Boolean propositions, and disjunctive propositions are declared linear if the equations and inequalities in them are.

This terminology is common but can be misleading. It is odd to declare a Boolean proposition linear—multiplication is not even a valid concept within it. It is similarly awkward to define disjunctive propositions as linear because they represent nonconvex regions. Really, the terminology refers to programs that can be transformed to programs that are linear, not ones that are linear themselves. Programs so classified could be converted to nonlinear programs also, depending on the transformation method used.

Let e LINEAR mean expression e can be transformed to a linear numeric expression. Its definition is

$$\frac{}{x \text{ LINEAR}} \tag{5.8a}$$

$$\frac{}{r \text{ LINEAR}} \tag{5.8b}$$

$$\frac{}{\texttt{true} \text{ LINEAR}} \tag{5.8c}$$

$$\frac{}{\texttt{false} \text{ LINEAR}} \tag{5.8d}$$

$$\frac{e \text{ LINEAR}}{-e \text{ LINEAR}} \tag{5.8e}$$

$$\frac{e_1 \text{ LINEAR} \quad e_2 \text{ LINEAR}}{e_1 + e_2 \text{ LINEAR}} \tag{5.8f}$$

$$\frac{e_1 \text{ LINEAR} \quad e_2 \text{ LINEAR}}{e_1 - e_2 \text{ LINEAR}} \tag{5.8g}$$

$$\frac{}{e_1 * e_2 \text{ LINEAR}} \text{ if } FV(e_1 * e_2) = \emptyset \tag{5.8h}$$

$$\frac{e_2 \text{ LINEAR}}{e_1 * e_2 \text{ LINEAR}} \text{ if } FV(e_1) = \emptyset \tag{5.8i}$$

$$\frac{e_1 \text{ LINEAR}}{e_1 * e_2 \text{ LINEAR}} \text{ if } FV(e_2) = \emptyset \tag{5.8j}$$

$$\frac{}{\texttt{not}\, e \text{ LINEAR}} \tag{5.8k}$$

$$\frac{}{e_1 \,\texttt{or}\, e_2 \text{ LINEAR}} \tag{5.8l}$$

$$\frac{}{e_1 \,\texttt{and}\, e_2 \text{ LINEAR}} \tag{5.8m}$$

The interesting rule is for multiplication $e_1 * e_2$. Either e_1 or e_2 must not contain any variables. Other arithmetic expressions simply check their sub-expressions recursively.

Let c LINEAR mean any expression e in c satisfies e LINEAR. We do not give the definition explicitly. It simply checks that all nested expressions within the proposition are linear. A program p is linear, p LINEAR, if its objective and feasible space are. Precisely,

$$\frac{e \text{ LINEAR} \quad c \text{ LINEAR}}{\delta_{x_1:\rho_1,\ldots,x_m:\rho_m} \{e \mid c\} \text{ LINEAR}} \tag{5.9}$$

5.1.3 Mixed-Integer Linear Programs

Finally, let p MILP mean program p is a mixed-integer linear program. Its definition is

$$\frac{p \text{ MIP} \quad p \text{ LINEAR}}{p \text{ MILP}} \tag{5.10}$$

According to Nemhauser and Wolsey (1999, p. 4), the numeric parameters of an MILP should be rational. Otherwise an optimization problem might have an infimum or supremum but no optimum. Strictly, the definition above does not check for this condition. However, our computer implementation anyways has no method for expressing irrational constants. So programs expressed in our software satisfy this requirement by default.

5.2 Conjunctive Normal Form

Boolean expressions are converted into integer constraints by first putting them into conjunctive normal form (CNF). We first define CNF and then provide a method for putting arbitrary expressions into this form. From the definition of $\Gamma \vdash e : \tau$, it is clear that only expressions of the form x, `true`, `false`, `not` e, e_1 `or` e_2, and e_1 `and` e_2 could possibly be involved in a Boolean expression. All discussions in this section disregard other expression forms.

5.2.1 Definition of CNF

The definition of CNF requires two auxiliary forms to be defined first, called literal and disjunctive literal forms. Let e LITERAL mean expression e is a literal. It is defined by the rules

$$\frac{}{x \text{ LITERAL}} \tag{5.11}$$

$$\frac{}{\texttt{true} \text{ LITERAL}} \tag{5.12}$$

$$\frac{}{\texttt{false} \text{ LITERAL}} \tag{5.13}$$

$$\frac{e \text{ LITERAL}}{\texttt{not}\, e \text{ LITERAL}} \tag{5.14}$$

A literal form is a lone variable, a Boolean constant, or a negation of these. Some definitions of CNF first introduce an atomic form and require a negated expression to be in atomic form. This will not be needed for our purposes.

Let e DLF mean expression e is in disjunctive literal form. Expressions satisfying the following rules are in this form,

$$\frac{e \text{ LITERAL}}{e \text{ DLF}} \tag{5.15a}$$

$$\frac{e_1 \text{ DLF} \quad e_2 \text{ DLF}}{e_1 \,\texttt{or}\, e_2 \text{ DLF}} \tag{5.15b}$$

Disjunctive literal forms include all literals and extend to include disjunctions.

Finally, let e CNF be the relation defining conjunctive normal forms. Its definition is

$$\frac{e \text{ DLF}}{e \text{ CNF}} \tag{5.16a}$$

$$\frac{e_1 \text{ CNF} \quad e_2 \text{ CNF}}{e_1 \texttt{ and } e_2 \text{ CNF}} \tag{5.16b}$$

Conjunctive normal forms include all disjunctive literal forms and extend to include conjunction.

The negation of the judgements will be written by prefixing with a $\neg$, e.g. e $\neg$LITERAL means e is not a literal. We will need to refer to CNF expressions that are not DLF. Let e CONJ be defined by

$$\frac{e \text{ CNF} \quad \neg e \text{ DLF}}{e \text{ CONJ}} \tag{5.17}$$

The name of this judgement is motivated by the following observation.

Remark 5.1. *e CONJ is satisfied only by expressions e in the form e_1 `and` e_2.*

Proof. Obvious from the definition of e CNF and e DLF. Any Boolean expression e must be in the form x, `true`, `false`, `not` e', e_1 `or` e_2, or e_1 `and` e_2. If it is any of these except e_1 `and` e_2, it will satisfy e DLF. □

5.2.2 Transforming to CNF

Let $e_1 \curvearrowright e_2$ be a binary relation on expressions. It relates an arbitrary Boolean expression e_1 to its conjunctive normal form e_2. Its definition is not just inductive on e_1. The form of e_1 is first considered, but further induction on its nested expressions is required. The basic idea follows known techniques; see for example Visser (2001) or Chang and Lee (1987, p. 12–15). Our definition can be considered tutorial. It demonstrates the set theoretic approach we espouse for describing such transformations. This is the only tenable approach for defining more complex program transformations needed later.

The main judgement $e_1 \curvearrowright e_2$ takes an expression e_1 and returns an expression e_2 such that e_2 CNF. The definition of $\curvearrowright$ is

$$\frac{e \text{ CNF}}{e \curvearrowright e} \tag{5.18a}$$

$$\frac{e_1 \ \neg\text{CNF} \quad e_1 \curvearrowright_* e_2}{e_1 \curvearrowright e_2} \tag{5.18b}$$

If the expression is already in CNF, nothing is done, i.e. the relation is reflexive. Expressions not in CNF are passed to an auxiliary relation $e_1 \curvearrowright_* e_2$, which is defined only for e_1 such that e_1 $\neg$CNF.

The definition of $e_1 \curvearrowright_* e_2$ is inductive on e_1 and its nested forms. Only expression forms which could possibly be Boolean need to be considered—there are six: x, `true`, `false`, `not` e, e_1 `or` e_2, and e_1 `and` e_2. Three have no nested expressions, one is a unary operator, and two are binary operators. Thus, up to $1+1+1+6+36+36 = 81$ rules could be needed. The total is reduced because some are already in CNF, a simple observation allows covering all `and` expressions with a single rule, and a less simple observation allows covering `or` expressions

with a few rules. The rules are

$$\frac{e_1 \curvearrowright e'}{\texttt{not not}\, e_1 \curvearrowright_* e'} \tag{5.19a}$$

$$\frac{(\texttt{not}\, e_1)\ \texttt{and}\ (\texttt{not}\, e_2) \curvearrowright e'}{\texttt{not}\ (e_1\ \texttt{or}\ e_2) \curvearrowright_* e'} \tag{5.19b}$$

$$\frac{(\texttt{not}\, e_1)\ \texttt{or}\ (\texttt{not}\, e_2) \curvearrowright e'}{\texttt{not}\ (e_1\ \texttt{and}\ e_2) \curvearrowright_* e'} \tag{5.19c}$$

$$\frac{(e_{11}\ \texttt{or}\ e_2)\ \texttt{and}\ (e_{12}\ \texttt{or}\ e_2) \curvearrowright e'}{(e_{11}\ \texttt{and}\ e_{12})\ \texttt{or}\ e_2 \curvearrowright_* e'} \tag{5.19d}$$

$$\frac{(e_1\ \texttt{or}\ e_{21})\ \texttt{and}\ (e_1\ \texttt{or}\ e_{22}) \curvearrowright e'}{e_1\ \texttt{or}\ (e_{21}\ \texttt{and}\ e_{22}) \curvearrowright_* e'} \text{ where } e_1 \text{ not } \texttt{and} \tag{5.19e}$$

$$\frac{e_1 \curvearrowright e_1' \quad e_2 \curvearrowright e_2' \quad e_1'\ \texttt{or}\ e_2' \curvearrowright e'}{e_1\ \texttt{or}\ e_2 \curvearrowright_* e'} \text{ where } e_1 \text{ nor } e_2 \text{ are } \texttt{and} \tag{5.19f}$$

$$\frac{e_1 \curvearrowright e_1' \quad e_2 \curvearrowright e_2' \quad e_1'\ \texttt{and}\ e_2' \curvearrowright e'}{e_1\ \texttt{and}\ e_2 \curvearrowright_* e'} \tag{5.19g}$$

There are many fewer rules than the 81 possibly needed. So it is not immediately obvious that all required cases are covered. The following theorem states that they are.

Theorem 5.2 (Completeness)**.** *Let e_1 be an expression such that $\Gamma \vdash e_1 : \texttt{bool}$ for some context Γ. Given e_1, there exists e_2 such that $e_1 \curvearrowright e_2$.*

Proof. The precondition states that the theorem concerns only expression forms that could be Boolean. Recall these forms are: x, `true`, `false`, $\texttt{not}\, e$, $e_{11}\ \texttt{or}\ e_{12}$, and $e_{11}\ \texttt{and}\ e_{12}$. The proof is by case on the form of e_1.

1. e_1 is x, `true`, or `false`. Variables and constants are in CNF. The reflexivity rule of $e_1 \curvearrowright e_2$ applies with e_2 being e_1.

2. e_1 is $\texttt{not}\, e$. The proof is by further induction on the form of e in $\texttt{not}\, e$.

   (a) e is x, `true`, or `false`. Again the overall expression is already in CNF.

   (b) e is $\texttt{not}\, e_{11}$. Then there are two possibilities. Either $\texttt{not not}\, e_{11}$ is already in CNF, handled by the first rule of $\curvearrowright$, or it is not, handled by the second rule of $\curvearrowright$. The second rule recursively calls $(\texttt{not not}\, e_{11}) \curvearrowright_* e_2$. Now the question is whether there exists e_2 such that $(\texttt{not not}\, e_{11}) \curvearrowright_* e_2$. The precondition for the rule handling this form is $e_{11} \curvearrowright e_2$. By inductive hypothesis (IH), there exists an e_2 such that $e_{11} \curvearrowright e_2$. Thus there exists an e_2 such that $(\texttt{not not}\, e_{11}) \curvearrowright_* e_2$. And thus there exists an e_2 such that $(\texttt{not not}\, e_{11}) \curvearrowright e_2$.

   (c) e is $e_{11}\ \texttt{or}\ e_{12}$. The overall expression then is $\texttt{not}\ (e_{11}\ \texttt{or}\ e_{12})$, which cannot be in CNF. The second rule of $\curvearrowright$ is the only one that could apply, and so the recursion calls $\curvearrowright_*$. The second rule of $\curvearrowright_*$ handles this case and the result follows by IH.

   (d) e is $e_{11}\ \texttt{and}\ e_{12}$. The situation is identical to the $e_{11}\ \texttt{or}\ e_{12}$ case.

3. e_1 is e_{11} `or` e_{12}. Possibly this expression is already in CNF, in which case the first rule of $\curvearrowright$ applies. If not, the recursion calls $\curvearrowright_*$. There are three rules in $\curvearrowright_*$ applying to an expression of the form e_{11} `or` e_{12}. The first matches forms where e_{11} is an `and` expression and e_{12} is any form. The second matches forms where e_{11} is any form except an `and` (to exclude cases already included by the previous rule) and e_{12} is an `and` expression. The third rule handles the case where neither e_{11} nor e_{12} are `and` expressions. Thus, collectively these three rules allow e_{11} and e_{12} to be any form, but they have been broken down into more specific cases. The final conclusion that there exists e_2 such that e_{11} `or` $e_{12} \curvearrowright e_2$ follows by IH because each of the three rules' preconditions call $\curvearrowright$.

4. e_1 is e_{11} `and` e_{12}. This could already be in CNF, handled by the reflexivity rule of $\curvearrowright$. If it is not, the second rule of $\curvearrowright$ calls $\curvearrowright_*$. There is one rule defining $\curvearrowright_*$, which handles all `and` expressions regardless of its nested forms, and the conclusion follows by IH because this rules' precondition calls $\curvearrowright$.

□

5.3 Compiling MP to MIP

We now define a compiler for converting general MPs to MIPs. However, as discussed in Appendix A, the convex hull method by which our compiler is motivated requires disjuncts to be bounded. We define a precondition that implies the satisfaction of this requirement and then define the compiler.

Checking that a proposition defines a bounded region would require a domain inference procedure. The condition we check for instead is whether every disjunct can be made bounded. This is easier; it requires that a refined context Υ provide bounds for every variable occurring in every disjunct. Let $\Upsilon \vdash c$ DisjVarsBounded mean Υ contains bounds for all variables free in or existentially introduced within any of the disjuncts in c.

The definition is

$$\frac{}{\Upsilon \vdash \texttt{T}\ \textsc{DisjVarsBounded}} \tag{5.20a}$$

$$\frac{}{\Upsilon \vdash \texttt{F}\ \textsc{DisjVarsBounded}} \tag{5.20b}$$

$$\frac{}{\Upsilon \vdash \texttt{isTrue}\ e\ \textsc{DisjVarsBounded}} \tag{5.20c}$$

$$\frac{}{\Upsilon \vdash e_1\ \texttt{op}\ e_2\ \textsc{DisjVarsBounded}}\ \text{where } \texttt{op} \in \{=, \leq\} \tag{5.20d}$$

$$\frac{\{\rho_j\ \textsc{bounded}\}_{j=1}^{m} \quad \{c_j\ \textsc{ExistVarsBounded}\}_{j=1}^{2}}{\Upsilon \vdash c_1 \vee c_2\ \textsc{DisjVarsBounded}}$$

$$\text{where } FV(c_1 \vee c_2) = \{x_1, \ldots, x_m\}, \text{ and } \Upsilon(x_j) = \rho_j \tag{5.20e}$$

$$\frac{\Upsilon \vdash c_1\ \textsc{DisjVarsBounded} \quad \Upsilon \vdash c_2\ \textsc{DisjVarsBounded}}{\Upsilon \vdash c_1 \wedge c_2\ \textsc{DisjVarsBounded}} \tag{5.20f}$$

$$\frac{\Upsilon, x : \rho \vdash c\ \textsc{DisjVarsBounded}}{\Upsilon \vdash \exists x : \rho\ \texttt{.}\ c\ \textsc{DisjVarsBounded}} \tag{5.20g}$$

Propositions that cannot involve any disjunctions, such as an equation, trivially satisfy this judgement. The conjunctive and existential propositions simply recurse on their sub-proposition. The rule for the disjunctive proposition first checks that all free variables in the disjunction have bounds and then checks that all variables introduced within the disjuncts are bounded.

The latter is done by employing the judgement c ExistVarsBounded, which means all variables existentially introduced within c are bounded. The definition of this judgement is

$$\frac{}{\texttt{T}\ \textsc{ExistVarsBounded}} \tag{5.21a}$$

$$\frac{}{\texttt{F}\ \textsc{ExistVarsBounded}} \tag{5.21b}$$

$$\frac{}{\texttt{isTrue}\ e\ \textsc{ExistVarsBounded}} \tag{5.21c}$$

$$\frac{}{e_1\ \texttt{op}\ e_2\ \textsc{ExistVarsBounded}}\ \text{where } \texttt{op} \in \{=, \leq\} \tag{5.21d}$$

$$\frac{\{c_j\ \textsc{ExistVarsBounded}\}_{j=1}^{2}}{c_1 \vee c_2\ \textsc{ExistVarsBounded}} \tag{5.21e}$$

$$\frac{\{c_j\ \textsc{ExistVarsBounded}\}_{j=1}^{2}}{c_1 \wedge c_2\ \textsc{ExistVarsBounded}} \tag{5.21f}$$

$$\frac{\rho\ \textsc{bounded} \quad c\ \textsc{ExistVarsBounded}}{\exists x : \rho\ \texttt{.}\ c\ \textsc{ExistVarsBounded}} \tag{5.21g}$$

The interesting rule is the one for the existential proposition. This checks that the domain ascribed to the introduced variable is bounded. Other forms simply recurse on their sub-propositions or are trivially satisfied if they do not have sub-propositions.

Let p DisjVarsBounded mean all variables in all disjunctions in program p have known bounds. Its definition is

$$\frac{x_1 : \rho_1, \ldots, x_m : \rho_m \vdash c \text{ DisjVarsBounded}}{\delta_{x_1:\rho_1,\ldots,x_m:\rho_m} \{e \mid c\} \text{ DisjVarsBounded}} \tag{5.22}$$

Our compiler is defined only for programs satisfying this condition. It is a sufficient but not necessary condition. There exist MPs that can be transformed to MIPs but that our compiler is not defined for.

In the next sections, we define a compiler for each syntactic construct in turn: types, expressions, propositions, and finally whole programs.

5.3.1 Type Compiler

All occurrences of types have been replaced with refined types in the syntax. So actually, the type compiler is not needed, but it is important to understand the relationship between the types of the languages. Let $\tau \overset{\text{TYPE}}{\longmapsto} \tau^{\text{MIP}}$ be a type compiler. Its definition is

$$\frac{}{\texttt{real} \overset{\text{TYPE}}{\longmapsto} \texttt{real}} \tag{5.23}$$

$$\frac{}{\texttt{bool} \overset{\text{TYPE}}{\longmapsto} \texttt{real}} \tag{5.24}$$

Let $\rho \overset{\text{RTYPE}}{\longmapsto} \rho^{\text{MIP}}$ be a refined type (domain) compiler. It retains more specific information than the type compiler. The definition of $\overset{\text{RTYPE}}{\longmapsto}$ is inductive on the form of ρ,

$$\frac{}{\langle r_L, r_U\rangle \overset{\text{RTYPE}}{\longmapsto} \langle r_L, r_U\rangle} \tag{5.25a}$$

$$\frac{}{\langle r_L, \infty) \overset{\text{RTYPE}}{\longmapsto} \langle r_L, \infty)} \tag{5.25b}$$

$$\frac{}{(-\infty, r_U\rangle \overset{\text{RTYPE}}{\longmapsto} (-\infty, r_U\rangle} \tag{5.25c}$$

$$\frac{}{\texttt{real} \overset{\text{RTYPE}}{\longmapsto} \texttt{real}} \tag{5.25d}$$

$$\frac{}{[r_L, r_U] \overset{\text{RTYPE}}{\longmapsto} [r_L, r_U]} \tag{5.25e}$$

$$\frac{}{[r_L, \infty) \overset{\text{RTYPE}}{\longmapsto} [r_L, \infty)} \tag{5.25f}$$

$$\frac{}{(-\infty, r_U] \overset{\text{RTYPE}}{\longmapsto} (-\infty, r_U]} \tag{5.25g}$$

$$\frac{}{\texttt{int} \overset{\text{RTYPE}}{\longmapsto} \texttt{int}} \tag{5.25h}$$

$$\frac{}{\{\texttt{true}\} \overset{\text{RTYPE}}{\longmapsto} [1, 1]} \tag{5.25i}$$

$$\frac{}{\{\texttt{false}\} \overset{\text{RTYPE}}{\longmapsto} [0, 0]} \tag{5.25j}$$

$$\frac{}{\texttt{bool} \overset{\text{RTYPE}}{\longmapsto} [0, 1]} \tag{5.25k}$$

The relation is simply reflexive for all numeric domains, since these are already MIP domains.

The `bool` type compiles to $[0, 1]$. The domain of a variable dictates the values it can take. So the domain compiler states a decision about how valuations in the source and target language relate. Consider a source program with the declaration x : `bool`. In the compiled program, this declaration becomes $x : [0, 1]$. In other words, choosing a value of `true` or `false` for x in the source language must correspond to choosing either the value 0

or 1 for x in the compiled program. Whether `true` corresponds to 0 or 1 is not stated by this rule alone.

Singleton domains give more precise information. By defining $\{\texttt{true}\} \overset{\text{RTYPE}}{\longmapsto} [1,1]$, we have decided to make `true` correspond to 1. This decision regards only values, canonical expressions, not general expressions. A non-canonical expression that evaluates to `true` in the source language need not take the value 1 in the compiled program.

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

$$\frac{}{\emptyset \overset{\text{CTXT}}{\longmapsto} \emptyset} \tag{5.26a}$$

$$\frac{\Upsilon \overset{\text{CTXT}}{\longmapsto} \Upsilon^{\text{MIP}} \quad \rho \overset{\text{RTYPE}}{\longmapsto} \rho^{\text{MIP}}}{\Upsilon, x : \rho \overset{\text{CTXT}}{\longmapsto} \Upsilon^{\text{MIP}}, x : \rho^{\text{MIP}}} \tag{5.26b}$$

5.3.2 Expression Compiler

Numeric expressions will be left as is, since these are already valid MIP expressions. Boolean expressions will be converted into CNF prior to compilation, but not all CNF expressions are compiled in the same way. Two compilers are needed. DLF expressions are compiled into integer expressions, and CONJ expressions are compiled directly into a proposition. This allows generating linear integer propositions from Boolean expressions. A single expression compiler could be defined, but that would generate nonlinear expressions.

5.3.2.1 DLF Expression Compiler

Let $e \overset{\text{DLF}}{\longmapsto} e^{\text{MIP}}$ be a judgement converting DLF expressions into 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 ≥ 1, respectively.

After giving the definition, we will prove that it adheres to these decisions. The definition of $e \overset{\text{DLF}}{\longmapsto} e^{\text{MIP}}$ is inductive on the form of e,

$$\frac{}{x \overset{\text{DLF}}{\longmapsto} x} \tag{5.27a}$$

$$\frac{}{\texttt{true} \overset{\text{DLF}}{\longmapsto} 1} \tag{5.27b}$$

$$\frac{}{\texttt{false} \overset{\text{DLF}}{\longmapsto} 0} \tag{5.27c}$$

$$\frac{e \overset{\text{DLF}}{\longmapsto} e'}{\texttt{not}\, e \overset{\text{DLF}}{\longmapsto} 1 - e'} \tag{5.27d}$$

$$\frac{e_1 \overset{\text{DLF}}{\longmapsto} e_1' \quad e_2 \overset{\text{DLF}}{\longmapsto} e_2'}{e_1 \,\texttt{or}\, e_2 \overset{\text{DLF}}{\longmapsto} e_1' + e_2'} \tag{5.27e}$$

The first four rules are for LITERAL forms. 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 following lemmas prove that this compiler adheres to the decisions stated above.

First, we define a valuation compiler, which will be needed in the lemmas. Let $\Psi \overset{\text{VAL}}{\longmapsto} \Psi^{\text{MIP}}$ be defined by

$$\frac{}{\emptyset \overset{\text{VAL}}{\longmapsto} \emptyset} \tag{5.28}$$

$$\frac{\Psi \overset{\text{VAL}}{\longmapsto} \Psi^{\text{MIP}}}{\Psi, x = r \overset{\text{VAL}}{\longmapsto} \Psi^{\text{MIP}}, x = r} \tag{5.29}$$

$$\frac{\Psi \overset{\text{VAL}}{\longmapsto} \Psi^{\text{MIP}}}{\Psi, x = \texttt{true} \overset{\text{VAL}}{\longmapsto} \Psi^{\text{MIP}}, x = 1} \tag{5.30}$$

$$\frac{\Psi \overset{\text{VAL}}{\longmapsto} \Psi^{\text{MIP}}}{\Psi, x = \texttt{false} \overset{\text{VAL}}{\longmapsto} \Psi^{\text{MIP}}, x = 0} \tag{5.31}$$

Numeric assignments are left unchanged, and Boolean assignments are converted as in $\overset{\text{DLF}}{\longmapsto}$.

Lemma 5.3. *Consider e* LITERAL*, $e \overset{\text{DLF}}{\longmapsto} e'$, and $\Psi \overset{\text{VAL}}{\longmapsto} \Psi'$. The following are true:*

1. *If $\Psi \vdash e \searrow \texttt{true}$, then $\Psi' \vdash e' \searrow 1$*
2. *If $\Psi \vdash e \searrow \texttt{false}$, then $\Psi' \vdash e' \searrow 0$.*

Proof. By induction on the form of e.

1. e is x. Then the values of e and e' are given immediately by valuations Ψ and Ψ'. The valuation compiler is defined such that `true` is replaced with 1, and `false` with 0.
2. e is `true` or `false`. The result is immediate from the second and third rules defining $\overset{\text{DLF}}{\longmapsto}$, which compile `true` to 1 and `false` to 0.
3. e is $\texttt{not}\, e_1$. The fourth rule defining $\overset{\text{DLF}}{\longmapsto}$ applies. It converts $\texttt{not}\, e_1$ to $1 - e_1'$ where $e_1 \overset{\text{DLF}}{\longmapsto} e_1'$. Consider first that $\Psi \vdash e_1 \searrow \texttt{true}$. By inductive hypothesis (IH), $\Psi' \vdash e_1' \searrow 1$. If $\Psi \vdash e_1 \searrow \texttt{true}$, then $\Psi \vdash (\texttt{not}\, e_1) \searrow \texttt{false}$. Also, if $\Psi' \vdash e_1' \searrow 1$, then $\Psi' \vdash (1 - e_1') \searrow 0$, giving the desired result. The argument for the case that $\Psi \vdash e_1 \searrow \texttt{false}$ is analogous.

□

Lemma 5.4. *Consider e* DLF*, $e \overset{\text{DLF}}{\longmapsto} e'$, and $\Psi \overset{\text{VAL}}{\longmapsto} \Psi'$. The following are true:*

1. *If $\Psi \vdash e \searrow \texttt{true}$, then $\Psi' \vdash e' \searrow r$, for some $r \geq 1$*
2. *If $\Psi \vdash e \searrow \texttt{false}$, then $\Psi' \vdash e' \searrow 0$.*

Proof. If e LITERAL, then the result follows from Lemma 5.3. If not, only the final rule defining $\overset{\text{DLF}}{\longmapsto}$ applies, which is for expressions of the form $e_1 \,\texttt{or}\, e_2$. The proof proceeds by case on the value of $e_1 \,\texttt{or}\, e_2$.

1. $\Psi \vdash (e_1 \,\texttt{or}\, e_2) \searrow \texttt{true}$. This can only be if either $\Psi \vdash e_1 \searrow \texttt{true}$ or $\Psi \vdash e_2 \searrow \texttt{true}$. Consider that only the first is true. Then, by IH, $\Psi' \vdash e_1' \searrow r_1$, where $r_1 \geq 1$ and

$\Psi' \vdash e_2' \searrow 0$, which leads to the desired result that $\Psi' \vdash (e_1' + e_2') \searrow r$, where $r \geq 1$. If the second were also true, then the value of the sum would only increase. Finally, the situation is analogous if e_2 were true and e_1 either true or false.

2. $\Psi \vdash (e_1 \,\texttt{or}\, e_2) \searrow \texttt{false}$. This can only be if $\Psi \vdash e_1 \searrow \texttt{false}$ and $\Psi \vdash e_2 \searrow \texttt{false}$. By IH, $\Psi' \vdash e_1' \searrow 0$ and $\Psi' \vdash e_2' \searrow 0$. Thus, $\Psi' \vdash (e_1' + e_2') \searrow 0$, as desired.

□

5.3.2.2 CONJ Expression Compiler

Next we define a compiler for CONJ expressions, which are of the form $e_1 \,\texttt{and}\, e_2$. A straightforward idea for converting $e_1 \,\texttt{and}\, e_2$ to an integer expression is to replace `and` with multiplication $*$, just as `or` was mapped to $+$. This naive strategy however leads to nonlinear expressions. Linear propositions can be obtained instead by compiling $e_1 \,\texttt{and}\, e_2$ directly to a proposition, rather than to an expression.

Let $e \overset{\text{CONJ}}{\longmapsto} c^{\text{MIP}}$ be a judgement converting CONJ expressions into MIP propositions. Its definition is

$$\frac{\left\{\emptyset \vdash \texttt{isTrue}\ e_j \overset{\text{PROP}}{\longmapsto} c_j\right\}_{j=1}^{2}}{e_1 \,\texttt{and}\, e_2 \overset{\text{CONJ}}{\longmapsto} c_1 \wedge c_2} \tag{5.32}$$

The preconditions make use of the proposition compiler to be defined in the next section. The central idea here is to transform Boolean conjunction `and` into propositional conjunction $\wedge$. Boolean expressions e_j are treated as propositions $\texttt{isTrue}\ e_j$, these are compiled with the proposition compiler, and the propositional conjunction of the resulting propositions is the final result.

5.3.3 Proposition Compiler

Let $\Upsilon \vdash c \overset{\text{PROP}}{\longmapsto} c^{\text{MIP}}$ be a proposition compiler. Unlike expressions, compiling a proposition requires knowledge of the variables' domains (only because of disjunctive propositions). The precondition $\Upsilon \vdash c$ DisjVarsBounded must be satisfied. The definition of the proposition compiler is inductive on the form of c,

$$\frac{}{\Upsilon \vdash \texttt{T} \overset{\text{PROP}}{\longmapsto} \texttt{T}} \tag{5.33a}$$

$$\frac{}{\Upsilon \vdash \texttt{F} \overset{\text{PROP}}{\longmapsto} \texttt{F}} \tag{5.33b}$$

$$\frac{e \curvearrowright e' \quad e' \text{ DLF} \quad e' \overset{\text{DLF}}{\longmapsto} e''}{\Upsilon \vdash \texttt{isTrue}\ e \overset{\text{PROP}}{\longmapsto} e'' \geq 1} \tag{5.33c}$$

$$\frac{e \curvearrowright e' \quad e' \text{ CONJ} \quad e' \overset{\text{CONJ}}{\longmapsto} c'}{\Upsilon \vdash \texttt{isTrue}\ e \overset{\text{PROP}}{\longmapsto} c'} \tag{5.33d}$$

$$\frac{}{\Upsilon \vdash e_1 = e_2 \overset{\text{PROP}}{\longmapsto} e_1 = e_2} \tag{5.33e}$$

$$\frac{}{\Upsilon \vdash e_1 \leq e_2 \overset{\text{PROP}}{\longmapsto} e_1 \leq e_2} \tag{5.33f}$$

$$\frac{\Upsilon \vdash c_1 \vee c_2 \overset{\text{DISJ}}{\longmapsto} c'}{\Upsilon \vdash c_1 \vee c_2 \overset{\text{PROP}}{\longmapsto} c'} \tag{5.33g}$$

$$\frac{\Upsilon \vdash c_1 \overset{\text{PROP}}{\longmapsto} c_1' \quad \Upsilon \vdash c_2 \overset{\text{PROP}}{\longmapsto} c_2'}{\Upsilon \vdash c_1 \wedge c_2 \overset{\text{PROP}}{\longmapsto} c_1' \wedge c_2'} \tag{5.33h}$$

$$\frac{\rho \overset{\text{RTYPE}}{\longmapsto} \rho' \quad \Upsilon, x : \rho' \vdash c \overset{\text{PROP}}{\longmapsto} c'}{\Upsilon \vdash \exists x : \rho \,.\, c \overset{\text{PROP}}{\longmapsto} \exists x : \rho' \,.\, c'} \tag{5.33i}$$

Boolean propositions are first converted into CNF. A CNF expression will be either DLF

or CONJ. The first rule handles the DLF case. Truth corresponds to a positive integer value. So the compiled proposition 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 compile them. Compilation of a disjunctive proposition is sufficiently complex to justify packaging it into a separate judgement $\overset{\text{DISJ}}{\longmapsto}$, discussed next. Conjunctive propositions simply recurse into their sub-propositions. Similarly for existential propositions, but the introduced variable must be added to the context.

5.3.4 Disjunctive Proposition Compiler

Our disjunctive proposition compiler is motivated by the convex hull method, discussed in Appendix A. 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.

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. Propositions corresponding to known bounds on a variable must be inserted into disjuncts. Equations relating the original and disaggregated variables must be produced. We first define a few auxiliary judgements to assist in the overall definition.

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

$$\frac{\{x_j : \rho_j \backsimeq c_j\}_{j=1}^{m}}{\Upsilon \vdash c \multimap (c_1 \wedge \cdots \wedge c_m \wedge c)} \tag{5.34}$$

where $FV(c) = \{x_1, \ldots, x_m\}$ and $\Upsilon(x_j) = \rho_j$ for $j = 1, \ldots, m$. The $x : \rho \backsimeq c$ relation was defined in Section 4.4. It provides a proposition c corresponding to the bounds declared by ρ.

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,

$$\frac{}{e \circledast x \hookrightarrow x} \tag{5.35a}$$

$$\frac{}{e \circledast r \hookrightarrow e * r} \tag{5.35b}$$

$$\frac{e \circledast e_1 \hookrightarrow e_2}{e \circledast -e_1 \hookrightarrow -e_2} \tag{5.35c}$$

$$\frac{e \circledast e_1 \hookrightarrow e_1' \quad e \circledast e_2 \hookrightarrow e_2'}{e \circledast (e_1 \,\mathtt{op}\, e_2) \hookrightarrow (e_1' \,\mathtt{op}\, e_2')} \text{ for } \mathtt{op} \in \{+, -\} \tag{5.35d}$$

$$\frac{}{e \circledast (e_1 * e_2) \hookrightarrow e * (e_1 * e_2)} \text{ if } FV\,(e_1 * e_2) = \emptyset \tag{5.35e}$$

$$\frac{}{e \circledast (x_1 * e_2) \hookrightarrow (x_1 * e_2)} \tag{5.35f}$$

$$\frac{}{e \circledast (e_1 * x_2) \hookrightarrow (e_1 * x_2)} \tag{5.35g}$$

$$\frac{e \circledast e_2 \hookrightarrow e_2'}{e \circledast (e_1 * e_2) \hookrightarrow (e_1 * e_2')} \text{ if } FV\,(e_1) \neq \emptyset \tag{5.35h}$$

$$\frac{e \circledast e_1 \hookrightarrow e_1'}{e \circledast (e_1 * e_2) \hookrightarrow (e_1' * e_2)} \text{ if } FV\,(e_2) \neq \emptyset \tag{5.35i}$$

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, $e_1 * e_2$ is a non-constant term and is returned unaltered.

Let $e \circledast c_1 \hookrightarrow c_2$ be an analogous judgement for a proposition. Its definition is inductive on c_1,

$$\frac{}{e \circledast \mathtt{T} \hookrightarrow \mathtt{T}} \tag{5.36a}$$

$$\frac{}{e \circledast \mathtt{F} \hookrightarrow \mathtt{F}} \tag{5.36b}$$

$$\frac{e \circledast e_1 \hookrightarrow e_2}{e \circledast (\mathtt{isTrue}\ e_1) \hookrightarrow (\mathtt{isTrue}\ e_2)} \tag{5.36c}$$

$$\frac{e \circledast e_{11} \hookrightarrow e_{21} \quad e \circledast e_{12} \hookrightarrow e_{22}}{e \circledast (e_{11} \,\mathtt{op}\, e_{12}) \hookrightarrow (e_{21} \,\mathtt{op}\, e_{22})} \text{ for } \mathtt{op} \in \{=, \leq\} \tag{5.36d}$$

$$\frac{e \circledast c_{11} \hookrightarrow c_{21} \quad e \circledast c_{12} \hookrightarrow c_{22}}{e \circledast (c_{11} \,\mathtt{op}\, c_{12}) \hookrightarrow (c_{21} \,\mathtt{op}\, c_{22})} \text{ for } \mathtt{op} \in \{\vee, \wedge\} \tag{5.36e}$$

$$\frac{e \circledast c_1 \hookrightarrow c_2}{e \circledast (\exists x : \rho \,\mathtt{.}\, c_1) \hookrightarrow (\exists x : \rho \,\mathtt{.}\, c_1)} \text{ where } x \notin FV\,(e) \tag{5.36f}$$

In the last rule, the existential proposition must be α-converted if $x \in FV\,(e)$. Other rules simply recurse into their nested expressions and propositions.

Finally, let $\Upsilon \vdash c \overset{\text{DISJ}}{\longmapsto} c^{\text{MIP}}$ be a disjunctive proposition compiler. Proposition c must be

a disjunctive proposition $c_A \vee c_B$. Of course $\Upsilon \vdash c$ DisjVarsBounded is a precondition of this judgement. The definition is

$$\frac{\left\{\Upsilon \vdash c_j \overset{\text{PROP}}{\longmapsto} c'_j\right\}_{j\in\{A,B\}} \qquad \Upsilon \overset{\text{CTXT}}{\longmapsto} \Upsilon' \qquad \left\{\Upsilon' \vdash c'_j \multimap c''_j\right\}_{j\in\{A,B\}} \qquad \left\{y^j \circledast \left\{\vec{x}^j/\vec{x}\right\} c''_j \hookrightarrow c'''_j\right\}_{j\in\{A,B\}}}{\Upsilon \vdash (c_A \vee c_B) \overset{\text{DISJ}}{\longmapsto} \left(\begin{array}{l} \exists \vec{x}^A : \vec{\rho} \centerdot \exists \vec{x}^B : \vec{\rho} \centerdot \exists y^A : [0,1] \centerdot \exists y^B : [0,1] \centerdot \\ \qquad \left(\vec{x} = \vec{x}^A + \vec{x}^B\right) \wedge \left(y^A + y^B = 1\right) \wedge \left(c'''_A \wedge c'''_B\right) \end{array}\right)} \tag{5.37}$$

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

The results of these operations are used to produce the final 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 $c'''_A \wedge c'''_B$.

5.3.5 Program Compiler

Expression and proposition compilation is where all the work is. Compiling a program is now straightforward. Let $p \overset{\text{PROG}}{\longmapsto} p^{\text{MIP}}$ be a program compiler. It is defined only for programs p satisfying p MP and p DisjVarsBounded. The definition is

$$\frac{\left\{\rho_j \overset{\text{RTYPE}}{\longmapsto} \rho'_j\right\}_{j=1}^{m} \qquad x_1 : \rho_1, \ldots, x_m : \rho_m \vdash c \overset{\text{PROP}}{\longmapsto} c'}{\delta_{x_1:\rho_1,\ldots,x_m:\rho_m}\{e \mid c\} \overset{\text{PROG}}{\longmapsto} \delta_{x_1:\rho'_1,\ldots,x_m:\rho'_m}\{e \mid c'\}} \tag{5.38}$$

Since the objective e must be of type `real`, it is already in MIP form and need not be compiled.

5.4 Results

The compiler consists principally of two features: conversion of Boolean and disjunctive propositions. These are demonstrated in the following examples.

Example 5.1 Consider the program

```
1 var x:real
2
3 min x subject_to
4
```

```
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 proposition in this example. In the concrete syntax, `e1 implies e2` is provided but this is converted into the expression `not e1 or e2` immediately. First, we verify p_1 MP, which is satisfied. Also, p_1 DISJVARSBOUNDED is trivially satisfied because there are no disjunctive constraints.

Then, the compiler $p_1 \overset{\text{PROG}}{\longmapsto} p_2$ is applied, and the returned program is

```
var x:real

min x subject_to

exists y1:[0,1] . exists y2:[0,1] . exists y3:[0,1] .
  ((1 - y1) + (1 - y2)) + y3 >= 1,
  y1 >= 1
```

Boolean variables have been converted into $[0, 1]$ variables. The conjunctive normal form is derived in the following sequence

▷ `((y1 and y2) implies y3) and y1`

▷ `((not (y1 and y2)) or y3) and y1`

▷ `((not y1 or not y2) or y3) and y1`

This expression is in CNF, and is CONJ. So the compiler $e \overset{\text{CONJ}}{\longmapsto} c^{\text{MIP}}$ gets applied to produce the program shown.

Example 5.2 Now consider a disjunctive proposition,

```
var x:real
var w:real

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

Variable `x` either has to be less than or greater than some function of `w`. The program satisfies p_1 MP, but p_1 DISJVARSBOUNDED fails. The following error messages are printed,

```
ERROR: variable in disjunct must be bounded
  variable at 5.7: w
  is of unbounded type at 2.7-2.10: real

ERROR: variable in disjunct must be bounded
  variable at 5.2: x
  is of unbounded type at 1.7-1.10: 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, p_1 DisjVarsBounded also passes. Thus, we are able to apply the compiler, which produces the pure mixed-integer program[1]

```
1  var x:<10.0, 100.0>
2  var w:<2.0, 50.0>
3
4  min x + w subject_to
5
6  exists y1:[0, 1]
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. In each, bounding constraints have been added.

The benefits of a language with disjunctive propositions is evident. The single disjunctive proposition is far easier to understand and declare than the equivalent MIP propositions. About 20 lines of code in the MIP language are required instead of a single disjunctive proposition.

The disjunction of the previous example involves only inequalities on reals. Thus, the MIP constraints generated by the compiler represent the convex hull of the disjunction. As reviewed in Appendix A, this method was proposed in the early 1970's and has been widely used since. However, only with our type theoretic treatment of MP have we been able to automate this procedure. The automation is a direct consequence of the fact that the set relation $p_1 \overset{\text{PROG}}{\longmapsto} p_2$ is a more precise definition than has been previously provided.

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

Chapter 6

Index Sets

In Chapter 4, we presented a logical formulation of mathematical programs (MPs). One of the motives for this was that it leads directly to a computer language for expressing MPs, in contrast to matrix-based definitions. The resulting language did not however include an essential feature—index sets—which are used in virtually every MP model.

Indexing can be discussed independently of its application to mathematical programming. It can be formulated as a separate logic with its own syntax and semantics. The goal of this chapter is to provide a language in which a rich variety of index sets can be declared in intuitive ways. In the next chapter, the theories of Chapter 4 and this one will be combined to define indexed MPs.

The indexing language is also a novel application of type theory because it has a feature rarely found in other languages—all types are finite. This allows us to define the type system semantically. This contrasts from the syntactically defined type system of MP. There, we first defined the syntax, then the type system, and then the semantics. The MP type system did not require any knowledge about the semantics of the language. The expression $2 + 3$ was defined to be a numeric expression because 2 and 3 are, and that is what $+$ expects. That the expression evaluates to 5 was not relevant.

In the indexing language, a construct will be declared well-formed precisely when it has meaning. The expression $2 + 3$ exists because it means 5. The expression $2 + \texttt{true}$ does not exist because it has no meaning. An open expression $x + 3$ is well-formed if it has meaning for all values of x. In the MP language, this could be an uncountable number of values, but, in the indexing language, x can only take a finite number of values. Thus, we will be able to provide an algorithmic implementation of the semantics.

This chapter also begins with a definition of the language's syntax, but then we immediately define the semantics. Finally, the semantics are employed to define the type system. The overall formulation is in the style of Martin-Löf's (1984) intuitionistic type theory.

6.1 Syntax

The principal construct we are after in the indexing language is an index set, which are the types of this language. Indexing is a dependent type theory because expressions are

used within types. For example, the type $[1, i+1]$ represents the set of all integers from 1 to $i+1$. Furthermore, types are themselves categorized into *kinds*. An expression is of a certain type, and a type is of a certain kind. We first provide the full syntax of the language, followed by basic meta-operations on the syntax.

6.1.1 Full Forms

6.1.1.1 Expressions

Let the expressions of the indexing language be defined by

$$\begin{aligned}\varepsilon ::= & \; x \mid l \mid k \\ & \mid (\varepsilon_1, \ldots, \varepsilon_m) \mid \varepsilon.k \\ & \mid -\varepsilon \mid \varepsilon_1 + \varepsilon_2 \mid \varepsilon_1 - \varepsilon_2 \mid \varepsilon_1 * \varepsilon_2 \\ & \mid \texttt{case}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow \varepsilon_j\}_{j=1}^{m} \mid \texttt{case}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow \varepsilon_j\}_{j=1}^{m}\end{aligned} \tag{6.1}$$

Any variable x is an expression. Next, constants of two forms are provided: the constant l stands for any label, which are denoted with single quotes, e.g. 'A', 'B', and k represents any integer constant.

Tuples $(\varepsilon_1, \ldots, \varepsilon_m)$ can be formed from m expressions. Projection $\varepsilon.k$ gives the k^{th} element of a tuple. For example, $(1, \text{'A'}).2$ is equal to 'A'. The ε in $\varepsilon.k$ need not explicitly be in a tuple form. It could be another expression that evaluates to a tuple. A tuple can contain $m = 0$ or ≥ 2 elements. When, $m = 0$, the tuple is (), a special expression known as unit. In mathematical texts, unit is sometimes denoted as a bold number one **1**.

Next, the standard arithmetic operators are available. These are only valid on integer expressions, but that is a matter of type checking, discussed later.

The case constructs can be thought of as generalized tables. The notation $\{l_j \Rightarrow \varepsilon_j\}_{j=1}^{m}$ is meta-language syntax denoting a list of m constructs of the form $l_j \Rightarrow \varepsilon_j$. This is more compact than writing $\{l_1 \Rightarrow \varepsilon_1, \ldots, l_m \Rightarrow \varepsilon_m\}$. Also, instead of commas, we use a vertical bar | to separate elements of this list. The ε in the case construct is called the *case object*, each l_j is called a *handle*, and each ε_j is called a *branch*. There are two forms of case expressions. They are similar except that one's handles are labels and the other's are integers.

For example, the expression

$$\texttt{case}\ i\ \texttt{of}\ \text{'A'} \Rightarrow 4 \mid \text{'B'} \Rightarrow 3 \mid \text{'C'} \Rightarrow j + 2$$

represents a one-dimensional table. If i evaluates to 'A', the whole case expression evaluates to 4. Multi-dimensional tables are built up by making the branches themselves be case expressions.

6.1.1.2 Types

Expressions are categorized into types, which are the index sets we strive for. Types are defined by the syntax

$$\begin{aligned}\sigma ::= {} & \{l_1, \ldots, l_m\} \mid [\varepsilon_L, \varepsilon_U] \mid x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m \\ & \mid \texttt{case}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow \sigma_j\}_{j=1}^{m} \mid \texttt{case}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow \sigma_j\}_{j=1}^{m} \\ & \mid \lambda x\,.\,\sigma \mid \sigma\,\varepsilon \\ & \mid \sigma :: \kappa \end{aligned} \tag{6.2}$$

Two basic kinds of index sets are provided. A set of the form $\{l_1, \ldots, l_m\}$ is an enumerated type, its elements are explicitly stated labels. Label sets are required to contain at least one element, i.e. $m \geq 1$. The second kind of index set is an integer interval $[\varepsilon_L, \varepsilon_U]$. The type $[1, 10]$ represents the integers from 1 through 10. However, the bounds can be any general expression. The type $[i, j + 1]$ represents all integers from i through $j + 1$. The actual elements of this set depend on the values of i and j. This is what makes the indexing language a dependent type theory. The remaining types are built from these two basic types.

The type $x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m$ is called a dependent product, a generalization of Cartesian products. The x's are optional. Elements of the set $[1, 2] \times [1, 3]$ are pairs. There are $2 * 3 = 6$ elements in this set,

$$[1, 2] \times [1, 3] = \{(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3)\}$$

In many physical systems, the elements of the second set actually depend on what the first element is. The set $x : [1, 2] \times [1, x]$ has only three elements,

$$x : [1, 2] \times [1, x] = \{(1, 1), (2, 1), (2, 2)\}$$

When the first element is 1, the second element can only take values from $[1, 1]$. When the first element is 2, the second set is $[1, 2]$. Examples at the end of this chapter demonstrate how such sets arise naturally in many applications.

It is possible to have a product of zero sets. In this case, the notation $x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m$ does not degenerate into anything visible. Let `unit` denote the dependent product type when $m = 0$. This set contains only the single element $()$.

The case types are similar to case expressions. They represent generalized tables, but the entries of the table are types. This allows conveniently picking different index sets depending on the value of ε. The type

$$\texttt{case}\ \varepsilon\ \texttt{of}\ 1 \Rightarrow \{\text{`A', `B'}\} \mid 2 \Rightarrow [1, 10]$$

is equivalent to {'A', 'B'} if ε is 1, and equivalent to $[1, 10]$ if ε is 2.

Finally, we provide functions. Functions are the most basic abstraction mechanism a

programming language can provide. They allow commonly occurring patterns to be represented in an abstract way, and applying them produces specific instances of that pattern. A function is represented by the notation $\lambda x \,.\, \sigma$. The function argument is x and the body of the function is σ. For example, we could define a function f as

$$\lambda x \,.\, \texttt{case}\ x\ \texttt{of}\ 1 \Rightarrow \{\text{`A'}, \text{`B'}\} \mid 2 \Rightarrow [1, 10] \tag{6.3}$$

This denotes the function that returns {'A', 'B'} when applied to 1, and returns $[1, 10]$ when applied to 2.

The type $\sigma\,\varepsilon$ denotes application of a function σ to an expression ε. In common practice, function arguments are enclosed in parentheses. So we would write $\sigma\,(\varepsilon)$. In fact, the parentheses are superfluous, and they are usually omitted in logical syntax. The type $f\,3$ is of the form $\sigma\,\varepsilon$. The result of the application would be {'A', 'B'}.

Mathematics as practiced in the sciences and engineering does not employ the λ notation, but it is required for functions to be treated properly. Informally, we would write

$$f\,(x) = \texttt{case}\ x\ \texttt{of}\ 1 \Rightarrow \{\text{`A'}, \text{`B'}\} \mid 2 \Rightarrow [1, 10]$$

to define a function called f. This is odd because it is impossible to refer to the function without referring to its name. The exact nature of this statement is clarified by the λ notation. We are giving the name f to the function defined in (6.3).

Functions have only a single argument. Multiple arguments are simulated by passing a single argument that happens to be a tuple. For example, let g be the function $\lambda x \,.\, [x.1,\ x.2 + 1]$. Then, we could write the application $g\,(2, 5)$, which is equal to the set $[2, 6]$. This appears to be a function with two arguments, but it is not. The single argument is the tuple $(2, 5)$.

In the discussion, we have been giving names such as f and g to functions. Actually, the syntax defined thus far does not provide a mechanism for doing this. In the core theory, it is the presence of functions that matters, not their names. Nonetheless, eventually this is necessary; the main benefit of functions is that they can be named and reused. The concrete syntax, discussed in Appendix C, provides a general mechanism for naming any syntactic construct.

Various other notational conveniences are also discussed there. One may prefer to write parentheses around the argument even when it is not a tuple, and this is allowed. Finally, functions can appear to take multiple arguments; instead of $\lambda x \,.\, [x.1,\ x.2 + 1]$, one can write $\lambda\,(x_1, x_2) \,.\, [x_1,\ x_2 + 1]$. These are only features of the concrete syntax, and we will not employ them except in examples at the end of the chapter.

The final type form is "$\sigma :: \kappa$", which is called kind ascription. We explain it after introducing kinds in the next section.

6.1.1.3 Kinds

The type function $\lambda x \,.\, \sigma$ is not a proper type. It does not represent an index set. It has no elements, or rather, the question of what its elements are should never be asked. It however might generate an index set when applied. Some types, such as $\lambda x \,.\, \sigma$, can be

applied. Others, such as $[1, 10]$, cannot be. Determining whether a programmer has not inadvertently applied a non-function requires knowing whether the type being applied is a function or not. Kinds provide this information.

The kinds in the indexing language are

$$\kappa ::= \texttt{IndexSet} \mid x : \sigma \Rightarrow \kappa \tag{6.4}$$

`IndexSet` represents the kind of all index sets. The singular is used for the same reason that the data type `real` in the MP logic is called `real` and not `reals`. We say 3.4 is of type `real` or that 3.4 is a `real`, not 3.4 is an element of the `reals`. Similarly, {'A', 'B'} is an `IndexSet`, which means `IndexSet` contains index sets.

$x : \sigma \Rightarrow \kappa$ denotes a dependent function kind, where the x is optional. It contains function types. For example, the function $\lambda x\,\texttt{.}\,[x,\ x + 3]$ is of kind $[1, 2] \Rightarrow \texttt{IndexSet}$, which is the set of all functions that, when applied to an element of $[1, 2]$, return an index set. The function $\lambda x\,\texttt{.}\,[x,\ x + 3]$ also belongs to other kinds, e.g. $[5, 10] \Rightarrow \texttt{IndexSet}$, as will be defined by the type system in Section 6.3.

This example does not illustrate dependency. In general, a function's codomain might depend on the particular argument it is applied to. The dependent function kind

$$x : [2, 3] \Rightarrow ([1, x] \Rightarrow \texttt{IndexSet})$$

represents the set of all functions which, when applied to 2, return an element in $[1, 2] \Rightarrow$ `IndexSet`, and which, when applied to 3, return an element in $[1, 3] \Rightarrow$ `IndexSet`. What function could possibly behave in this way? An example is

$$\begin{aligned}&\lambda x\,\texttt{.}\,\texttt{case}\ x\ \texttt{of}\\ &\qquad 2 \Rightarrow \lambda x\,\texttt{.}\,\texttt{case}\ x\ \texttt{of}\ 1 \Rightarrow \{\text{'A'}, \text{'B'}\} \mid 2 \Rightarrow \{\text{'C'}, \text{'D'}\}\\ &\qquad \mid 3 \Rightarrow \lambda x\,\texttt{.}\,\texttt{case}\ x\ \texttt{of}\ 1 \Rightarrow [1, 10] \mid 2 \Rightarrow [11, 20] \mid 3 \Rightarrow [21, 30]\end{aligned}$$

Unlike in MPs, the types of the indexing language can be rather complex. Sometimes it might not even be clear that a type σ is of the intended kind. It is useful to allow a kind to be ascribed to a type. This feature is provided in the type form "$\sigma :: \kappa$", which checks that σ is indeed of the kind κ. For example, one might follow a complex σ with the ascription `IndexSet` to verify that σ is indeed an index set.

6.1.1.4 Context

Let Δ be a context providing variables' types,

$$\Delta ::= \emptyset \mid \Delta, x : \sigma \tag{6.5}$$

A context is a possibly empty list of elements of the form $x : \sigma$, which means x is of type σ. It is assumed that variable names in the context are unique. Unlike in the MP context Γ, the order of items matters because a type might involve a variable. The context

$x_1 : [1, 10], x_2 : [1, x_1]$ means the type of x_2 depends on the value that x_1 takes. In contrast, the reverse context $x_2 : [1, x_1], x_1 : [1, 10]$ is ill-formed because x_1 in $[1, x_1]$ has not yet been declared.

6.1.2 Free Variables

6.1.2.1 Free Variables of Expression

Let $FV(\varepsilon)$ denote the free variables of an expression. Its definition is by case on the form of ε,

1. $FV(x) = \{x\}$
2. $FV(l) = \emptyset$
3. $FV(k) = \emptyset$
4. $FV((\varepsilon_1, \ldots, \varepsilon_m)) = \cup_{j=1}^{m} FV(\varepsilon_j)$
5. $FV(\varepsilon.k) = FV(\varepsilon)$
6. $FV(-\varepsilon) = FV(\varepsilon)$
7. $FV(\varepsilon_1 \,\mathtt{op}\, \varepsilon_2) = FV(\varepsilon_1) \cup FV(\varepsilon_2)$, where $\mathtt{op} \in \{+, -, *\}$
8. $FV\left(\mathtt{case}\ \varepsilon\ \mathtt{of}\ \{l_j \Rightarrow \varepsilon_j\}_{j=1}^{m}\right) = FV(\varepsilon) \cup \left(\cup_{j=1}^{m} FV(\varepsilon_j)\right)$
9. $FV\left(\mathtt{case}\ \varepsilon\ \mathtt{of}\ \{k_j \Rightarrow \varepsilon_j\}_{j=1}^{m}\right) = FV(\varepsilon) \cup \left(\cup_{j=1}^{m} FV(\varepsilon_j)\right)$

Let ε CLOSED mean $FV(\varepsilon) = \emptyset$.

6.1.2.2 Free Variables of Type

1. $FV(\{l_1, \ldots, l_m\}) = \emptyset$
2. $FV([\varepsilon_L, \varepsilon_U]) = FV(\varepsilon_L) \cup FV(\varepsilon_U)$
3. (a) $FV(\mathtt{unit}) = \emptyset$
   (b) $FV(x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m) =$
   $FV(\sigma_1) \cup (FV(\sigma_2) \setminus \{x_1\}) \cup \cdots \cup (FV(\sigma_m) \setminus \{x_1, \ldots, x_{m-1}\})$, where $m \geq 2$
4. $FV\left(\mathtt{case}\ \varepsilon\ \mathtt{of}\ \{l_j \Rightarrow \sigma_j\}_{j=1}^{m}\right) = FV(\varepsilon) \cup \left(\cup_{j=1}^{m} FV(\sigma_j)\right)$
5. $FV\left(\mathtt{case}\ \varepsilon\ \mathtt{of}\ \{k_j \Rightarrow \sigma_j\}_{j=1}^{m}\right) = FV(\varepsilon) \cup \left(\cup_{j=1}^{m} FV(\sigma_j)\right)$
6. $FV(\lambda x\,\mathtt{.}\,\sigma) = FV(\sigma) \setminus \{x\}$
7. $FV(\sigma\,\varepsilon) = FV(\sigma) \cup FV(\varepsilon)$
8. $FV(\sigma :: \kappa) = FV(\sigma) \cup FV(\kappa)$

In the dependent product type $x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m$, each x_j is visible only in types further down the list. Consider

$$i : [1, i] \times [1, i + 1]$$

in which there are three i's. The second i is free. Only the third i corresponds to the first.

Let σ CLOSED mean $FV(\sigma) = \emptyset$.

6.1.2.3 Free Variables of Kind

1. $FV(\texttt{IndexSet}) = \emptyset$
2. $FV(x : \sigma \Rightarrow \kappa) = (FV(\kappa) \setminus \{x\}) \cup FV(\sigma)$

Let κ CLOSED mean $FV(\kappa) = \emptyset$.

6.1.3 Substitution

Variables occur in every construct because the indexing language is a dependent type theory. Let a generically refer to a construct ε, σ, or κ. Then $\{\varepsilon/x\}\, a$ is the substitution of ε for x in a. Also let $\{\varepsilon_1/x_1, \ldots, \varepsilon_m/x_m\}\, a$ denote the simultaneous substitution of each ε_j for each x_j into a. The definitions for substitution are provided in the next sections.

6.1.3.1 Substitution Into Expression

The definition of $\{\varepsilon/x\}\, \varepsilon'$ is

1. $\{\varepsilon/x\}\, x' = \begin{cases} x' \text{ if } x \neq x' \\ \varepsilon \text{ if } x = x' \end{cases}$
2. $\{\varepsilon/x\}\, l = l$
3. $\{\varepsilon/x\}\, k = k$
4. $\{\varepsilon/x\}\, (\varepsilon_1, \ldots, \varepsilon_m) = (\{\varepsilon/x\}\, \varepsilon_1, \ldots, \{\varepsilon/x\}\, \varepsilon_m)$
5. $\{\varepsilon/x\}\, (\varepsilon'.k) = (\{\varepsilon/x\}\, \varepsilon')\, .k$
6. $\{\varepsilon/x\}\, (\varepsilon_1\, \texttt{op}\, \varepsilon_2) = \{\varepsilon/x\}\, \varepsilon_1\, \texttt{op}\, \{\varepsilon/x\}\, \varepsilon_2$, where $\texttt{op} \in \{+, -, *\}$
7. $\{\varepsilon/x\} \left(\texttt{case}\, \varepsilon'\, \texttt{of}\, \left\{l_j \Rightarrow \varepsilon''_j\right\}_{j=1}^{m}\right) = \texttt{case}\, \{\varepsilon/x\}\, \varepsilon'\, \texttt{of}\, \left\{l_j \Rightarrow \{\varepsilon/x\}\, \varepsilon''_j\right\}_{j=1}^{m}$
8. $\{\varepsilon/x\} \left(\texttt{case}\, \varepsilon'\, \texttt{of}\, \left\{l_j \Rightarrow \varepsilon''_j\right\}_{j=1}^{m}\right) = \texttt{case}\, \{\varepsilon/x\}\, \varepsilon'\, \texttt{of}\, \left\{l_j \Rightarrow \{\varepsilon/x\}\, \varepsilon''_j\right\}_{j=1}^{m}$

6.1.3.2 Substitution Into Type

Substitution into types $\{\varepsilon/x\}\, \sigma$ is given by the rules

1. $\{\varepsilon/x\}\, \{l_1, \ldots, l_m\} = \{l_1, \ldots, l_m\}$
2. $\{\varepsilon/x\}\, [\varepsilon_L, \varepsilon_U] = [\{\varepsilon/x\}\, \varepsilon_L, \{\varepsilon/x\}\, \varepsilon_U]$

3. (a) $\{\varepsilon/x\}\,(x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m) = x_1 : \{\varepsilon/x\}\,\sigma_1 \times \{\varepsilon/x\}\,(x_2 : \sigma_2 \times \cdots \times x_m : \sigma_m)$ if
   $x \neq x_1$
   (b) $\{\varepsilon/x\}\,(x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m) = x_1 : \{\varepsilon/x\}\,\sigma_1 \times x_2 : \sigma_2 \times \cdots \times x_m : \sigma_m$ if
   $x = x_1$
4. $\{\varepsilon/x\}\left(\texttt{case}\ \varepsilon'\ \texttt{of}\ \{l_j \Rightarrow \sigma_j\}_{j=1}^{m}\right) = \texttt{case}\ \{\varepsilon/x\}\,\varepsilon'\ \texttt{of}\ \{l_j \Rightarrow \{\varepsilon/x\}\,\sigma_j\}_{j=1}^{m}$
5. $\{\varepsilon/x\}\left(\texttt{case}\ \varepsilon'\ \texttt{of}\ \{k_j \Rightarrow \sigma_j\}_{j=1}^{m}\right) = \texttt{case}\ \{\varepsilon/x\}\,\varepsilon'\ \texttt{of}\ \{k_j \Rightarrow \{\varepsilon/x\}\,\sigma_j\}_{j=1}^{m}$
6. (a) $\{\varepsilon/x\}\,(\lambda x' \texttt{.}\, \sigma) = \lambda x' \texttt{.}\, \{\varepsilon/x\}\,\sigma$ if
   $x \neq x'$
   (b) $\{\varepsilon/x\}\,(\lambda x' \texttt{.}\, \sigma) = \lambda x' \texttt{.}\, \sigma$ if
   $x = x'$
7. $\{\varepsilon/x\}\,(\sigma\ \varepsilon') = \{\varepsilon/x\}\,\sigma\ \{\varepsilon/x\}\,\varepsilon'$
8. $\{\varepsilon/x\}\,(\sigma :: \kappa) = \{\varepsilon/x\}\,\sigma : \{\varepsilon/x\}\,\kappa$

Rule 3.a for the dependent product type abuses notation. As written, it converts an m-ary product into a 2-ary product, but that is not the intention. Substitution into the 2^{nd} through m^{th} types should be performed as written. But then, the result should be unwrapped to reconstruct the original m-ary product.

6.1.3.3 Substitution Into Kind

Finally, the definition of substitution into kinds $\{\varepsilon/x\}\,\kappa$ is

1. $\{\varepsilon/x\}\,\texttt{IndexSet} = \texttt{IndexSet}$
2. (a) $\{\varepsilon/x\}\,(x' : \sigma \Rightarrow \kappa) = (x' : \{\varepsilon/x\}\,\sigma \Rightarrow \{\varepsilon/x\}\,\kappa)$ if
   $x \neq x'$
   (b) $\{\varepsilon/x\}\,(x' : \sigma \Rightarrow \kappa) = (x' : \{\varepsilon/x\}\,\sigma \Rightarrow \kappa)$ if
   $x = x'$

6.1.3.4 Substitution Into Context

Contexts contain types, which can involve variables. Thus, it is possible to substitute for variables in a context. Let $\{\varepsilon/x\}\,\Delta$ denote this operation. It is defined by case on the form of a context,

1. $\{\varepsilon/x\}\,\emptyset = \emptyset$
2. $\{\varepsilon/x\}\,(\Delta, x' : \sigma) = \{\varepsilon/x\}\,\Delta, (x' : \{\varepsilon/x\}\,\sigma)$

There is no problem in substituting for x into a context that contains x. As an example,

$$\{y + 1/x\}\,(y : [1, 3]\,, x : [1, 10]\,, z : [1, x])$$

results in the context

$$y : [1, 3]\,, x : [1, 10]\,, z : [1, y + 1]\,.$$

6.1.4 Canonical Forms

Canonical forms are a subset of the syntax. It refers to those forms which are considered irreducible in their outermost form.

6.1.4.1 Canonical Expressions

Let ε CANONICAL mean expression ε is canonical. Its definition is

$$\frac{}{l \text{ CANONICAL}} \tag{6.6a}$$

$$\frac{}{k \text{ CANONICAL}} \tag{6.6b}$$

$$\frac{(\varepsilon_1, \ldots, \varepsilon_m) \text{ CLOSED}}{(\varepsilon_1, \ldots, \varepsilon_m) \text{ CANONICAL}} \tag{6.6c}$$

Canonical expressions are also called values or constants. Values are important enough to justify a special notation. Let η refer to an expression ε such that ε CANONICAL. By our definition the tuple $(2 + 1, 3 + 5)$ is canonical. Its sub-expressions can be reduced, but its outermost form will remain a tuple. Canonicity of tuples could require that each of its elements are canonical, but that does affect any results.

6.1.4.2 Canonical Types

Let σ CANONICAL mean type σ is canonical. The rules defining this judgement are

$$\frac{}{\{l_1, \ldots, l_m\} \text{ CANONICAL}} \tag{6.7a}$$

$$\frac{[\varepsilon_L, \varepsilon_U] \text{ CLOSED}}{[\varepsilon_L, \varepsilon_U] \text{ CANONICAL}} \tag{6.7b}$$

$$\frac{x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m \text{ CLOSED}}{x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m \text{ CANONICAL}} \tag{6.7c}$$

$$\frac{\lambda x \,.\, \sigma \text{ CLOSED}}{\lambda x \,.\, \sigma \text{ CANONICAL}} \tag{6.7d}$$

All label sets are canonical. Integer intervals, products, and type functions are canonical if they have no free variables. The case types and application forms are not canonical. They could be reduced to one of the above forms, given values for their free variables.

We mentioned that canonicity of tuples could require the sub-expressions to be canonical. Similarly, for the interval type. However, there is no choice with product types and functions.

The function $\lambda x\,.\,[1,\ x+1]$ contains the non-canonical type $[1, x+1]$ within it. There is no way to reduce the function further because x is of unknown value. Judgements on canonical forms are thus mutually inductive with those on non-canonical forms.

6.1.4.3 Canonical Kinds

The definition of κ CANONICAL is

$$\frac{}{\texttt{IndexSet}\ \text{CANONICAL}} \tag{6.8a}$$

$$\frac{x : \sigma \Rightarrow \kappa\ \text{CLOSED}}{x : \sigma \Rightarrow \kappa\ \text{CANONICAL}} \tag{6.8b}$$

Closed and canonical kinds happen to be the same.

6.2 Semantics

The semantics are defined as in the MP language. We define the relationship between closed and canonical forms, and the meaning of open forms is with respect to a valuation. Discussions of truth do not enter here because there are no propositions.

6.2.1 Expression Evaluation

Given ε_1 CLOSED and ε_2 CANONICAL, $\varepsilon_1 \searrow \varepsilon_2$ means ε_1 evaluates to ε_2. Its definition is inductive on the form of ε_1,

$$\frac{}{l \searrow l} \tag{6.9a}$$

$$\frac{}{k \searrow k} \tag{6.9b}$$

$$\frac{}{(\varepsilon_1, \ldots, \varepsilon_m) \searrow (\varepsilon_1, \ldots, \varepsilon_m)} \tag{6.9c}$$

$$\frac{\varepsilon \searrow (\varepsilon_1, \ldots, \varepsilon_m) \quad \varepsilon_k \searrow \varepsilon'}{\varepsilon.k \searrow \varepsilon'} \text{ where } k \in \{1, \ldots, m\} \tag{6.9d}$$

$$\frac{\varepsilon \searrow k}{-\varepsilon \searrow -k} \tag{6.9e}$$

$$\frac{\varepsilon_1 \searrow k_1 \quad \varepsilon_2 \searrow k_2}{\varepsilon_1\ \texttt{op}\ \varepsilon_2 \searrow k} \text{ where } k_1\,\text{op}\,k_2 = k, \text{ for } \texttt{op} \in \{+, -, *\} \tag{6.9f}$$

$$\frac{\varepsilon \searrow l_k \quad \varepsilon_k \searrow \varepsilon'}{\texttt{case}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow \varepsilon_j\}_{j=1}^{m} \searrow \varepsilon'} \text{ where } l_k \in \{l_1, \ldots, l_m\} \tag{6.9g}$$

$$\frac{\varepsilon \searrow k \quad \varepsilon_k \searrow \varepsilon'}{\texttt{case}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow \varepsilon_j\}_{j=1}^{m} \searrow \varepsilon'} \text{ where } k \in \{k_1, \ldots, k_m\} \tag{6.9h}$$

Labels, integers, and tuples are already canonical. Evaluation of a projection $\varepsilon.k$ requires ε to evaluate to a tuple $(\varepsilon_1, \ldots, \varepsilon_m)$ because a tuple is the only canonical form that can be projected. The k^{th} element ε_k must itself be evaluated to obtain a canonical form. The

arguments to arithmetic operators must evaluate to integers. The final result is obtained by employing integer addition in the meta-language.

In the case constructs, the case object ε must evaluate to one of the handles. Consider

$$\texttt{case}\ \varepsilon\ \texttt{of}\ \text{`A'} \Rightarrow 4 \mid \text{`B'} \Rightarrow 3 \mid \text{`C'} \Rightarrow 2$$

The case object is ε and the handles are 'A', 'B', and 'C'. If ε evaluates to say 'D' or an integer k, then there is no way to evaluate the expression. If it evaluates to one of the handles, then the entire expression is equal to the branch for that handle.

6.2.2 Type Evaluation

Given σ_1 CLOSED and σ_2 CANONICAL, $\sigma_1 \searrow \sigma_2$ means σ_1 evaluates to σ_2. Its definition is

$$\frac{}{\{l_1, \ldots, l_m\} \searrow \{l_1, \ldots, l_m\}} \tag{6.10a}$$

$$\frac{}{[\varepsilon_L, \varepsilon_U] \searrow [\varepsilon_L, \varepsilon_U]} \tag{6.10b}$$

$$\frac{}{x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m \searrow x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m} \tag{6.10c}$$

$$\frac{\varepsilon \searrow l_k \quad \sigma_k \searrow \sigma}{\texttt{case}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow \sigma_j\}_{j=1}^{m} \searrow \sigma} \text{ where } l_k \in \{l_1, \ldots, l_m\} \tag{6.10d}$$

$$\frac{\varepsilon \searrow k \quad \sigma_k \searrow \sigma}{\texttt{case}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow \sigma_j\}_{j=1}^{m} \searrow \sigma} \text{ where } k \in \{k_1, \ldots, k_m\} \tag{6.10e}$$

$$\frac{}{\lambda x \,.\, \sigma \searrow \lambda x \,.\, \sigma} \tag{6.10f}$$

$$\frac{\sigma \searrow \lambda x \,.\, \sigma' \quad \{\varepsilon/x\}\,\sigma' \searrow \sigma''}{\sigma\,\varepsilon \searrow \sigma''} \tag{6.10g}$$

$$\frac{\sigma \searrow \sigma' \quad \kappa \searrow \kappa' \quad \vdash^c \sigma' :: \kappa'}{\sigma :: \kappa \searrow \sigma'} \tag{6.10h}$$

Label sets, integer intervals, and products are already canonical. Evaluation rules for case types are motivated by the same reasons given for the analogous expression constructs.

Closed functions are already canonical. Function application $\sigma\,\varepsilon$ requires σ to evaluate to a function $\lambda x \,.\, \sigma'$. Analogously to projection in expressions, this is correct because no other canonical form can be applied. Performing the application is known as β-reduction. It requires substituting the argument into the function body, viz $\{\varepsilon/x\}\,\sigma'$. The result must itself be evaluated because it is only guaranteed to be closed, not canonical.

Evaluating a type that has been ascribed a kind requires that the ascription is valid. The ascription is disregarded once this is checked.

6.2.3 Kind Evaluation

All closed kinds are canonical. Thus, evaluation of a kind $\kappa_1 \searrow \kappa_2$ is simply an identity relation,

$$\frac{}{\kappa \searrow \kappa} \tag{6.11}$$

6.2.4 Meaning of Open Forms

Let

$$\Phi ::= \emptyset \mid \Phi, i = \eta \tag{6.12}$$

be a valuation for index variables. The judgement $\Phi \vdash \varepsilon_1 \searrow \varepsilon_2$ means the possibly open expression e_1 evaluates to the canonical expression e_2, under the valuation Φ. Its definition is

1. $\emptyset \vdash \varepsilon \searrow \eta$ if

   $\varepsilon \searrow \eta$

2. $\Phi, x = \eta' \vdash \varepsilon \searrow \eta$ if

   $\Phi \vdash \{\eta'/x\}\,\varepsilon \searrow \eta$

The corresponding judgements for types $\Phi \vdash \sigma_1 \searrow \sigma_2$ and kinds $\Phi \vdash \kappa_1 \searrow \kappa_2$ can be defined analogously.

6.3 Type System

Three syntactic forms have been defined: full, closed, and canonical forms. In this section, typing judgements, and some other judgements needed later, are defined for these forms in turn. The majority of the work is in the judgements on canonical forms. The types of non-canonical forms will be defined by interpreting their canonical form.

The definitions are mutually coupled, making it impossible to order the discussion such that prerequisites have been covered. A notational convention will help to conceptually understand the meaning of a judgement prior to its definition. Let $\vdash^c J$ be a judgement on canonical forms, where J involves one or more syntactic constructs. The corresponding judgement on closed forms will be notated $\vdash^q J$ and on full forms $\Delta \vdash J$. The context is needed only for judgements on full forms because there are no free variables in closed or canonical forms. Figure 6.1 shows how these three categories of judgements depend on each other.

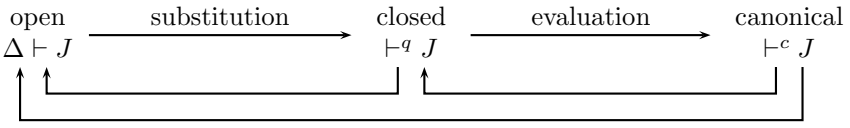


Figure 6.1: Judgement dependencies in a semantic type theory. $A \longrightarrow B$ means definition of judgement A may depend on judgement B. Open forms are closed by substituting the possible values of variables. Closed forms are canonized by evaluating them.

6.3.1 Judgements on Canonical Forms

6.3.1.1 Well-Formed Canonical Kind

$\vdash^c \kappa$ KIND means κ is a canonical kind. Its definition is inductive on the form of κ,

$$\frac{}{\vdash^c \texttt{IndexSet} \text{ KIND}} \tag{6.13a}$$

$$\frac{\vdash^q \sigma :: \texttt{IndexSet} \quad x : \sigma \vdash \kappa \text{ KIND}}{\vdash^c x : \sigma \Rightarrow \kappa \text{ KIND}} \tag{6.13b}$$

The function kind $x : \sigma \Rightarrow \kappa$ is well-formed only if σ is an index set. Elements of this kind are type functions which take as input an expression and return a type. If an expression is the input argument, the input space must be an index set because expressions can only be of a type which is of kind `IndexSet`. The x can occur free in κ. The judgement on full forms must be called with $x : \sigma$ as the context.

6.3.1.2 Canonical Kind of Canonical Type

$\vdash^c \sigma :: \kappa$ means σ is a canonical type of canonical kind κ, where $\vdash^c \kappa$ KIND is assumed. Its definition is inductive on the form of σ,

$$\frac{}{\vdash^c \{l_1, \ldots, l_m\} :: \texttt{IndexSet}} \tag{6.14a}$$

$$\frac{\vdash^q \varepsilon_L \leq \varepsilon_U}{\vdash^c [\varepsilon_L, \varepsilon_U] :: \texttt{IndexSet}} \tag{6.14b}$$

$$\frac{\begin{array}{r}\emptyset \vdash \sigma_1 :: \texttt{IndexSet} \\ x_1 : \sigma_1 \vdash \sigma_2 :: \texttt{IndexSet} \\ \vdots \\ x_1 : \sigma_1, \ldots, x_{m-1} : \sigma_{m-1} \vdash \sigma_m :: \texttt{IndexSet}\end{array}}{\vdash^c x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m :: \texttt{IndexSet}} \tag{6.14c}$$

$$\frac{x : \sigma \vdash \sigma' :: \kappa'}{\vdash^c (\lambda x \,.\, \sigma') :: (x : \sigma \Rightarrow \kappa')} \tag{6.14d}$$

All label sets are index sets. An integer interval is an index set only if its lower bound is less than or equal to its upper bound. Otherwise, it is ill-formed. The dependent product type is an index set only if each of its components is. Type σ_m must be checked in a context containing all variables introduced in previous components. Finally, the type function $\lambda x . \sigma'$ is of kind $x : \sigma \Rightarrow \kappa'$ if for all values that x can take in σ, we can check that σ' is of kind κ'.

6.3.1.3 Canonical Type of Canonical Expression

$\vdash^c \varepsilon : \sigma$ means ε is a canonical element of canonical type σ, where $\vdash^c \sigma :: \texttt{IndexSet}$ is assumed. Its definition is inductive on the form of ε,

$$\frac{}{\vdash^c l : \{l_1, \ldots, l_m\}} \text{ if } l \in \{l_1, \ldots, l_m\} \tag{6.15a}$$

$$\frac{\vdash^q \varepsilon_L \leq k \vdash \quad \vdash^q k \leq \varepsilon_U}{\vdash^c k : [\varepsilon_L, \varepsilon_U]} \tag{6.15b}$$

$$\frac{\vdash^q \varepsilon_1 : \sigma_1 \quad \vdash^q \varepsilon_2 : \{\varepsilon_1/x_1\}\,\sigma_2 \quad \cdots \quad \vdash^q \varepsilon_m : \{\varepsilon_1/x_1, \ldots, \varepsilon_{m-1}/x_{m-1}\}\,\sigma_m}{\vdash^c (\varepsilon_1, \ldots, \varepsilon_m) : (x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m)} \tag{6.15c}$$

A label l is of type $\{l_1, \ldots, l_m\}$ if the label is in the set. Multiple, indeed an infinity of, sets satisfy this requirement. Thus, a label can be ascribed an infinity of types. In contrast, in the MP logic, each expression had at most one type. Similarly, an integer k belongs to all intervals whose lower bound is less than or equal to k and whose upper bound is greater than or equal to k. A tuple $(\varepsilon_1, \ldots, \varepsilon_m)$ belongs to a product type $x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m$. The second element ε_2 must belong to $\{\varepsilon_1/x_1\}\,\sigma_2$. In other words, the first element affects what the second element can be.

6.3.1.4 Canonical Subtyping

Let $\vdash^c \sigma_1 \leq: \sigma_2$ mean canonical type σ_1 is a subtype of canonical type σ_2, where $\vdash^c \sigma_1 :: \texttt{IndexSet}$ and $\vdash^c \sigma_2 :: \texttt{IndexSet}$ are assumed. If σ_1 is a subtype of σ_2, any expression of type σ_1 is also of type σ_2. The definition of subtyping is

$$\frac{}{\vdash^c \{l_1, \ldots, l_m\} \leq: \{l'_1, \ldots, l'_n\}} \text{ if } \{l_1, \ldots, l_m\} \subseteq \{l'_1, \ldots, l'_n\} \tag{6.16a}$$

$$\frac{\vdash^q \varepsilon'_L \leq \varepsilon_L \quad \vdash^q \varepsilon_U \leq \varepsilon'_U}{\vdash^c [\varepsilon_L, \varepsilon_U] \leq: [\varepsilon'_L, \varepsilon'_U]} \tag{6.16b}$$

$$\frac{\begin{array}{r}\emptyset \vdash \sigma_1 \leq: \sigma'_1 \\ x_1 : \sigma_1 \vdash \sigma_2 \leq: \sigma'_2 \\ \vdots \\ x_1 : \sigma_1, \ldots, x_{m-1} : \sigma_{m-1} \vdash \sigma_m \leq: \sigma'_m\end{array}}{\vdash^c (x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m) \leq: (x_1 : \sigma'_1 \times \cdots \times x_m : \sigma'_m)} \tag{6.16c}$$

Subtyping of label sets relies on the meta-language notion of subsets. An interval is a sub-interval of another if its lower bound is greater and upper bound less than the other's.

Product types require each of their corresponding components to be subtypes. Firstly, σ_1 must be a subtype of σ'_1. The second precondition requires σ_2 to be a subtype of σ'_2. The question is in what context this must be true. Should it be $x_1 : \sigma_1$ or $x_1 : \sigma'_1$? Both σ_2 and σ'_2 potentially involve x_1. In σ_2, x_1 can take any value in σ_1, and in σ'_2, x_1 can take any value in σ'_1. The judgement needing to be checked involves both σ_2 and σ'_2. Thus, the more restrictive type for x_1 must be chosen, which is σ_1. Imagine if instead the precondition was stated as $x_1 : \sigma'_1 \vdash \sigma_2 \leq: \sigma'_2$. Well, now for some values of σ'_1, σ_2 is not even well-formed.

So the correct choice must be $x_1 : \sigma_1$. However, this then fails to consider some valid values of x_1 in σ_2'. Considering these additional values could only increase the elements in σ_2', so the check would not become invalidated.

The subtyping relation tells us which expressions can be substituted for others, without causing a well-formed program to become ill-formed. For example, let $\sigma_1 = [2, 3]$ and $\sigma_2 = [1, 4]$, and consider two expressions $\varepsilon_1 : \sigma_1$ and $\varepsilon_2 : \sigma_2$. Since σ_1 is a subtype of σ_2, it is okay to replace ε_2 with ε_1. For example,

$$\texttt{case}\ \varepsilon_2\ \texttt{of}\ 1 \Rightarrow \text{`A'} \mid 2 \Rightarrow \text{`B'} \mid 3 \Rightarrow \text{`C'} \mid 4 \Rightarrow \text{`D'}$$

is a well-formed expression; expression ε_2 is of type $[1, 4]$ and there is branch for each value in $[1, 4]$. Imagine replacing ε_2 with ε_1. Expression ε_1 can only take fewer values because its type is a subtype of ε_2's. The expression remains well-formed because there is surely still a branch for the fewer possible values of the case object. Said another way, if ε_1 is of type σ_1, and σ_1 is a subtype of σ_2, then ε_1 is also of type σ_2.

6.3.1.5 Canonical Type Equivalence

Let $\vdash^c \sigma_1 \equiv \sigma_2 :: \kappa$ mean canonical type σ_1 is equivalent to canonical type σ_2, where $\vdash^c \kappa$ KIND and $\vdash^c \sigma_1 :: \kappa$ and $\vdash^c \sigma_2 :: \kappa$ are assumed. This means any expression of either type is also of the other type. The definition is

$$\frac{}{\vdash^c \{l_1, \ldots, l_m\} \equiv \{l_1', \ldots, l_n'\} :: \texttt{IndexSet}}\ \text{if}\ \{l_1, \ldots, l_m\} = \{l_1', \ldots, l_n'\} \tag{6.17a}$$

$$\frac{\vdash^q \varepsilon_L \equiv \varepsilon_L' : [\varepsilon_L, \varepsilon_L] \quad \vdash^q \varepsilon_U \equiv \varepsilon_U' : [\varepsilon_U, \varepsilon_U]}{\vdash^c [\varepsilon_L, \varepsilon_U] \equiv [\varepsilon_L', \varepsilon_U'] :: \texttt{IndexSet}} \tag{6.17b}$$

$$\frac{\begin{array}{r}\emptyset \vdash \sigma_1 \equiv \sigma_1' :: \texttt{IndexSet} \\ x_1 : \sigma_1 \vdash \sigma_2 \equiv \sigma_2' :: \texttt{IndexSet} \\ \vdots \\ x_1 : \sigma_1, \ldots, x_{m-1} : \sigma_{m-1} \vdash \sigma_m \equiv \sigma_m' :: \texttt{IndexSet}\end{array}}{\vdash^c x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m \equiv x_1 : \sigma_1' \times \cdots \times x_m : \sigma_m' :: \texttt{IndexSet}} \tag{6.17c}$$

$$\frac{x : \sigma \vdash \sigma_1 \equiv \sigma_2 :: \kappa}{\vdash^c \lambda x \,.\, \sigma_1 \equiv \lambda x \,.\, \sigma_2 :: (x : \sigma \Rightarrow \kappa)} \tag{6.17d}$$

The first three rules are similar to the subtyping judgement, but equality is also defined for type functions. Two functions are equal if for every $x : \sigma$, their input type, the output will be equivalent.

6.3.1.6 Canonical Subkinding

Let $\vdash^c \kappa_1 \leq:: \kappa_2$ mean κ_1 is a subkind of κ_2, where $\vdash^c \kappa_1$ KIND and $\vdash^c \kappa_2$ KIND are assumed. This means any type that is an element of κ_1 is also an element of κ_2. The definition of

subkinding is

$$\frac{}{\vdash^c \texttt{IndexSet} \leq:: \texttt{IndexSet}} \tag{6.18a}$$

$$\frac{\vdash^q \sigma_2 \leq: \sigma_1 \quad x : \sigma_2 \vdash \kappa_1 \leq:: \kappa_2}{\vdash^c (x : \sigma_1 \Rightarrow \kappa_1) \leq:: (x : \sigma_2 \Rightarrow \kappa_2)} \tag{6.18b}$$

A function kind $x : \sigma_1 \Rightarrow \kappa_1$ can be a subkind of another function kind $x : \sigma_2 \Rightarrow \kappa_2$ if its domain is larger and codomain smaller. Let g_1 be a function of kind $x : \sigma_1 \Rightarrow \kappa_1$ and g_2 a function of kind $x : \sigma_2 \Rightarrow \kappa_2$. Since $x : \sigma_1 \Rightarrow \kappa_1$ is a subkind of $x : \sigma_2 \Rightarrow \kappa_2$, it should be possible to replace any occurrence of g_2 with g_1. Thus, it should be possible to apply g_1 to at least any argument that g_2 might be applied to, i.e. the domain of g_1 should contain at least all the elements of g_2's domain. Conversely, $g_2\,(\varepsilon)$ returns some value which is valid in the location that $g_2\,(\varepsilon)$ occurs. The application $g_1\,(\varepsilon)$ must not return any value that $g_2\,(\varepsilon)$ would not, i.e. the codomain of g_1 should be smaller than that of g_2. Else, replacing $g_2\,(\varepsilon)$ with $g_1\,(\varepsilon)$ would lead to a possibly ill-formed program.

6.3.1.7 Canonical Kind Equivalence

Let $\vdash^c \kappa_1 \equiv \kappa_2$ mean the two canonical kinds are equivalent, where $\vdash^c \kappa_1$ KIND and $\vdash^c \kappa_2$ KIND are assumed. The definition is analogous to subkinding,

$$\frac{}{\vdash^c \texttt{IndexSet} \equiv \texttt{IndexSet}} \tag{6.19a}$$

$$\frac{\vdash^q \sigma_2 \equiv \sigma_1 :: \texttt{IndexSet} \quad x : \sigma_2 \vdash \kappa_1 \equiv \kappa_2}{\vdash^c (x : \sigma_1 \Rightarrow \kappa_1) \equiv (x : \sigma_2 \Rightarrow \kappa_2)} \tag{6.19b}$$

6.3.1.8 Canonical Expression Comparison

$\vdash^c \varepsilon_1 \leq \varepsilon_2$ means ε_1 is less than or equal to ε_2. The notion of comparison is defined only for integers. It is assumed that both expressions are of an integer interval type. The definition is by the single rule

$$\frac{}{\vdash^c k_1 \leq k_2} \text{ if } k_1 \leq k_2 \tag{6.20}$$

which resorts to the meta-language notion of comparison.

6.3.1.9 Canonical Expression Equivalence

$\vdash^c \varepsilon_1 \equiv \varepsilon_2 : \sigma$ means the two canonical expressions are equivalent, where $\vdash^c \varepsilon_1 : \sigma$ and $\vdash^c \varepsilon_2 : \sigma$ are assumed. Unlike comparison, the equivalence notion applies to all types of

expressions. The definition is given by the rules

$$\frac{}{\vdash^c l_1 \equiv l_2 : \{l_1, \ldots, l_m\}} \text{ where } l_1 = l_2 \tag{6.21a}$$

$$\frac{}{\vdash^c k_1 \equiv k_2 : [\varepsilon_L, \varepsilon_U]} \text{ where } k_1 = k_2 \tag{6.21b}$$

$$\frac{\begin{array}{c}\vdash^q e_1 \equiv e_1' : \sigma_1 \\ \vdash^q e_2 \equiv e_2' : \{e_1/x_1\}\,\sigma_2 \\ \vdots \\ \vdash^q e_m \equiv e_m' : \{e_1/x_1, \ldots, e_{m-1}/x_{m-1}\}\,\sigma_m\end{array}}{\vdash^c (e_1, \ldots, e_m) \equiv (e_1', \ldots, e_m') : (x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m)} \tag{6.21c}$$

Two tuples are equivalent if each of their components are. The only slight challenge is to state at what type these components are supposed to be equivalent. Elements ε_j and ε_j' should be consider at type σ_j with all the x's from previous components substituted in to σ_j. This substitution is guaranteed to close the types but they may not be canonical. Hence, the recursion calls the equivalence relation on closed forms.

Functions are equivalent if their returned values are equivalent when applied to equivalent values. This can be checked by requiring the function bodies to be equivalent in the context $x : \sigma_1$. The judgement on full forms will check equivalence under each possible value of x in σ_1.

6.3.2 Judgements on Closed Forms

Judgements on closed forms simply evaluate constructs and then employ the corresponding judgement on canonical forms. The general form of a judgement on canonical forms is $\vdash^q J$, where J involves any of ε, σ, and/or κ. The general definition is

$$\frac{J \searrow J' \quad \vdash^c J'}{\vdash^q J} \tag{6.22}$$

To check a property of closed forms, we evaluate all constructs to their canonical form and then call the corresponding judgement on canonical forms. As one example, kind checking is defined by the rule

$$\frac{\sigma \searrow \sigma' \quad \kappa \searrow \kappa' \quad \vdash^c \sigma' :: \kappa'}{\vdash^q \sigma :: \kappa} \tag{6.23}$$

The important point is that this definition has an implementation because evaluation in the indexing language is algorithmic. Evaluation relations in the MP language are not because of the presence of real numbers.

6.3.3 Judgements on Full Forms

Judgements on full forms are generically of the form $\Delta \vdash J$, and the general definition of the judgement is

1. $\emptyset \vdash J$ if

$\vdash^q J$

2. $x : \sigma, \Delta \vdash J$ if

   $\forall \eta \in \sigma \centerdot \{\eta/x\}\,\Delta \vdash \{\eta/x\}\,J$

If the context is empty, then all constructs in the judgement must be closed and the corresponding judgement on closed forms is called. If there is a variable, the judgement must be satisfied for every possible value of that variable.

As an example, consider the typing judgement for expressions $\Delta \vdash \varepsilon : \sigma$. Its definition is

1. $\emptyset \vdash \varepsilon : \sigma$ if

   $\vdash^q \varepsilon : \sigma$

2. $x : \sigma, \Delta \vdash \varepsilon : \sigma'$ if

   $\forall \eta \in \sigma \centerdot \{\eta/x\}\,\Delta \vdash \{\eta/x\}\,\varepsilon : \{\eta/x\}\,\sigma'$

Given that x is of type σ, the expression $x+3$ is well-formed if $\eta+3$ is well-formed for every $\eta \in \sigma$.

This defines the judgement, but it is not yet clear if this judgement has an implementation. It does. The values of any closed type σ can be enumerated to literally check all instances of the universally quantified judgement. More efficient implementations might be possible too, but we present just this one.

Let $S^q(\sigma)$ denote the elements of closed type σ. Its definition is

- $S^q(\sigma) = S^c(\sigma')$ if

  $\sigma \searrow \sigma'$

which simply evaluates the type to its canonical form and calculates the elements of the canonical type, given by $S^c(\sigma)$. The definition of $S^c(\sigma)$ is by induction on the form of σ,

1. $S^c(\{l_1, \ldots, l_m\}) = \{l_1, \ldots, l_m\}$
2. $S^c([\varepsilon_L, \varepsilon_U]) = \{\eta_L, \eta_L + 1, \ldots, \eta_U\}$ if

   $\varepsilon_L \searrow \eta_L$ and $\varepsilon_U \searrow \eta_U$

3. (a) $S^c(\mathtt{unit}) = \{()\}$

   (b) $S^c(x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m) =$

$$\bigcup_{\eta_1 \in S^q(\sigma_1)} \bigcup_{\eta_2 \in S^q(\{\eta_1/x_1\}\sigma_2)} \cdots \bigcup_{\eta_m \in S^q(\{\eta_1/x_1, \ldots, \eta_{m-1}/x_{m-1}\}\sigma_m)} \{(\eta_1, \ldots, \eta_m)\}$$

Rule 1 says that the elements of an enumerated type are exactly those elements. The argument $\{l_1, \ldots, l_m\}$ in $S^c(\{l_1, \ldots, l_m\})$ is a type in the object language, and the set on the right of the equal sign is in the meta-language.

In Rule 2, the elements of an interval $[\varepsilon_L, \varepsilon_U]$ are determined by first evaluating the lower and upper bound to integers η_L and η_U. The set consists of the integers from η_L through η_U.

Rule 3 considers two cases for the dependent product $x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m$. If $m = 0$, then this type is called `unit`. The set `unit` is defined to contain just the single element (). The case $m = 1$ is not allowed; it is considered invalid syntax. When $m \geq 2$, the Cartesian product of the individual sets is taken. However, subsequent sets can depend on the actual value of previous components, requiring a more subtle definition.

Consider enumerating the values of $x_1 : [1, 2] \times [1, x_1]$. The first type, $[1, 2]$ in this example, must be closed. If it were not, the dependent product would not be canonical, violating the precondition. Its elements can thus be enumerated. The first union is over $\eta_1 \in S^q([1, 2])$. The enumeration procedure for dependent products recursively calls the enumeration procedure on its component types. The second union is over $\eta_2 \in S^q(\{\eta_1/x_1\}[1, x_1])$. For each η_1, the second union first closes σ_2 and then enumerates its values. The free variables of η_2 can only be x_1 because otherwise the dependent product would not be canonical. In this way each possible tuple $(\eta_1, \ldots, \eta_m)$ is formed, and the union of all is the final answer.

6.4 Results

The main goal of the indexing logic is to enable the definition of sophisticated sets in a compact and intuitive manner. A rich syntax has been provided, and the judgement $\Delta \vdash \sigma :: \kappa$ allows determining that the syntax is being used correctly. Finally, the enumeration procedure allows explicitly obtaining the index set declared implicitly within the logic's syntax.

We show some examples of how index sets can be declared in the novel language defined in this chapter. The enumeration procedure is used to output the explicit set of elements. Interesting examples are difficult without use of the "let" constructs defined in Appendix B. In the following examples, we allow use of this syntax although it was not introduced in this chapter.

The concrete syntax is slightly different than the abstract syntax used in the theory. Type functions $\lambda x \,.\, \sigma$ are written in ASCII text as `fni x . sigma`, where sigma would be any type. The argument of a function application $\sigma\, \varepsilon$ is surrounded in square brackets. Other concrete syntax is a fairly obvious translation from the abstract syntax, but a complete discussion is provided in Appendix C.

Example 6.1 The following is a simple product type,

```
1 {'a','b'} * [1,3]
```

The enumeration procedure allows checking the explicit set being declared. It gives

```
{('a',1), ('a',2), ('a',3),
 ('b',1), ('b',2), ('b',3)}
```

which is a set of pairs.

Example 6.2 Next, consider a simple dependent product type,

```
1 i:[1,3] * [2,i]
```

We first check if this type satisfies $\emptyset \vdash (i : [1,3] * [2,i])$ :: IndexSet. This check fails and the following messages are printed,

```
ERROR: first expri should be less than or equal to second
  expri at 1.13: 2
  expri at 1.15: 1

MSG: ill-formed type, lower bound must be less than or equal to upper bound
  in typei at 1.12-1.16: [2, 1]

MSG: typei not of required kindi, previous messages should explain why
  contexti: i:[1, 3]
  typei at 1.12-1.16: [2, i]
  kindi: IndexSet
```

The first error states that 2 must be less than or equal to 1. The second message explains that this is needed to check that $[2,1]$ is a well-formed type. However, even $[2,1]$ is not a type that exists in the program as written. In a more complex example, it might still not be clear how this interval arose. The third message states that the judgement $i : [1,3] \vdash [2,i]$ :: IndexSet is being checked. It is now clear that the second component of the dependent product is being checked within the context containing the variable introduced by the first component. When i takes the value 1, the interval $[2,1]$ must be an IndexSet, which fails.

Let's fix the program to be

```
1 i:[1,3] * [1,i]
```

This type is a well-formed index set. The enumeration procedure shows the elements explicitly to be

```
{(1,1),
 (2,1), (2,2),
 (3,1), (3,2), (3,3)}
```

The values of the second component are always less than or equal to the first.

Example 6.3 A more physical example demonstrates the importance of additional features. Index sets defined in the following program would be useful in a scheduling problem.

```
1  let
2    set JOBS = {'a','b','c'}
3
4    typei RUNS_ON = fni j . case j of
5                              'a' => {'s1','s2'}
6                            | 'b' => {'s1','s3','s4'}
7                            | 'c' => {'s3','s4'}
8  in
9    j:JOBS * RUNS_ON[j]
10 end
```

A set of jobs {'a','b','c'} is given the name JOBS. The keyword set indicates that an index type is being named. Additionally, it checks that the index type is of kind IndexSet. Next, another index type RUNS_ON is defined. It is not an index set, so the more general keyword typei is used. RUNS_ON is a type function. Given a job j, it returns the stages that job runs on. Finally, these two definitions are used in the overall index type being defined, j:JOBS * RUNS_ON[j]. The second component applies the function RUNS_ON to whatever value the first component takes.

The entire above input is a type in the indexing logic, i.e. of the syntactic form σ. We first check that the judgement $\emptyset \vdash \sigma$ :: IndexSet is satisfied. It is. Then, the elements of this type can be enumerated. Explicitly, the above type is

```
{('a','s1'), ('a','s2'),
 ('b','s1'), ('b','s3'), ('b','s4'),
 ('c','s3'), ('c','s4')}
```

The set pairs up jobs with only the stages that that job runs on.

Another part of a complete program may require only the stages that job 'a' runs on. The type RUNS_ON['a'] would provide this. Yet another part might be concerned with only jobs 'a' and 'b', and their corresponding stages. The type j:{'a','b'} * RUNS_ON[j] would refer to the set

```
{('a','s1'), ('a','s2'),
 ('b','s1'), ('b','s3'), ('b','s4')}
```

Chapter 7

Indexed Mathematical Programs

In Chapter 4, we formulated unindexed mathematical programs (MPs) as a formal logic. Chapter 6 presented a logic for modeling index sets. Now, we combine the works of these two chapters to present a theory of indexed mathematical programs.

Referring to the resulting systems as mathematical programs is somewhat misleading. The theory we present enhances traditional MP with functions (with finite domains) and indexed operations. These allow genuinely novel algorithms, in which indices are not merely a syntactical convenience for expressing a large number of similar computations. In some cases, the operation can actually be implemented as a single computation. This is possible because, just like numbers, indexed constructs are part of the formal theory, not a syntactic feature eliminated prior to computation.

On the other hand, indexed constructs can be eliminated if desired, to interface to current solvers for example. In this case, they still serve the important role of providing an enriched modeling language. As reviewed in the Introduction chapter, indexing is one of the primary features of existing MP modeling software. Our work provides a rigorous design methodology for such software.

The syntax of the logic defined here is an extension of that in Chapter 4. So there is some repetition in the definitions, but the discussion focuses on the extensions only. Adding indexing is sometimes straightforward, but some judgements become fundamentally more complex.

7.1 Syntax

7.1.1 Full Forms

The current language specification will use the notation τ for types and e for expressions. The types and expressions of the indexing language were denoted σ and ε. References to constructs in the indexing language will always be qualified as such. The unqualified terms "expression" and "type" refer to constructs of the full programming language, but

sometimes we use the qualification "program" for emphasis.

Also, now we use i to denote variables in the indexing language—these are of some type σ and are called index variables. The symbol x is reserved for variables in the full programming language—these are of some type τ and are called program variables.

7.1.1.1 Types

The types of the language are

$$\tau ::= \texttt{real} \mid \texttt{bool} \mid i : \sigma \to \tau \tag{7.1}$$

The type $i : \sigma \to \tau$ represents an indexed family of types, or alternatively, a mapping from an index set σ to a type τ. This is a dependent type, but the dependency is only on index variables, not program variables. An example is $i : [1, 10] \to ([1, i] \to \texttt{real})$. Given an i equal to a particular value k, functions of this type will return a function of type $[1, k] \to \texttt{real}$. The type $i : \sigma \to \tau$ formalizes the notion of indexed terms. A variable of this type is what is normally call an indexed variable, and a constant of this type is an indexed parameter.

7.1.1.2 Expressions

The syntax for expressions is

$$\begin{aligned} e ::= {} & x \mid r \mid \texttt{true} \mid \texttt{false} \\ & \mid -e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \\ & \mid \texttt{not}\, e \mid e_1 \,\texttt{or}\, e_2 \mid e_1 \,\texttt{and}\, e_2 \\ & \mid \sum_{i:\sigma} e \\ & \mid \texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow e_j\}_{j=1}^{m} \mid \texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow e_j\}_{j=1}^{m} \\ & \mid \lambda i : \sigma \,\texttt{.}\, e \mid e\,\varepsilon \\ & \mid e : \tau \end{aligned} \tag{7.2}$$

The first expression involving indexing is $\sum_{i:\sigma} e$, a generalization of binary addition. The $\sum$ is a variable binder. It introduces a new index variable i of index type σ. The scope of i is e, the body of the $\sum$.

Case expressions are familiar from the indexing logic, but now an ascription $i.\tau$ is required to provide the type of the i^{th} branch. The semantically defined type system circumvented this need, but the MP type system requires this information.

The notation $\lambda i : \sigma \,\texttt{.}\, e$ refers to a function whose input argument is named i and is of type σ. The expression $e\,\varepsilon$ applies an expression e to an index expression ε.

Finally, the expression "$e : \tau$" is called a type ascription. It allows ascribing a type to an expression. This is useful for documentation purposes. If expression e is very complicated,

programming errors might cause it to be of a type other than the intended one. Ascribing the type τ to e causes the type checker to verify that e is indeed of type τ. Also, type ascription is necessary to implement a type inference procedure discussed in Section 7.2.5.

7.1.1.3 Propositions

The syntax for propositions, or constraints, is

$$
\begin{aligned}
c ::= {} & \mathtt{T} \mid \mathtt{F} \\
\mid {} & \mathtt{isTrue}\ e \mid e_1 = e_2 \mid e_1 \leq e_2 \\
\mid {} & c_1 \vee c_2 \mid c_1 \wedge c_2 \\
\mid {} & \exists x : \tau \,\mathtt{.}\, c \\
\mid {} & \bigvee_{i:\sigma} c \mid \bigwedge_{i:\sigma} c \\
\mid {} & \mathtt{case}_{i.\zeta}\ \varepsilon\ \mathtt{of}\ \{l_j \Rightarrow c_j\}_{j=1}^{m} \mid \mathtt{case}_{i.\zeta}\ \varepsilon\ \mathtt{of}\ \{k_j \Rightarrow c_j\}_{j=1}^{m} \\
\mid {} & \lambda i : \sigma \,\mathtt{.}\, c \mid c\,\varepsilon \\
\mid {} & c : \zeta
\end{aligned}
\tag{7.3}
$$

The extensions are similar to that for expressions. Indexed versions of binary operations are provided. Case constructs represent tables of propositions. The propositional function

$$
\begin{aligned}
\lambda i : [1,2] \,\mathtt{.}\, \mathtt{case}\ i\ \mathtt{of}& \\
& 1 \Rightarrow (x_1 = 3) \\
& \mid 2 \Rightarrow (x_1 = x_2 + x_3)
\end{aligned}
$$

can be applied to an index expression of type $[1,2]$. Let this function be named f. Then $f\,(1)$ is equivalent to the proposition $x_1 = 3$, and $f\,(2)$ gives the proposition $x_1 = x_2 + x_3$. Finally, analogously to the expression $e : \tau$, we provide propositional type ascription $c : \zeta$.

7.1.1.4 Propositional Types

In unindexed MPs, the judgement c PROP means c is a well-formed proposition. The only category to which propositions can belong is PROP. Indexed programs require enriching the categories into which propositions are placed. Some are truly propositions, statements which are satisfied or not. Others, indexed families of propositions. The types of propositions are

$$
\zeta ::= \mathtt{Prop} \mid i : \sigma \to \zeta \tag{7.4}
$$

`Prop` represents the set of proper propositions, just like PROP in the unindexed theory. The propositional function type $i : \sigma \to \zeta$ categorizes propositional functions. It includes

those functions which, when applied to an index expression of type σ, return a proposition of type ζ.

7.1.1.5 Programs

Finally, the syntax of a program is

$$p ::= \delta_{x_1:\tau_1,\ldots,x_m:\tau_m} \{e \mid c\} \tag{7.5}$$

where $\delta ::= \texttt{min} \mid \texttt{max}$. This is unchanged from unindexed MPs, but the types, expressions, and propositions it employs are of the enriched syntax.

7.1.2 Meta-Operations Relating to Variables

Variables of two different natures exist in indexed programs: index variables and MP variables, which are being denoted with i and x, respectively. All constructs can contain index variables. Expressions, propositions, and programs contain MP variables. In general, we need methods for determining the free index and MP variables for each syntactic construct. Let s refer to any one of the syntactic constructs ε, σ, κ, e, τ, c, ζ, or p.

We need to define $\{\varepsilon/i\}\, s$—the substitution of an index expression into every syntactic construct—and $\{e/x\}\, e'$, $\{e/x\}\, c$, and $\{e/x\}\, p$—the substitution of MP expressions. That is already eleven substitution procedures. In our full implementation we actually have variables in every syntactic construct. For example, one can write $R = [1, 10]$, which gives the name R to the index type $[1, 10]$. This is only a concrete syntax feature. All instances of R are replaced with $[1, 10]$ prior to any operation. But this means we actually need to define $\{s/x\}\, s'$ for every combination of syntactic categories s and s', which is $8 * 8 = 64$ functions (actually somewhat less because not all combinations are needed). Other operations involving variables are also required, such as α-conversion and determining the set of free variables.

These are not difficult operations to define. They were defined for the unindexed programs and indexing logic, but there is now a combinatorial increase in the number of such functions needed. Writing such a large number of functions is tedious and error prone. Instead, we present a meta-logic in Appendix B, within which the present logic can be defined. Several operations relating to variables are defined once in the meta-logic and become available for free for the numerous special cases needed.

Suffice it here to know that the following operations are available for all combinations of s and s':

- $\{s/x\}\, s'$, substitution of s for a variable x (of the same syntactic category as s) into s',
- s CLOSED, construct s has no free variables of any syntactic category,
- α-conversion, renaming of variable names for any construct.

Several other variable operations are also presented in Appendix B but are not needed directly in this chapter.

7.2 Type System

The syntax of indexed mathematical programs involves both index and program variables. Correspondingly, typing judgements in this logic require two contexts, a context of index variables Δ and a context of program variables Γ. The judgements defined here employ judgements defined in the indexing logic.

7.2.1 Well-Formed Type

Let $\vdash_\Delta \tau$ TYPE mean τ is a well-formed type in the given indexing context, which is assumed to satisfy Δ CTXT. Types do not contain program variables, so a program context is not required. Its definition is

$$\frac{}{\vdash_\Delta \texttt{real}\ \text{TYPE}} \tag{7.6a}$$

$$\frac{}{\vdash_\Delta \texttt{bool}\ \text{TYPE}} \tag{7.6b}$$

$$\frac{\Delta \vdash \sigma :: \texttt{IndexSet} \quad \vdash_{\Delta, i:\sigma} \tau\ \text{TYPE}}{\vdash_\Delta i : \sigma \to \tau\ \text{TYPE}} \tag{7.6c}$$

The function type $i : \sigma \to \tau$ firstly requires that σ be an `IndexSet` because i must be allowed to take some value, and only index types of kind `IndexSet` contain expressions. Secondly, the type τ in $i : \sigma \to \tau$ is required to be itself well-formed in the index context augmented with $i : \sigma$. Note the scope of i in $i : \sigma \to \tau$ is only τ, not σ.

7.2.2 Well-Formed Context

Let $\vdash_\Delta \Gamma$ CTXT mean context Γ is well-formed, where Δ CTXT is assumed. The definition is by induction on the construction of Γ,

$$\frac{}{\vdash_\Delta \emptyset\ \text{CTXT}} \tag{7.7a}$$

$$\frac{\vdash_\Delta \Gamma\ \text{CTXT} \quad \vdash_\Delta \tau\ \text{TYPE}}{\vdash_\Delta \Gamma, x : \tau\ \text{CTXT}} \tag{7.7b}$$

This simply checks that all declared types are well-formed.

7.2.3 Type Equivalence

In unindexed programs, only syntactically identical types could be equivalent. Now, we have the possibility that two types which are not obviously the same, i.e. syntactically identical, might still be definitionally equivalent. Let $\vdash_\Delta \tau_1 \equiv \tau_2$ mean types τ_1 and τ_2 are equivalent, where Δ CTXT, $\vdash_\Delta \tau_1$ TYPE, and $\vdash_\Delta \tau_2$ TYPE are assumed. It is assumed that the two types are well-formed. There is no reason to check equivalence of ill-formed types. The definition

is given by the rules

$$\frac{}{\vdash_\Delta \texttt{real} \equiv \texttt{real}} \tag{7.8a}$$

$$\frac{}{\vdash_\Delta \texttt{bool} \equiv \texttt{bool}} \tag{7.8b}$$

$$\frac{\Delta \vdash \sigma_2 \equiv \sigma_1 :: \texttt{IndexSet} \quad \vdash_{\Delta, i:\sigma_2} \tau_1 \equiv \tau_2}{\vdash_\Delta (i : \sigma_1 \to \tau_1) \equiv (i : \sigma_2 \to \tau_2)} \tag{7.8c}$$

Equivalence of the basic types `real` and `bool` is just a reflexivity relation. Two function types are equivalent if both the input and output types are equivalent. Consider first function types with index domains. The input types are index types σ_1 and σ_2, and we check that these are equivalent using a judgement defined in the previous chapter. Secondly, the output types τ_1 and τ_2 must be equivalent in the context augmented with $i : \sigma_2$. It would be equally valid to add $i : \sigma_1$ instead because σ_1 and σ_2 are equivalent.

As in the indexing logic, judgement definitions are presented modulo α-conversion. We have assumed the introduced variable in both function types is i. In actuality, the variables might be distinct but the types still equivalent. It is understood that the bound variable names are made identical prior to applying the above rules. This can always be done using the α-conversion method defined in Appendix B.

7.2.4 Type of Expression

Let $\Gamma \vdash_\Delta e : \tau$ mean e is of type τ in the given contexts, where Δ CTXT, $\vdash_\Delta \Gamma$ CTXT, and $\vdash_\Delta \tau$ TYPE are assumed. Its definition is

$$\frac{}{\Gamma \vdash_\Delta x : \tau} \text{ where } (x : \tau) \in \Gamma \tag{7.9a}$$

$$\frac{}{\Gamma \vdash_\Delta r : \texttt{real}} \tag{7.9b}$$

$$\frac{}{\Gamma \vdash_\Delta \texttt{true} : \texttt{bool}} \tag{7.9c}$$

$$\frac{}{\Gamma \vdash_\Delta \texttt{false} : \texttt{bool}} \tag{7.9d}$$

$$\frac{\Gamma \vdash_\Delta e : \texttt{real}}{\Gamma \vdash_\Delta -e : \texttt{real}} \tag{7.9e}$$

$$\frac{\Gamma \vdash_\Delta e_1 : \texttt{real} \quad \Gamma \vdash_\Delta e_2 : \texttt{real}}{\Gamma \vdash_\Delta e_1 \,\texttt{op}\, e_2 : \texttt{real}} \text{ for } \texttt{op} \in \{+, -, *\} \tag{7.9f}$$

$$\frac{\Gamma \vdash_\Delta e : \texttt{bool}}{\Gamma \vdash_\Delta \texttt{not}\, e : \texttt{bool}} \tag{7.9g}$$

$$\frac{\Gamma \vdash_\Delta e_1 : \texttt{bool} \quad \Gamma \vdash_\Delta e_2 : \texttt{bool}}{\Gamma \vdash_\Delta e_1 \,\texttt{op}\, e_2 : \texttt{bool}} \text{ for } \texttt{op} \in \{\texttt{or}, \texttt{and}\} \tag{7.9h}$$

$$\frac{\Delta \vdash \sigma :: \texttt{IndexSet} \quad \Gamma \vdash_{\Delta, i:\sigma} e : \texttt{real}}{\Gamma \vdash_\Delta \sum_{i:\sigma} e : \texttt{real}} \tag{7.9i}$$

$$\frac{\vdash_{\Delta, i:\{l_1,\ldots,l_m\}} \tau \text{ TYPE} \quad \Delta \vdash \varepsilon : \{l_1, \ldots, l_m\} \quad \{\Gamma \vdash_\Delta e_j : \{l_j/i\}\,\tau\}_{j=1}^m}{\Gamma \vdash_\Delta \texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow e_j\}_{j=1}^m : \{\varepsilon/i\}\,\tau} \tag{7.9j}$$

$$\frac{\vdash_{\Delta, i:[k_1,k_m]} \tau \text{ TYPE} \quad \Delta \vdash \varepsilon : [k_1, k_m] \quad \{\Gamma \vdash_\Delta e_j : \{k_j/i\}\,\tau\}_{j=1}^m}{\Gamma \vdash_\Delta \texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow e_j\}_{j=1}^m : \{\varepsilon/i\}\,\tau} \tag{7.9k}$$

$$\frac{\Delta \vdash \sigma' :: \texttt{IndexSet} \quad \Delta \vdash \sigma' \equiv \sigma :: \texttt{IndexSet} \quad \Gamma \vdash_{\Delta, i:\sigma} e : \tau}{\Gamma \vdash_\Delta (\lambda i : \sigma' \,.\, e) : (i : \sigma \to \tau)} \tag{7.9l}$$

$$\frac{\Gamma \vdash_\Delta e : (i : \sigma \to \tau) \quad \Delta \vdash \varepsilon : \sigma}{\Gamma \vdash_\Delta e\,\varepsilon : \{\varepsilon/i\}\,\tau} \tag{7.9m}$$

$$\frac{\vdash_\Delta \tau' \text{ TYPE} \quad \vdash_\Delta \tau' \equiv \tau \quad \Gamma \vdash_\Delta e : \tau}{\Gamma \vdash_\Delta (e : \tau') : \tau} \tag{7.9n}$$

$$\frac{\Gamma \vdash_\Delta e : \tau' \quad \vdash_\Delta \tau' \equiv \tau}{\Gamma \vdash_\Delta e : \tau}\text{subsumption} \tag{7.9o}$$

Rules not involving index expressions are similar to those given in Chapter 4, except now an index context is also required.

The rule for type checking $\sum_{i:\sigma} e$ first requires that the σ is indeed an index set. Otherwise, it is erroneous to declare a variable i of type σ. Within the context augmented with $i : \sigma$, the body e must be of type `real`. If these conditions are met, the whole summation expression is of type `real`.

Type checking expression $\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow e_j\}_{j=1}^{m}$ firstly requires the introduced τ to be well-formed and the case object ε to be of type $\{l_1, \ldots, l_m\}$. The rule requires information about what the type of each branch is supposed to be, as provided by the ascription $i \,.\, \tau$. This declaration is interpreted to mean the j^{th} branch has type $\{l_j/i\}\,\tau$, which is checked by the final precondition. If this is satisfied, the type of the case expression is $\{\varepsilon/i\}\,\tau$.

Next, a function $\lambda i : \sigma \,.\, e$ is declared to be of type $i : \sigma \to \tau$ if its body e is of type τ, after adding $i : \sigma$ to the indexing context. Following this, we have function application $e\,\varepsilon$. The expression e being applied must be of a function type $i : \sigma \to \tau$. The argument ε being applied must be of the type expected by the function type, which is σ. The returned expression will be of type $\{\varepsilon/i\}\,\tau$.

As an example, let f be the expression

$$\begin{aligned}\lambda i_1 : [1,2] \,.\, \lambda i_2 : [1, i_1] \,.\, \texttt{case}\ i_1\ \texttt{of}&\\ 1 &\Rightarrow (\texttt{case}\ i_2\ \texttt{of}\ 1 \Rightarrow 3.4)\\ \mid 2 &\Rightarrow (\texttt{case}\ i_2\ \texttt{of}\ 1 \Rightarrow 1.3 \mid 2 \Rightarrow 5.6)\end{aligned}$$

which is of type $i_1 : [1,2] \to ([1, i_1] \to \texttt{real})$. If f is applied to 1, the resulting expression is

$$\begin{aligned}\lambda i_2 : [1,1] \,.\, \texttt{case}\ 1\ \texttt{of}&\\ 1 &\Rightarrow (\texttt{case}\ i_2\ \texttt{of}\ 1 \Rightarrow 3.4)\\ \mid 2 &\Rightarrow (\texttt{case}\ i_2\ \texttt{of}\ 1 \Rightarrow 1.3 \mid 2 \Rightarrow 5.6)\,.\end{aligned}$$

This case object is a constant and so this reduces to

$$\lambda i_2 : [1,1] \,.\, \texttt{case}\ i_2\ \texttt{of}\ 1 \Rightarrow 3.4$$

which is of type $[1,1] \to \texttt{real}$, i.e. $\{1/i_1\}\,([1, i_1] \to \texttt{real})$. Similarly, $f\,(2)$ would be of type $[1,2] \to \texttt{real}$, which is $\{2/i_1\}\,([1, i_1] \to \texttt{real})$.

The expression "$e : \tau$" is of type τ if e is of type τ. This simply verifies that the ascription is correct.

The final rule is called a subsumption rule and applies to all expressions. It says that if e is of type τ' and τ' is equivalent to τ, then e is also of type τ. An expression of type τ' can also be treated as being of any equivalent type τ.

7.2.5 Algorithmic Type of Expression

In unindexed programs, all typing judgements immediately lended themselves to the top-down proof strategy. The definition of $\Gamma \vdash_\Delta e : \tau$ does not because of complications arising in the subsumption rule and the rule for the application form $e\,\varepsilon$.

There are now two rules for every expression form: one rule that is specifically for that form, and the subsumption rule which applies to every expression. A particular e can satisfy the judgement if the preconditions of either rule are satisfied. The top-down proof strategy

thus requires that both sets of preconditions be checked, causing branching in the prover. This is less efficient but possible.

The real trouble with the subsumption rule is in its preconditions. The τ' in the preconditions does not appear in its conclusion. Imagine we are checking $\Gamma \vdash_\Delta e : \tau$ for some specific Γ, Δ, e, and τ. The precondition requires us to check $\Gamma \vdash_\Delta e : \tau'$, but from where do we get the τ'? It must be guessed from amongst an infinity of possibilities. The same situation occurs in the rule for the application form. This issue is well known, and a technique called bidirectional type checking resolves it.

Let $\Gamma \vdash_\Delta e \downarrow \tau$ be an analysis judgement and $\Gamma \vdash_\Delta e \uparrow \tau$ a synthesis judgement. Analysis means it is checked whether a given Γ, Δ, e, and τ satisfy the typing judgement. This is the normal way of reading a judgement. Synthesis means only Γ, Δ, and e are given. A type τ must be synthesized, i.e. returned, such that the given Γ, Δ, and e, and the synthesized τ satisfy the typing judgement.

Since the synthesis judgement generates the type of an expression, it is possible to reduce the amount of type declarations required from a user of the language. A function $\lambda i : \sigma \centerdot e$ requires the type of its input argument to be specified. We can now modify this syntax to $\lambda i \centerdot e$, omitting the type declaration. Sometimes this omission makes it impossible to determine whether an expression is well-formed or not. This is not a problem because one can always use the expression form $e : \tau$ to explicitly provide the type information.

Expressions are split into those whose types can be synthesized versus those that can be analyzed. The idea is to synthesize whenever possible to provide maximum type inference. Also because if a type can be synthesized it can certainly be analyzed but not vice versa. If we can synthesize a type τ for a given e, we can certainly check that e is of a given type τ.

The analysis judgement $\Gamma \vdash_\Delta e \downarrow \tau$ is the one that corresponds to the original judgement $\Gamma \vdash_\Delta e : \tau$ of interest. Its definition is

$$\frac{\Gamma \vdash_{\Delta, i:\sigma} e \downarrow \tau}{\Gamma \vdash_\Delta (\lambda i \centerdot e) \downarrow (i : \sigma \to \tau)} \tag{7.10a}$$

$$\frac{\Gamma \vdash_\Delta e \uparrow \tau' \quad \vdash_\Delta \tau' \equiv \tau}{\Gamma \vdash_\Delta e \downarrow \tau} \text{ where } e \text{ not } \lambda i \centerdot e' \tag{7.10b}$$

In the first rule, a function is checked to be of a given a function type. The second rule is the subsumption rule. When checking whether a given e is of a given type τ, we attempt to synthesize a type τ' for e and check if τ' is equivalent to τ. Type τ' no longer needs to be guessed because the synthesis judgement provides it.

The definition of $\Gamma \vdash_\Delta e \uparrow \tau$ includes a rule for every form except $\lambda i \,.\, e$,

$$\frac{}{\Gamma \vdash_\Delta x \uparrow \tau} \text{ where } (x : \tau) \in \Gamma \tag{7.11a}$$

$$\frac{}{\Gamma \vdash_\Delta r \uparrow \texttt{real}} \tag{7.11b}$$

$$\frac{}{\Gamma \vdash_\Delta \texttt{true} \uparrow \texttt{bool}} \tag{7.11c}$$

$$\frac{}{\Gamma \vdash_\Delta \texttt{false} \uparrow \texttt{bool}} \tag{7.11d}$$

$$\frac{\Gamma \vdash_\Delta e \downarrow \texttt{real}}{\Gamma \vdash_\Delta -e \uparrow \texttt{real}} \tag{7.11e}$$

$$\frac{\Gamma \vdash_\Delta e_1 \downarrow \texttt{real} \quad \Gamma \vdash_\Delta e_2 \downarrow \texttt{real}}{\Gamma \vdash_\Delta e_1 \,\texttt{op}\, e_2 \uparrow \texttt{real}} \text{ for } \texttt{op} \in \{+, -, *\} \tag{7.11f}$$

$$\frac{\Gamma \vdash_\Delta e \downarrow \texttt{bool}}{\Gamma \vdash_\Delta \texttt{not}\, e \uparrow \texttt{bool}} \tag{7.11g}$$

$$\frac{\Gamma \vdash_\Delta e_1 \downarrow \texttt{bool} \quad \Gamma \vdash_\Delta e_2 \downarrow \texttt{bool}}{\Gamma \vdash_\Delta e_1 \,\texttt{op}\, e_2 \uparrow \texttt{bool}} \text{ for } \texttt{op} \in \{\texttt{or}, \texttt{and}\} \tag{7.11h}$$

$$\frac{\Delta \vdash \sigma :: \texttt{IndexSet} \quad \Gamma \vdash_{\Delta, i:\sigma} e \downarrow \texttt{real}}{\Gamma \vdash_\Delta \sum_{i:\sigma} e \uparrow \texttt{real}} \tag{7.11i}$$

$$\frac{\vdash_{\Delta, i:\{l_1,\ldots,l_m\}} \tau \text{ TYPE} \quad \Delta \vdash \varepsilon : \{l_1, \ldots, l_m\} \quad \{\Gamma \vdash_\Delta e_j \downarrow \{l_j/i\}\,\tau\}_{j=1}^m}{\Gamma \vdash_\Delta \texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow e_j\}_{j=1}^m \uparrow \{\varepsilon/i\}\,\tau} \tag{7.11j}$$

$$\frac{\vdash_{\Delta, i:[k_1,k_m]} \tau \text{ TYPE} \quad \Delta \vdash \varepsilon : [k_1, k_m] \quad \{\Gamma \vdash_\Delta e_j \downarrow \{k_j/i\}\,\tau\}_{j=1}^m}{\Gamma \vdash_\Delta \texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow e_j\}_{j=1}^m \uparrow \{\varepsilon/i\}\,\tau} \tag{7.11k}$$

$$\frac{\Gamma \vdash_\Delta e \uparrow (i : \sigma \to \tau) \quad \Delta \vdash \varepsilon : \sigma}{\Gamma \vdash_\Delta e\,\varepsilon \uparrow \{\varepsilon/i\}\,\tau} \tag{7.11l}$$

$$\frac{\Gamma \vdash_\Delta e \downarrow \tau}{\Gamma \vdash_\Delta (e : \tau) \uparrow \tau} \tag{7.11m}$$

A variable x's type can be synthesized. Given the variable, we simply check what its declared type is in the context and return that type. An arithmetic expression such as $e_1 + e_2$ can also have its type synthesized. It can only return type `real`, and only if the sub-expressions can each be analyzed to be of type `real`. Other numeric and Boolean expressions are similar.

It is possible to synthesize the type of a case expression because of the information $i \,.\, \tau$ provided as part of the expression. The result is $\{\varepsilon/i\}\,\tau$ if each branch can be checked to be of type $\{l_j/i\}\,\tau$.

The type of an application $e\,\varepsilon$ can also be synthesized, assuming the type of e can be synthesized. The type synthesized for e must be of the form $i : \sigma \to \tau$ because it is being applied to an index expression. Given that such a type is synthesized, we now have the input type of the function and can check that ε is of this type. If so, it is possible to conclude that τ is the type of the whole expression.

In conclusion, the type analysis judgement $\Gamma \vdash_\Delta e \downarrow \tau$ is the algorithmic version of $\Gamma \vdash_\Delta e : \tau$. We consider the latter to be the definition of the typing relation, and always refer to it in the theory, and the former is its implementation.

7.2.6 Well-Formed Propositional Type

Let $\vdash_\Delta \zeta$ PROP_TYPE mean ζ is a well-formed propositional type, where Δ CTXT is assumed Its definition is

$$\frac{}{\vdash_\Delta \texttt{Prop}\ \text{PROP\_TYPE}} \tag{7.12a}$$

$$\frac{\Delta \vdash \sigma :: \texttt{IndexSet} \quad \vdash_{\Delta, i:\sigma} \zeta\ \text{PROP\_TYPE}}{\vdash_\Delta i : \sigma \to \zeta\ \text{PROP\_TYPE}} \tag{7.12b}$$

These rules are analogous to those defining well-formed types τ.

7.2.7 Propositional Type Equivalence

Let $\vdash_\Delta \zeta_1 \equiv \zeta_2$ mean propositional types ζ_1 and ζ_2 are equivalent, where Δ CTXT, $\vdash_\Delta \zeta_1$ PROP_TYPE and $\vdash_\Delta \zeta_2$ PROP_TYPE are assumed. The definition is given by the rules

$$\frac{}{\vdash_\Delta \texttt{Prop} \equiv \texttt{Prop}} \tag{7.13a}$$

$$\frac{\Delta \vdash \sigma_2 \equiv \sigma_1 :: \texttt{IndexSet} \quad \vdash_{\Delta, i:\sigma_2} \zeta_1 \equiv \zeta_2}{\vdash_\Delta (i : \sigma_1 \to \zeta_1) \equiv (i : \sigma_2 \to \zeta_2)} \tag{7.13b}$$

Again, the rules are analogous to type equivalence.

7.2.8 Type of Proposition

Just as expressions e are categorized into types τ, propositions c are categorized into propositional types ζ. Let $\Gamma \vdash_\Delta c : \zeta$ mean c is of propositional type ζ, where Δ CTXT, $\vdash_\Delta \Gamma$ CTXT,

and $\vdash_\Delta \zeta$ PROP_TYPE are assumed. The definition is inductive on c,

$$\frac{}{\Gamma \vdash_\Delta \mathtt{T} : \mathtt{Prop}} \tag{7.14a}$$

$$\frac{}{\Gamma \vdash_\Delta \mathtt{F} : \mathtt{Prop}} \tag{7.14b}$$

$$\frac{\Gamma \vdash_\Delta e : \mathtt{bool}}{\Gamma \vdash_\Delta \mathtt{isTrue}\, e : \mathtt{Prop}} \tag{7.14c}$$

$$\frac{\Gamma \vdash_\Delta e_1 : \mathtt{real} \quad \Gamma \vdash_\Delta e_2 : \mathtt{real}}{\Gamma \vdash_\Delta e_1 \,\mathtt{op}\, e_2 : \mathtt{Prop}} \text{ for } \mathtt{op} \in \{=, \leq\} \tag{7.14d}$$

$$\frac{\Gamma \vdash_\Delta c_1 : \mathtt{Prop} \quad \Gamma \vdash_\Delta c_2 : \mathtt{Prop}}{\Gamma \vdash_\Delta c_1 \,\mathtt{op}\, c_2 : \mathtt{Prop}} \text{ for } \mathtt{op} \in \{\vee, \wedge\} \tag{7.14e}$$

$$\frac{\vdash_\Delta \tau \text{ TYPE} \quad \Gamma, x : \tau \vdash c : \mathtt{Prop}}{\Gamma \vdash_\Delta \exists x : \tau \,.\, c : \mathtt{Prop}} \tag{7.14f}$$

$$\frac{\Delta \vdash \sigma :: \mathtt{IndexSet} \quad \Gamma \vdash_{\Delta, i:\sigma} c : \mathtt{Prop}}{\Gamma \vdash_\Delta \underset{i:\sigma}{\mathtt{OP}}\, c : \mathtt{Prop}} \text{ for } \mathtt{OP} \in \{\bigvee, \bigwedge\} \tag{7.14g}$$

$$\frac{\vdash_{\Delta, i:\{l_1,\ldots,l_m\}} \zeta \text{ PROP\_TYPE} \quad \Delta \vdash \varepsilon : \{l_1, \ldots, l_m\} \quad \{\Gamma \vdash_\Delta c_j : \{l_j/i\}\,\zeta\}_{j=1}^m}{\Gamma \vdash_\Delta \mathtt{case}_{i.\zeta}\ \varepsilon\ \mathtt{of}\ \{l_j \Rightarrow c_j\}_{j=1}^m : \{\varepsilon/i\}\,\zeta} \tag{7.14h}$$

$$\frac{\vdash_{\Delta, i:\{l_1,\ldots,l_m\}} \zeta \text{ PROP\_TYPE} \quad \Delta \vdash \varepsilon : [k_1, k_m] \quad \{\Gamma \vdash_\Delta c_j : \{k_j/i\}\,\zeta\}_{j=1}^m}{\Gamma \vdash_\Delta \mathtt{case}_{i.\zeta}\ \varepsilon\ \mathtt{of}\ \{k_j \Rightarrow c_j\}_{j=1}^m : \{\varepsilon/i\}\,\zeta} \tag{7.14i}$$

$$\frac{\Delta \vdash \sigma' :: \mathtt{IndexSet} \quad \Delta \vdash \sigma' \equiv \sigma \quad \Gamma \vdash_{\Delta, i:\sigma} c : \zeta}{\Gamma \vdash_\Delta (\lambda i : \sigma' \,.\, c) : (i : \sigma \to \zeta)} \tag{7.14j}$$

$$\frac{\Gamma \vdash_\Delta c : (i : \sigma \to \zeta) \quad \Delta \vdash \varepsilon : \sigma}{\Gamma \vdash_\Delta c\varepsilon : \{\varepsilon/i\}\,\zeta} \tag{7.14k}$$

$$\frac{\vdash_\Delta \zeta' \text{ PROP\_TYPE} \quad \vdash_\Delta \zeta' \equiv \zeta \quad \Gamma \vdash_\Delta c : \zeta}{\Gamma \vdash_\Delta (c : \zeta') : \zeta} \tag{7.14l}$$

$$\frac{\Gamma \vdash_\Delta c : \zeta' \quad \vdash_\Delta \zeta' \equiv \zeta}{\Gamma \vdash_\Delta c : \zeta} \text{subsumption} \tag{7.14m}$$

The rules are similar to those defining the types of expressions. Again, there is a subsumption rule and application forms, and the identical issues arise. An algorithmic propositional typing judgement can be provided in the same way as for expression type checking.

7.2.9 Well-Formed Program

Let p MP mean program p is a well-formed program. Its definition is

$$\frac{\begin{array}{c}\{\vdash_\emptyset \tau_j \text{ TYPE}\}_{j=1}^m \\ x_1 : \tau_1, \ldots, x_m : \tau_m \vdash_\emptyset e : \mathtt{real} \\ x_1 : \tau_1, \ldots, x_m : \tau_m \vdash_\emptyset c : \mathtt{Prop}\end{array}}{\delta_{x_1:\tau_1,\ldots,x_m:\tau_m}\,\{e \mid c\} \text{ MP}} \tag{7.15}$$

which is similar to the unindexed theory because the outer syntactic form of programs is unchanged. The only difference is that we now pass in an empty index context to the preconditions because index variables are not introduced at the program level.

7.3 Refined Types

The general idea of refined types, or domains, was discussed in Section 4.4. The present logic has an enriched type system and requires a corresponding enrichment of type refinements. Their syntax is

$$\begin{aligned}\rho ::= {} & \langle r_L, r_U\rangle \mid \langle r_L, \infty) \mid (-\infty, r_U\rangle \mid \texttt{real} \\ & \mid [r_L, k_U] \mid [r_L, \infty) \mid (-\infty, r_U] \mid \texttt{int} \\ & \mid \{\texttt{true}\} \mid \{\texttt{false}\} \mid \texttt{bool} \\ & \mid i : \sigma \to \rho \end{aligned} \tag{7.16}$$

These are a straightforward extension of the refined types in the unindexed theory.

The coarse type τ of a refined type ρ is given by the judgement $\rho \subseteq \tau$, defined by the rules

$$\frac{}{\langle r_L, r_U\rangle \subseteq \texttt{real}} \tag{7.17a}$$

$$\frac{}{\langle r_L, \infty) \subseteq \texttt{real}} \tag{7.17b}$$

$$\frac{}{(-\infty, r_U\rangle \subseteq \texttt{real}} \tag{7.17c}$$

$$\frac{}{\texttt{real} \subseteq \texttt{real}} \tag{7.17d}$$

$$\frac{}{[r_L, r_U] \subseteq \texttt{real}} \tag{7.17e}$$

$$\frac{}{[r_L, \infty) \subseteq \texttt{real}} \tag{7.17f}$$

$$\frac{}{(-\infty, r_U] \subseteq \texttt{real}} \tag{7.17g}$$

$$\frac{}{\texttt{int} \subseteq \texttt{real}} \tag{7.17h}$$

$$\frac{}{\{\texttt{true}\} \subseteq \texttt{bool}} \tag{7.17i}$$

$$\frac{}{\{\texttt{false}\} \subseteq \texttt{bool}} \tag{7.17j}$$

$$\frac{}{\texttt{bool} \subseteq \texttt{bool}} \tag{7.17k}$$

$$\frac{\rho \subseteq \tau}{(i : \sigma \to \rho) \subseteq (i : \sigma \to \tau)} \tag{7.17l}$$

Type declarations are available in a few places in the language's syntax, but only some are replaced with refined type declarations. Our need for refined types is to restrict the values a variable can take when solving a program. Propositions $\exists x : \tau \centerdot c$ are replaced with $\exists x : \rho \centerdot c$ and programs $\delta_{x_1:\tau_1,\ldots,x_m:\tau_m}\ \{e \mid c\}$ with $\delta_{x_1:\rho_1,\ldots,x_m:\rho_m}\ \{e \mid c\}$, but the type in $e : \tau$ and case expressions remain as is. Changing $e : \tau$ to $e : \rho$ would be more difficult because it requires checking that e is of a refined type ρ. This would affect our notion of well-formed programs, but that is not our intention for them.

As in the theory of unindexed programs, we define the judgement $x : \rho \simeq c$, meaning the type declaration $x : \rho$ corresponds to a proposition c. Its definition is

$$\frac{x : \rho \simeq_* c}{x : \rho \simeq c} \tag{7.18}$$

All the work is done by a helper judgement $e : \rho \simeq_* c$. This judgement is similar but the recursion allows general expressions, not just a variable. The definition of $\simeq_*$ is inductive on the form of ρ,

$$\frac{}{e : \langle r_L, r_U\rangle \backsimeq_* r_L \le e \wedge e \le r_U} \quad (7.19a) \qquad \frac{}{e : (-\infty, r_U] \backsimeq_* e \le r_U} \quad (7.19g)$$

$$\frac{}{e : \langle r_L, \infty) \backsimeq_* r_L \le e} \quad (7.19b) \qquad \frac{}{e : \texttt{int} \backsimeq_* \texttt{T}} \quad (7.19h)$$

$$\frac{}{e : (-\infty, r_U\rangle \backsimeq_* e \le r_U} \quad (7.19c) \qquad \frac{}{e : \{\texttt{true}\} \backsimeq_* \texttt{isTrue}\ e} \quad (7.19i)$$

$$\frac{}{e : \texttt{real} \backsimeq_* \texttt{T}} \quad (7.19d) \qquad \frac{}{e : \{\texttt{false}\} \backsimeq_* \texttt{isTrue}\ (\texttt{not}\, e)} \quad (7.19j)$$

$$\frac{}{e : [r_L, r_U] \backsimeq_* r_L \le e \wedge e \le r_U} \quad (7.19e) \qquad \frac{}{e : \texttt{bool} \backsimeq_* \texttt{T}} \quad (7.19k)$$

$$\frac{}{e : [r_L, \infty) \backsimeq_* r_L \le e} \quad (7.19f) \qquad \frac{(e\, i) : \rho \simeq_* c}{e : (i : \sigma \to \rho) \simeq_* \bigwedge_{i:\sigma} c} \quad (7.19l)$$

The final rule requires explanation. Consider a variable declaration $x : (i : \sigma \to [1, 10])$. This is a variable indexed by the set σ. According to the judgement, this is equivalent to stating the bounding proposition $\bigwedge_{i:\sigma} 1 \le (x\, i) \wedge (x\, i) \le 10$, which means x applied to i is bounded between 1 and 10 for every i. This is exactly the intended interpretation of the declaration $x : (i : \sigma \to [1, 10])$.

As for unindexed programs, we define a refined context

$$\Upsilon ::= \emptyset \mid \Upsilon, x : \rho \quad (7.20)$$

Finally, let ρ BOUNDED mean ρ represents a bounded domain. Its definition is

$$\frac{}{\langle r_L, r_U\rangle \ \text{BOUNDED}} \quad (7.21a) \qquad \frac{}{\{\texttt{false}\} \ \text{BOUNDED}} \quad (7.21d)$$

$$\frac{}{[r_L, r_U] \ \text{BOUNDED}} \quad (7.21b) \qquad \frac{}{\texttt{bool} \ \text{BOUNDED}} \quad (7.21e)$$

$$\frac{}{\{\texttt{true}\} \ \text{BOUNDED}} \quad (7.21c) \qquad \frac{\rho \ \text{BOUNDED}}{i : \sigma \to \rho \ \text{BOUNDED}} \quad (7.21f)$$

7.4 Semantics

In Section 4.5, we defined the semantics of unindexed programs. The syntax of expressions and propositions has been enriched in this chapter. So we need to define expression evaluation and propositional truth. The complete definitions are provided but we discuss only the extensions.

7.4.1 Evaluation of Expression

Let e CANONICAL be defined by the rules

$$\frac{}{r \text{ CANONICAL}} \quad (7.22a) \qquad \frac{}{\texttt{false} \text{ CANONICAL}} \quad (7.22c)$$

$$\frac{}{\texttt{true} \text{ CANONICAL}} \quad (7.22b) \qquad \frac{\lambda i : \sigma \,\texttt{.}\, e \text{ CLOSED}}{\lambda i : \sigma \,\texttt{.}\, e \text{ CANONICAL}} \quad (7.22d)$$

Functions are considered canonical simply if they are closed. For example, $\lambda i : \sigma \,\texttt{.}\, 1 + 2$ is canonical even though the body of the function could be reduced further. In general, the body would probably involve the variable i. For example, $\lambda i : \sigma \,\texttt{.}\, (x\, i) + 2$, and this cannot be reduced further. An important consequence of this is that canonical expressions that are syntactically distinct could nonetheless be equivalent, e.g. the functions $\lambda i : \sigma \,\texttt{.}\, (x\, i) + 2$ and $\lambda i : \sigma \,\texttt{.}\, (x\, i) + 1 + 1$. This contrasts with the canonical forms of unindexed programs.

Recall canonical expressions are also called values. Just as r is a value of type `real`, the function $\lambda i : \sigma \,\texttt{.}\, e$ is a value of type $i : \sigma \to \tau$. It is a function constant. The language allows variables of type `real` and also function variables. Let v denote an expression e such that e CANONICAL.

Given e CLOSED, let $e \searrow v$ mean e evaluates to v. Its definition is inductive on the form of e,

1. $r \searrow r$
2. $\texttt{true} \searrow \texttt{true}$
3. $\texttt{false} \searrow \texttt{false}$
4. $-e \searrow \begin{cases} r \text{ if } e \searrow -r \\ -r \text{ if } e \searrow r \end{cases}$
5. $e_1 \,\texttt{op}\, e_2 \searrow r$ if
   $e_1 \searrow r_1$ and $e_2 \searrow r_2$, where $r_1 \,\mathrm{op}\, r_2 = r$, for $\texttt{op} \in \{+, -, *\}$
6. $\texttt{not}\, e \searrow \begin{cases} \texttt{true} \text{ if } e \searrow \texttt{false} \\ \texttt{false} \text{ if } e \searrow \texttt{true} \end{cases}$
7. $e_1 \,\texttt{or}\, e_2 \searrow \begin{cases} \texttt{true} \text{ if either } e_1 \searrow \texttt{true} \text{ or } e_2 \searrow \texttt{true} \\ \texttt{false} \text{ otherwise} \end{cases}$
8. $e_1 \,\texttt{and}\, e_2 \searrow \begin{cases} \texttt{true} \text{ if } e_1 \searrow \texttt{true} \text{ and } e_2 \searrow \texttt{true} \\ \texttt{false} \text{ otherwise} \end{cases}$
9. $\sum_{i:\sigma} e \searrow r$ if
   $\{\{\eta_j / i\}\, e \searrow r_j\}_{j=1}^{m}$ and $r_1 + \cdots + r_m = r$, where $S_\sigma^q = \{\eta_1, \ldots, \eta_m\}$
10. $\texttt{case}_{i:\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow e_j\}_{j=1}^{m} \searrow e$ if
    $\varepsilon \searrow l_k$ and $e_k \searrow e$, where $k \in \{l_1, \ldots, l_m\}$
11. $\texttt{case}_{i:\tau}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow e_j\}_{j=1}^{m} \searrow e$ if
    $\varepsilon \searrow k$ and $e_k \searrow e$, where $k \in \{k_1, \ldots, k_m\}$

12. $\lambda i : \sigma \,.\, e \searrow \lambda i : \sigma \,.\, e$

13. $e\,\varepsilon \searrow e''$ if

    $e \searrow \lambda i : \sigma \,.\, e'$ and $\{\varepsilon/i\}\, e' \searrow e''$

14. $e : \tau \searrow e'$ if

    $e \searrow e'$

Rules covering the extensions of this chapter begin with rule 9.

Indexed summation requires first enumerating the values of the index set over which the summation is declared. Each of the indices is substituted into the sum's body, and evaluated. The sum of each of these terms is the overall solution.

Evaluation of case expressions is familiar from the indexing language. The case object must evaluate to one of the handles, and evaluation of the corresponding branch is the final answer.

The function $\lambda i : \sigma \,.\, e$ is already canonical (if it is closed which is a precondition of evaluation). Evaluation of an application form $e\,\varepsilon$ is done by first evaluating the expression e. Since we are assuming that expressions have been type checked, e must evaluate to a function $\lambda i : \sigma . e'$ because there is no other canonical form that could be applied to an index expression. Now, we have $(\lambda i : \sigma \,.\, e')\ \varepsilon$, application of a function constant to ε. Application is done by substituting the argument into the function body, i.e. $\{\varepsilon/i\}\, \varepsilon'$, which is called β-reduction. The resulting expression is guaranteed to be closed but it must be evaluated to obtain the final canonical answer.

Evaluation of an ascribed expression simply disregards the ascription.

7.4.2 Truth of Proposition

In the unindexed logic, all well-formed propositions could be treated as objects to be proven true or false. In the current logic, only propositions of type `Prop` can be interpreted this way. It is invalid to ask if propositions of type $i : \sigma \to \zeta$ are true. These are in the syntactic category we call "propositions", but they are not propositions in the usual sense.

Certain propositional forms, the case's, applications, and propositional type ascriptions, can be of any type. It is necessary to have propositions in a form such that their type is determinate from their syntactic structure. Let c CANONICAL mean c is a proposition that cannot be reduced further. Given a closed c, the definition of c CANONICAL is

$$\frac{}{\texttt{T}\ \text{CANONICAL}} \quad (7.23a) \qquad \frac{}{c_1 \wedge c_2\ \text{CANONICAL}} \quad (7.23g)$$

$$\frac{}{\texttt{F}\ \text{CANONICAL}} \quad (7.23b) \qquad \frac{}{\exists x : \rho \centerdot c\ \text{CANONICAL}} \quad (7.23h)$$

$$\frac{}{\texttt{isTrue}\ e\ \text{CANONICAL}} \quad (7.23c) \qquad \frac{}{\bigvee_{i:\sigma} c\ \text{CANONICAL}} \quad (7.23i)$$

$$\frac{}{e_1 = e_2\ \text{CANONICAL}} \quad (7.23d) \qquad \frac{}{\bigwedge_{i:\sigma} c\ \text{CANONICAL}} \quad (7.23j)$$

$$\frac{}{e_1 \leq e_2\ \text{CANONICAL}} \quad (7.23e) \qquad \frac{}{\lambda i : \sigma \centerdot c\ \text{CANONICAL}} \quad (7.23k)$$

$$\frac{}{c_1 \vee c_2\ \text{CANONICAL}} \quad (7.23f)$$

All this definition does is exclude case propositions, applications $c\,\varepsilon$, and ascriptions $c : \zeta$. When closed, these can all be reduced. In binary disjunctions and conjunctions we could require each of c_1 and c_2 to be canonical, but this is not needed for our purposes.

Given c_1 CLOSED and c_2 CANONICAL, let $c_1 \searrow c_2$ mean c_1 evaluates to c_2. Its definition is inductive on the form of c_1,

1. $\texttt{T} \searrow \texttt{T}$
2. $\texttt{F} \searrow \texttt{F}$
3. $\texttt{isTrue}\ e \searrow \texttt{isTrue}\ e$
4. $e_1\ \texttt{op}\ e_2 \searrow e_1\ \texttt{op}\ e_2$, for $\texttt{op} \in \{=, \leq\}$
5. $c_1\ \texttt{op}\ c_2 \searrow c_1\ \texttt{op}\ c_2$, for $\texttt{op} \in \{\vee, \wedge\}$
6. $\exists x : \rho \centerdot c \searrow \exists x : \rho \centerdot c$
7. $\underset{i:\sigma}{\texttt{OP}}\, c \searrow \underset{i:\sigma}{\texttt{OP}}\, c$, for $\texttt{OP} \in \{\bigvee, \bigwedge\}$
8. $\texttt{case}_{i.\zeta}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow c_j\}_{j=1}^{m} \searrow c$ if
   $\varepsilon \searrow l_k$ and $c_k \searrow c$, where $k \in \{l_1, \ldots, l_m\}$
9. $\texttt{case}_{i.\zeta}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow c_j\}_{j=1}^{m} \searrow c$ if
   $\varepsilon \searrow k$ and $c_k \searrow c$, where $k \in \{k_1, \ldots, k_m\}$
10. $\lambda i : \sigma \centerdot c \searrow \lambda i : \sigma \centerdot c$
11. $c\,\varepsilon \searrow c''$ if
    $c \searrow \lambda i : \sigma \centerdot c'$ and $\{\varepsilon/i\}\, c' \searrow c''$
12. $c : \zeta \searrow c'$ if
    $c \searrow c'$

Evaluation only prepares propositions for proof. It does not provide any information about whether a proposition is true. Truth of a proposition is given by the judgement c TRUE, which is defined only for propositions c satisfying c CLOSED and $\emptyset \vdash_\emptyset c : \texttt{Prop}$. Its definition is

1. $\texttt{T}$ TRUE
2. $(\texttt{isTrue}\ e)$ TRUE if

   $e \searrow \texttt{true}$
3. $(e_1\ \texttt{op}\ e_2)$ TRUE if

   $e_1 \searrow r_1$ and $e_2 \searrow r_2$ and $r_1\ \mathrm{op}\ r_2$, for $\texttt{op} \in \{=, \leq\}$
4. $(c_1 \vee c_2)$ TRUE if

   either c_1 TRUE or c_2 TRUE
5. $(c_1 \wedge c_2)$ TRUE if

   both c_1 TRUE and c_2 TRUE
6. $(\exists x : \rho\,\texttt{.}\,c)$ TRUE if

   $\{v/x\}\,c$ TRUE for some $v \in \rho$
7. $\bigvee_{i:\sigma} c$ TRUE if

   $\{\eta/i\}\,c$ TRUE for some $\eta \in S_\sigma = \{\eta_1, \ldots, \eta_m\}$
8. $\bigwedge_{i:\sigma} c$ TRUE if

   $\{\eta/i\}\,c$ TRUE for all $\eta \in S_\sigma = \{\eta_1, \ldots, \eta_m\}$
9. $\left(\texttt{case}_{i.\zeta}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow c_j\}_{j=1}^m\right)$ TRUE if

   $\left(\texttt{case}_{i.\zeta}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow c_j\}_{j=1}^m\right) \searrow c$ and c TRUE
10. $\left(\texttt{case}_{i.\zeta}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow c_j\}_{j=1}^m\right)$ TRUE if

    $\left(\texttt{case}_{i.\zeta}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow c_j\}_{j=1}^m\right) \searrow c$ and c TRUE
11. $c\,\varepsilon$ TRUE if

    $c\,\varepsilon \searrow c'$ and c' TRUE
12. $c : \zeta$ TRUE if

    $c : \zeta \searrow c'$ and c' TRUE

Proving that $\exists x : \rho\texttt{.}c$ is true requires proving that c is true for a particular value v, called the witness, of x. We explained in Section 4.5 how this constructive view of propositional truth coincides to current practice in mathematical programming.

Indexed disjunctive propositions are a generalization of binary disjunction and a special case of existential propositions. The proposition $\bigvee_{i:\sigma} c$ is declared true if c can be proven true for a witness η, a specific value of i.

Truth of $\bigwedge_{i:\sigma} c$ follows from the truth of $\{\eta/i\}\, c$ for every value η of i. This is directly a generalization of binary conjunction.

Remaining forms are not canonical. Their truth is determined by evaluating them to a canonical form and proving the truth of that form.

7.4.3 Solution of Program

Let $p \twoheadrightarrow r_{\text{option}}$ mean program p has the solution r_{option}. The outer syntactic form of a program p is identical to that of unindexed programs. Correspondingly, the definition of the solution to a program is identical to that in Section 4.5.3 on page 61. We do not repeat it. Of course, that definition's references to expression evaluation and propositional truth now refer to those given above.

7.4.4 Open Forms

The meaning of an open construct can be explained only with respect to an assignment of values for all its free variables. As with unindexed programs let

$$\Psi ::= \emptyset \mid \Psi, x = v \tag{7.24}$$

be a valuation for program variables. On page 95, the valuation Φ of index variables was similarly defined.

The judgement $\Psi \vdash_\Phi e \searrow v$ means e, not necessarily closed, evaluates to v under the valuations Ψ and Φ. Its definition is inductive on the forms of Ψ and Φ,

1. $\emptyset \vdash_\emptyset e \searrow v$ if

   $e \searrow v$

2. $\emptyset \vdash_{\Phi, i=\eta} e \searrow v$ if

   $\emptyset \vdash_\Phi \{\eta/i\}\, e \searrow v$

3. $\Psi, x = v \vdash_\Phi e \searrow v$ if

   $\Psi \vdash_\Phi \{v/x\}\, e \searrow v$

Similarly, let $\Psi \vdash_\Phi c$ TRUE mean proposition c is true under the given valuations. Its definition is analogous to expression evaluation,

1. $\emptyset \vdash_\emptyset c$ TRUE if

   c TRUE

2. $\emptyset \vdash_{\Phi, i=\eta} c$ TRUE if

   $\emptyset \vdash_\Phi \{\eta/i\}\, c$ TRUE

3. $\Psi, x = v \vdash_\Phi c$ TRUE if

   $\Psi \vdash_\Phi \{v/x\}\, c$ TRUE

A well-formed program must be closed. So we do not provide a corresponding judgement for the solution of an open program.

7.5 Results

The overall goal of this chapter was to combine the unindexed programs of Chapter 4 and the indexing logic of Chapter 6 into a theory of indexed programs. The syntax of the programming language is now rich enough to enable modeling real systems. An application is provided in Chapter 9. Here, we give smaller examples to demonstrate the new features.

Example 7.1 The following program is a simple use of indices,

```
1 var x : [{'a','b','c'}] -> real
2
3 min 0.0 subject_to
4   x['d'] >= 0.0
```

A real variable indexed by the set {'a','b','c'} is introduced. Formally, it is a function variable and can be applied to an index expression. In the constraint it is applied to the expression 'd'.

We check if the program satisfies p MP. It does not, and the following are (some of) the error messages printed,

```
ERROR: expri not member of given typei
  expri at 4.5-4.7: 'd'
  typei at 1.10-1.22: {'a','b','c'}

MSG: unable to synthesize type, previous messages should explain why
  contexti:
  context: x:[{'a','b','c'}] -> real
  expr at 4.3-4.8: x['d']

MSG: type analysis failed, previous messages should explain why
  contexti:
  context: x:[{'a','b','c'}] -> real
  expr at 4.3-4.8: x['d']
  type: real
```

The first message states that 'd' is not in the set {'a','b','c'}. The judgement producing this error is the canonical type checker on index expressions, $\vdash^c$'d':{'a','b','c'}. The next message states how this judgement was reached. It says that an attempt was being made to synthesize the type of expression x['d']. Finally, the fourth message states that the synthesis algorithm was called in attempt to check that this expression is of type real. Clearly, the variable has been applied to an invalid index expression.

Example 7.2 The constraint

$$w = \min(x_1, \ldots, x_m) + 4.0$$

is not expressible in our language directly and is not considered an MP constraint. However, the minimization function can be represented with a combination of conjunctive and disjunctive constraints. The constraint in the following program is equivalent to the above.

```
var w : <10.0, 90.0>
var x : [{1,...,10}] -> <5.0, 75.0>

min w subject_to
  (CONJ i:{1,...,10} . w <= x[i] + 4.0),
  (DISJ i:{1,...,10} . w >= x[i] + 4.0)
```

We check the judgement p MP, and it is satisfied.

Chapter 8

Compiling Indexed Mathematical Programs

Chapter 5 defined a method for transforming unindexed mathematical programs to pure mixed-integer programs. We now define an analogous procedure for indexed programs.

One possibility is to first eliminate the indices, and then use the compiler for unindexed programs. The proposition $\bigwedge_{i:\{A,B\}} (x\,(i) \geq 0)$ would be expanded to $(x\,(A) \geq 0) \wedge (x\,(B) \geq 0)$, and the applications $x\,(A)$ and $x\,(B)$ would be replaced with variables x_A and x_B. This is the current state-of-the-art. Indices are mere notational conveniences, and they must be eliminated prior to sending models to algorithms, which do not accommodate them.

It is now possible to do better because indexed constructs are first class entities in our theory. Retaining indices complicates the compiler definition. Conversion to conjunctive normal form (CNF) is considered an established procedure, and our discussion of CNF for unindexed programs, beginning on page 70, was largely tutorial. In contrast, for the enhanced syntax now being considered, it is not even clear what a conjunctive normal form is. We will define it and provide a procedure for converting to this form.

After this, the sub-language of indexed MIP is defined, and a compiler from indexed MP to indexed MIP is provided.

8.1 Application Normal Form

It is difficult to define certain properties such as CNF and linearity in the presence of functions. For example, let f be the function

$$\lambda i : \sigma \,\texttt{.case}_{i,\tau}\; i \;\texttt{of}\; \text{`A'} \Rightarrow x \mid \text{`B'} \Rightarrow (x \,\texttt{and}\, z) \;\texttt{or}\; y \tag{8.1}$$

Then we ask whether $f\,\varepsilon$ is in conjunctive normal form? If ε is 'A', then we want to say yes, but if it is 'B', then no. If it is an open expression, it is not clear what the answer should be.

The general issue is that an expression's CNF status seems not expressible in terms of its constituent parts. Rather, an application's CNF status requires evaluating the application,

and considering whether the result is in CNF.

In this section, we define normalized applications, which have lambda expressions eliminated, and then provide a method for putting any application into this form. The form we define is closely related to the concept of head normalized form, which arises in studies of the λ-calculus (see e.g. Barendregt, 1984).

8.1.1 Definition of ANF

Let *application form* refer to the expression form $e\,\varepsilon$. The *head position* of $e\,\varepsilon$ is e, and $\lambda i : \sigma \,.\, e$ is called a *lambda expression*.

Remark 8.1. *The head position of a well-formed application must be one of the following forms:*

- *lone variable:* x
- *case expression:* $\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow e_j\}_{j=1}^{m}$, $\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow e_j\}_{j=1}^{m}$
- *lambda expression:* $\lambda i : \sigma \,.\, e$
- *application:* $e\,\varepsilon$
- *ascription:* $e : \tau$

Proof. Easily seen by inspecting the definition of the typing judgement $\Gamma \vdash_\Delta e : \tau$. □

Certain head positions are such that the application cannot be reduced further. We call such an application an application normal form (ANF). Formally, let e ANF be defined by the rules

$$e^{\text{ANF}} ::= x\,\varepsilon \mid e^{\text{ANF}}\,\varepsilon \tag{8.2}$$

For example, the application $(x\,\varepsilon)\,\varepsilon'$ is in ANF. Since x is a variable whose value is unknown, there is no way to reduce the application any further. On the other hand, $((\lambda i : \sigma \,.\, e')\ \varepsilon)\ \varepsilon'$ could be reduced to $(\{\varepsilon/i\}\,e')\ \varepsilon'$, and this might be further reduced depending on whether $\{\varepsilon/i\}\,e'$ can be reduced.

The analogous judgement c ANF is provided for propositions. However, since there are no propositional variables, this relation is empty. This means it will always be possible to eliminate a propositional application $c\,\varepsilon$. In contrast, it might not be possible to eliminate the expression application $e\,\varepsilon$, but it can at least be put in ANF.

8.1.2 Transformation to ANF

Given an application form, we define a procedure for eliminating the application form, or at least converting it to an application normal form.

The overall procedure depends on a one-step reduction relation $e_1 \rightsquigarrow_{\text{hr}} e_2$. This is a method for eliminating the outermost application form if possible. It is possible whenever

the head position is not a normalized head. Its definition consists of a rule for each possible head position form except a variable,

$$\frac{}{(\lambda i : \sigma \centerdot e)\ \varepsilon \rightsquigarrow_{\mathrm{hr}} \{\varepsilon/i\}\, e}\ \beta\text{-reduction} \tag{8.3a}$$

$$\frac{e\,\varepsilon' \rightsquigarrow_{\mathrm{hr}} e'}{(e\,\varepsilon')\ \varepsilon \rightsquigarrow_{\mathrm{hr}} e'\,\varepsilon} \tag{8.3b}$$

$$\frac{}{\left(\mathtt{case}_{i.\tau}\ \varepsilon'\ \mathtt{of}\ \{l_j \Rightarrow e_j\}_{j=1}^{m}\right)\ \varepsilon \rightsquigarrow_{\mathrm{hr}} \mathtt{case}_{i.\tau}\ \varepsilon'\ \mathtt{of}\ \{l_j \Rightarrow e_j\,\varepsilon\}_{j=1}^{m}} \tag{8.3c}$$

$$\frac{}{\left(\mathtt{case}_{i.\tau}\ \varepsilon'\ \mathtt{of}\ \{k_j \Rightarrow e_j\}_{j=1}^{m}\right)\ \varepsilon \rightsquigarrow_{\mathrm{hr}} \mathtt{case}_{i.\tau}\ \varepsilon'\ \mathtt{of}\ \{k_j \Rightarrow e_j\,\varepsilon\}_{j=1}^{m}} \tag{8.3d}$$

$$\frac{}{(e : \tau)\ \varepsilon \rightsquigarrow_{\mathrm{hr}} e\,\varepsilon} \tag{8.3e}$$

In the first rule, a lambda expression is being applied. Reduction is by performing the application, called β-reduction. In the second rule, the head position is itself an application form. This rule just reduces the head position. If a case expression is being applied, each branch must itself be of a function type. It is equivalent to move the application into each branch. This leads to an expansion in program size because the applied argument is being replicated. Finally, the ascription in an ascribed expression can be disregarded.

Let $e_1 \not\rightsquigarrow_{\mathrm{hr}} e_2$ mean, for a given e_1, there does not exist an e_2 such that $e_1 \rightsquigarrow_{\mathrm{hr}} e_2$. Since there is no e_2, it is intuitive to abbreviate the notation to $e_1 \not\rightsquigarrow_{\mathrm{hr}}$, meaning e_1 does not head reduce.

As the name implies, one-step reduction produces expressions whose head positions might be further reduced. The judgement $e_1 \rightsquigarrow e_2$ relates e_1 to an expression e_2 that either is not an application form or satisfies e_2 ANF. Its definition is

$$\frac{e \not\rightsquigarrow_{\mathrm{hr}}}{e \rightsquigarrow e} \tag{8.4a}$$

$$\frac{e \rightsquigarrow_{\mathrm{hr}} e' \quad e' \rightsquigarrow e''}{e \rightsquigarrow e''} \tag{8.4b}$$

The first rule checks whether a one-step reduction is possible. If not, application normalization is complete. If reduction is possible, the second rule does so and recursively attempts further reduction.

Lemma 8.2. *If $e_1 \not\rightsquigarrow_{\mathrm{hr}}$, then either:*

1. *e_1 is not an application form, or*
2. *e_1 is an application form satisfying* ANF.

Proof. None of the rules defining $\rightsquigarrow_{\mathrm{hr}}$ are for non-application forms. So this is certainly one reason that e_1 might not reduce. The second property states that if it is an application form, the only reason it would not reduce is if e_1 ANF is true. This is proven by induction on the head position of an application form:

1. The head position of e_1 is a lone variable, i.e. e_1 is $x\,\varepsilon$. There are no rules for applications in this form. Thus, $e_1 \not\rightsquigarrow_{\text{hr}}$, and it is easily seen that e_1 ANF.

2. The head position of e_1 is a lambda expression, case expression, or ascription. The rules for these forms have no preconditions. Thus, $e_1 \not\rightsquigarrow_{\text{hr}}$ is never true, making the lemma trivially true.

3. The head position of e_1 is itself an application, i.e. e_1 is $(e_2\,\varepsilon')\ \varepsilon$. This form will not reduce if $(e_2\,\varepsilon')$ does not, as seen from the rule for the form $(e_2\,\varepsilon')\ \varepsilon$. By inductive hypothesis, $(e_2\,\varepsilon')$ is either not an application or satisfies ANF. Clearly, it is an application and so $(e_2\,\varepsilon')$ ANF must be true. Finally, from the definition of ANF, this implies $(e_2\,\varepsilon')\ \varepsilon$ ANF.

□

Theorem 8.3 (Correctness)**.** *If $e \rightsquigarrow e''$, then either:*

1. *e'' is not an application form, or*

2. *e'' is an application form satisfying* ANF*.*

Proof. Two rules define $\rightsquigarrow$. In the first, e'' is e; the expression is returned unmodified. The precondition requires $e \not\rightsquigarrow_{\text{hr}}$, and thus the result follows from Lemma 8.2. In the second rule, e'' is such that $e' \rightsquigarrow e''$. By inductive hypothesis, e'' is not an application form or satisfies ANF. □

Let $c_1 \rightsquigarrow c_2$ be a head normalization procedure for propositions. Its definition is precisely analogous to that for expressions. The above theorem can be made more specific. Consider a proposition c_1 that is an application form $c\,\varepsilon$. If $c_1 \rightsquigarrow c_2$, then c_2 is not an application form. This is simply because c ANF is empty.

8.2 Sub-Languages

8.2.1 Indexed Mixed-Integer Programs

The following restricted syntaxes define indexed mixed-integer programs (MIPs):

$$\tau^{\text{MIP}} ::= \texttt{real} \mid i : \sigma \to \tau^{\text{MIP}} \tag{8.5}$$

$$\begin{aligned}\rho^{\text{MIP}} ::= {} & \langle r_L, r_U \rangle \mid \langle r_L, \infty) \mid (-\infty, r_U \rangle \mid \texttt{real} \\ \mid {} & [r_L, r_U] \mid [r_L, \infty) \mid (-\infty, r_U] \mid \texttt{int} \\ \mid {} & i : \sigma \to \rho^{\text{MIP}}\end{aligned} \tag{8.6}$$

$$\begin{aligned}e^{\text{MIP}} ::= {} & x \mid r \\ \mid {} & -e^{\text{MIP}} \mid e_1^{\text{MIP}} + e_2^{\text{MIP}} \mid e_1^{\text{MIP}} - e_2^{\text{MIP}} \mid e_1^{\text{MIP}} * e_2^{\text{MIP}} \\ \mid {} & \sum_{i:\sigma} e^{\text{MIP}} \\ \mid {} & \texttt{case}_{i \bullet \tau^{\text{MIP}}}\ \varepsilon\ \texttt{of}\ \left\{l_j \Rightarrow e_j^{\text{MIP}}\right\}_{j=1}^{m} \mid \texttt{case}_{i \bullet \tau^{\text{MIP}}}\ \varepsilon\ \texttt{of}\ \left\{k_j \Rightarrow e_j^{\text{MIP}}\right\}_{j=1}^{m} \\ \mid {} & \lambda i : \sigma \,\texttt{.}\, e^{\text{MIP}} \mid e^{\text{MIP}}\ \varepsilon \\ \mid {} & e^{\text{MIP}} : \tau^{\text{MIP}}\end{aligned} \tag{8.7}$$

$$\begin{aligned}c^{\text{MIP}} ::= {} & \texttt{T} \mid \texttt{F} \\ \mid {} & e_1^{\text{MIP}} = e_2^{\text{MIP}} \mid e_1^{\text{MIP}} \leq e_2^{\text{MIP}} \\ \mid {} & c_1^{\text{MIP}} \wedge c_2^{\text{MIP}} \\ \mid {} & \exists x : \rho^{\text{MIP}} \,\texttt{.}\, c^{\text{MIP}} \\ \mid {} & \bigwedge_{i:\sigma} c^{\text{MIP}} \\ \mid {} & \texttt{case}_{i \bullet \zeta}\ \varepsilon\ \texttt{of}\ \left\{l_j \Rightarrow c_j^{\text{MIP}}\right\}_{j=1}^{m} \mid \texttt{case}_{i \bullet \zeta}\ \varepsilon\ \texttt{of}\ \left\{k_j \Rightarrow c_j^{\text{MIP}}\right\}_{j=1}^{m} \\ \mid {} & \lambda i : \sigma \,\texttt{.}\, c^{\text{MIP}} \mid c^{\text{MIP}}\ \varepsilon \\ \mid {} & c^{\text{MIP}} : \zeta\end{aligned} \tag{8.8}$$

$$\zeta^{\text{MIP}} ::= \zeta \tag{8.9}$$

$$p^{\text{MIP}} ::= \delta_{x_1 : \rho_1^{\text{MIP}}, \ldots, x_m : \rho_m^{\text{MIP}}} \left\{e^{\text{MIP}} \mid c^{\text{MIP}}\right\} \tag{8.10}$$

$$\Upsilon^{\text{MIP}} ::= \emptyset \mid \Upsilon^{\text{MIP}}, x : \rho^{\text{MIP}} \tag{8.11}$$

$$\Psi^{\text{MIP}} ::= \emptyset \mid \Psi^{\text{MIP}}, x = v^{\text{MIP}} \tag{8.12}$$

The judgement style, e.g. p MIP, will also be used. The basic restrictions are the elimination of the `bool` type, Boolean expressions, and binary and indexed disjunctions. There are no restrictions on propositional types.

8.2.2 Indexed Linearity

We now extend the concept of linearity, originally discussed on page 69, to indexed programs. Conceptually, a linear expression is a numerical expression, i.e. of type `real`, in which two variables are never multiplied. The concept of linearity does not apply to function types. Whether $\lambda i : \sigma \,.\, x\,(i) + 1.0$ is linear should never be asked. The expression $x\,(i) + 1.0$ by itself is considered linear. Although it contains a function variable, it is a variable applied to an index. The properties normally associated with linear systems are preserved under such application forms.

Let e be an expression that satisfies $\Gamma \vdash_\Delta e : \texttt{real}$ or $\Gamma \vdash_\Delta e : \texttt{bool}$ for some Γ and Δ. Given such an e, let e LINEAR mean e can be transformed to a linear numerical expression. Its definition is

$$\frac{}{x \text{ LINEAR}} \tag{8.13a}$$

$$\frac{}{r \text{ LINEAR}} \tag{8.13b}$$

$$\frac{}{\texttt{true} \text{ LINEAR}} \tag{8.13c}$$

$$\frac{}{\texttt{false} \text{ LINEAR}} \tag{8.13d}$$

$$\frac{e \text{ LINEAR}}{-e \text{ LINEAR}} \tag{8.13e}$$

$$\frac{e_1 \text{ LINEAR} \quad e_2 \text{ LINEAR}}{e_1 + e_2 \text{ LINEAR}} \tag{8.13f}$$

$$\frac{e_1 \text{ LINEAR} \quad e_2 \text{ LINEAR}}{e_1 - e_2 \text{ LINEAR}} \tag{8.13g}$$

$$\frac{}{e_1 * e_2 \text{ LINEAR}} \text{ if } FV\,(e_1 * e_2) = \emptyset \tag{8.13h}$$

$$\frac{e_2 \text{ LINEAR}}{e_1 * e_2 \text{ LINEAR}} \text{ if } FV\,(e_1) = \emptyset \tag{8.13i}$$

$$\frac{e_1 \text{ LINEAR}}{e_1 * e_2 \text{ LINEAR}} \text{ if } FV\,(e_2) = \emptyset \tag{8.13j}$$

$$\frac{}{\texttt{not}\, e \text{ LINEAR}} \tag{8.13k}$$

$$\frac{}{e_1 \,\texttt{or}\, e_2 \text{ LINEAR}} \tag{8.13l}$$

$$\frac{}{e_1 \,\texttt{and}\, e_2 \text{ LINEAR}} \tag{8.13m}$$

$$\frac{e \text{ LINEAR}}{\sum\limits_{i:\sigma} e \text{ LINEAR}} \tag{8.13n}$$

$$\frac{\{e_j \text{ LINEAR}\}_{j=1}^{m}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow e_j\}_{j=1}^{m} \text{ LINEAR}} \tag{8.13o}$$

$$\frac{\{e_j \text{ LINEAR}\}_{j=1}^{m}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow e_j\}_{j=1}^{m} \text{ LINEAR}} \tag{8.13p}$$

$$\frac{e\,\varepsilon \rightsquigarrow e' \quad e' \text{ ANF}}{e\,\varepsilon \text{ LINEAR}} \tag{8.13q}$$

$$\frac{e\,\varepsilon \rightsquigarrow e' \quad e' \neg\text{ANF} \quad e' \text{ LINEAR}}{e\,\varepsilon \text{ LINEAR}} \tag{8.13r}$$

$$\frac{e \text{ LINEAR}}{e : \tau \text{ LINEAR}} \tag{8.13s}$$

The interesting rules are the two for the application forms. Each of these applies $\rightsquigarrow$ to convert the application to an expression e'. Theorem 8.3 guarantees that either e' ANF is true or e' is not an application, which implies e' $\neg$ANF. Any expression e' satisfying e' ANF is declared linear. If e' is not an application, its linearity is determined by recursion.

Consider a proposition c that satisfies c MP and $\Gamma \vdash_\Delta c : \texttt{Prop}$ for some Γ and Δ. Then, let c LINEAR mean every expression e within c can be transformed into a linear numerical

expression. Its definition is

$$\frac{}{\texttt{T}\ \text{LINEAR}} \tag{8.14a}$$

$$\frac{}{\texttt{F}\ \text{LINEAR}} \tag{8.14b}$$

$$\frac{e\ \text{LINEAR}}{\texttt{isTrue}\ e\ \text{LINEAR}} \tag{8.14c}$$

$$\frac{e_1\ \text{LINEAR} \quad e_2\ \text{LINEAR}}{e_1 = e_2\ \text{LINEAR}} \tag{8.14d}$$

$$\frac{e_1\ \text{LINEAR} \quad e_2\ \text{LINEAR}}{e_1 \leq e_2\ \text{LINEAR}} \tag{8.14e}$$

$$\frac{c_1\ \text{LINEAR} \quad c_2\ \text{LINEAR}}{c_1 \vee c_2\ \text{LINEAR}} \tag{8.14f}$$

$$\frac{c_1\ \text{LINEAR} \quad c_2\ \text{LINEAR}}{c_1 \wedge c_2\ \text{LINEAR}} \tag{8.14g}$$

$$\frac{c\ \text{LINEAR}}{\exists x : \rho \centerdot c\ \text{LINEAR}} \tag{8.14h}$$

$$\frac{c\ \text{LINEAR}}{\bigvee_{i:\sigma} c\ \text{LINEAR}} \tag{8.14i}$$

$$\frac{c\ \text{LINEAR}}{\bigwedge_{i:\sigma} c\ \text{LINEAR}} \tag{8.14j}$$

$$\frac{\{c_j\ \text{LINEAR}\}_{j=1}^{m}}{\texttt{case}_{i.\zeta}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow c_j\}_{j=1}^{m}\ \text{LINEAR}} \tag{8.14k}$$

$$\frac{\{c_j\ \text{LINEAR}\}_{j=1}^{m}}{\texttt{case}_{i.\zeta}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow c_j\}_{j=1}^{m}\ \text{LINEAR}} \tag{8.14l}$$

$$\frac{c\,\varepsilon \rightsquigarrow c' \quad c'\ \text{LINEAR}}{c\,\varepsilon\ \text{LINEAR}} \tag{8.14m}$$

$$\frac{c\ \text{LINEAR}}{c : \zeta\ \text{LINEAR}} \tag{8.14n}$$

The rule for the application form normalizes the application. With propositions, this is guaranteed to produce a c' that is not an application form. The precondition declares the application linear if c' is. All other rules simply recurse on their nested expressions and propositions.

Finally, given a program p satisfying p MP, let p LINEAR be defined by the rule

$$\frac{e\ \text{LINEAR} \quad c\ \text{LINEAR}}{\delta_{x_1:\rho_1,\ldots,x_m:\rho_m}\ \{e \mid c\}\ \text{LINEAR}} \tag{8.15}$$

8.3 Indexed Conjunctive Normal Form

In Section 5.2, we presented a definition of CNF and a method for transforming to CNF for unindexed expressions. Defining conjunctive normal forms on indexed expressions is significantly more complex. Certain matters that were too obvious to state in the unindexed theory must now be addressed explicitly. Care must be taken to define CNF in the presence of case expressions and functions.

8.3.1 Definition of Indexed CNF

Let e LITERAL be defined by the rules

$$\frac{}{x \text{ LITERAL}} \tag{8.16a}$$

$$\frac{}{\texttt{true} \text{ LITERAL}} \tag{8.16b}$$

$$\frac{}{\texttt{false} \text{ LITERAL}} \tag{8.16c}$$

$$\frac{e \text{ LITERAL}}{\texttt{not}\, e \text{ LITERAL}} \tag{8.16d}$$

$$\frac{\{e_j \text{ LITERAL}\}_{j=1}^{m}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow e_j\}_{j=1}^{m} \text{ LITERAL}} \tag{8.16e}$$

$$\frac{\{e_j \text{ LITERAL}\}_{j=1}^{m}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow e_j\}_{j=1}^{m} \text{ LITERAL}} \tag{8.16f}$$

$$\frac{e\,\varepsilon \text{ ANF}}{e\,\varepsilon \text{ LITERAL}} \tag{8.16g}$$

$$\frac{e \text{ LITERAL}}{e : \tau \text{ LITERAL}} \tag{8.16h}$$

A case expression is a literal if all of its branches are. We also declare expressions in ANF to be literals. The justification for this is simply that it works. It leads to a definition that allows converting expressions in CNF to linear integer propositions.

Let e DLF be defined by the rules

$$\frac{e \text{ LITERAL}}{e \text{ DLF}} \tag{8.17a}$$

$$\frac{e_1 \text{ DLF} \quad e_2 \text{ DLF}}{e_1\, \texttt{or}\, e_2 \text{ DLF}} \tag{8.17b}$$

$$\frac{\{e_j \text{ DLF}\}_{j=1}^{m}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow e_j\}_{j=1}^{m} \text{ DLF}} \tag{8.17c}$$

$$\frac{\{e_j \text{ DLF}\}_{j=1}^{m}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow e_j\}_{j=1}^{m} \text{ DLF}} \tag{8.17d}$$

$$\frac{e \text{ DLF}}{e : \tau \text{ DLF}} \tag{8.17e}$$

Literal forms are also disjunctive literal forms. Binary and indexed disjunctions are in DLF if their disjuncts are. Case expressions are in DLF if all their branches are.

Finally, let e CNF be defined by the rules

$$\frac{e \text{ DLF}}{e \text{ CNF}} \tag{8.18a}$$

$$\frac{e_1 \text{ CNF} \quad e_2 \text{ CNF}}{e_1 \,\texttt{and}\, e_2 \text{ CNF}} \tag{8.18b}$$

$$\frac{\{e_j \text{ CNF}\}_{j=1}^{m}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow e_j\}_{j=1}^{m} \text{ CNF}} \tag{8.18c}$$

$$\frac{\{e_j \text{ CNF}\}_{j=1}^{m}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow e_j\}_{j=1}^{m} \text{ CNF}} \tag{8.18d}$$

$$\frac{e \text{ CNF}}{e : \tau \text{ CNF}} \tag{8.18e}$$

Conceptually, conjunctive normal forms are conjunctions of disjunctive literal forms.

Certain judgements on CNF expressions will depend on whether the expression is or is not a DLF expression. Let e CONJ$'$ be defined by the rule

$$\frac{e \text{ CNF} \quad e \neg\text{DLF}}{e \text{ CONJ}'} \tag{8.19}$$

CONJ$'$ expressions are CNF expressions that are not DLF. On page 71, we gave the corresponding definition for unindexed programs. The syntactic forms of the expressions satisfying that judgement were rather obvious. It is less clear in the indexed case.

Given e CNF, let e CONJ be a more direct definition of the same judgement. The definition is

$$\frac{}{e_1 \,\texttt{and}\, e_2 \text{ CONJ}} \tag{8.20a}$$

$$\frac{e_k \text{ CONJ}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow e_j\}_{j=1}^{m} \text{ CONJ}} \text{ for } k \in \{1, \ldots, m\} \tag{8.20b}$$

$$\frac{e_k \text{ CONJ}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow e_j\}_{j=1}^{m} \text{ CONJ}} \text{ for } k \in \{1, \ldots, m\} \tag{8.20c}$$

$$\frac{e \text{ CONJ}}{e : \tau \text{ CONJ}} \tag{8.20d}$$

The idea is that e CONJ is satisfied by expressions having occurrences of `and`.

Figure 8.1 shows how the various judgements are related, and the following theorems prove that the figure is drawn correctly.

Lemma 8.4. DLF *and* CONJ *are mutually exclusive. Precisely, if* e DLF *then* e $\neg$CONJ*, and if* e CONJ *then* e $\neg$DLF*.*

Proof. It is easy to see that CONJ expressions must contain an `and` expression within them, and that DLF expressions cannot. □

Definition (Partition)**.** Judgements J_j for $j \in \{1, \ldots, m\}$ partition some other judgement J if both

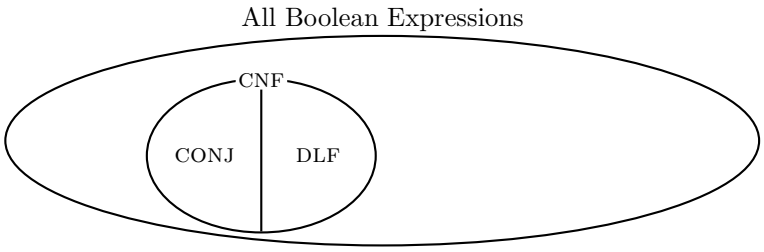


Figure 8.1: Venn diagram of various forms of Boolean expressions.

1. J_j implies J for every $j \in \{1, \ldots, m\}$, and
2. J implies J_k for exactly one $k \in \{1, \ldots, m\}$.

Theorem 8.5. *e DLF and e CONJ partition e CNF.*

Proof. e DLF implies e CNF. This is obvious from the first rule defining CNF. Also, e CONJ implies e CNF is obvious because that is explicitly stated as a precondition of CONJ.

Now we prove the second requirement, that e CNF implies either e DLF or e CONJ, but not both. It suffices to show that e CNF implies either e DLF or e CONJ. Mutual exclusivity was provided by Lemma 8.4. The proof is by induction on the definition of CNF.

1. The first rule defining CNF declares any DLF expression to be CNF. Expressions determined to be in CNF because of this rule thus immediately are seen to be in DLF.
2. Next, the expression $e_1 \texttt{ and } e_2$ can be in CNF if both e_1 and e_2 are. This expression is seen to be in CONJ by the first rule defining CONJ.
3. A case expression $\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow e_j\}_{j=1}^{m}$ is in CNF if all e_j are. By inductive hypothesis, each e_j is either in CONJ or in DLF, but not both. The possibilities can be split into two categories: either all e_j DLF, or not. If the former, then the whole case expression is in DLF. If the latter, then at least one particular e_j, say e_k, is in CONJ. Then, we can see that the third rule defining CONJ applies, and the whole case expression is in CONJ. Of course, the argument is identical for case expressions with integer handles.
4. An ascribed expression $e : \tau$ is in CNF if e is. The result follows immediately by IH.

□

Corollary 8.6. *CONJ′ and CONJ are equivalent.*

Proof. From the previous theorem, CONJ includes exactly those CNF expressions that are not DLF, which is explicitly the definition of CONJ′. □

8.3.2 Transforming to Indexed CNF

Consider e_1 such that $\Gamma \vdash_\Delta e : \texttt{bool}$ in some contexts Γ and Δ. Then, let $e_1 \curvearrowright e_2$ mean e_1 is transformed to e_2 such that e_2 CNF. The definition of $\curvearrowright$ is

$$\frac{e \text{ CNF}}{e \curvearrowright e} \tag{8.21a}$$

$$\frac{e \neg\text{CNF} \quad e_1 \curvearrowright_* e_2}{e_1 \curvearrowright e_2} \tag{8.21b}$$

which makes use of the auxiliary relation $\curvearrowright_*$.

Given $e_1 \neg$CNF, the judgement $e_1 \curvearrowright_* e_2$ provides e_2 such that e_2 CNF. Its definition is

$$\frac{\texttt{not}\, e_1 \overset{\text{not}}{\curvearrowright_*} e'}{\texttt{not}\, e_1 \curvearrowright_* e'} \tag{8.22a}$$

$$\frac{e_1\, \texttt{or}\, e_2 \overset{\text{or}}{\curvearrowright_*} e'}{e_1\, \texttt{or}\, e_2 \curvearrowright_* e'} \tag{8.22b}$$

$$\frac{\left\{e_j \curvearrowright e'_j\right\}_{j=1}^{2}}{e_1\, \texttt{and}\, e_2 \curvearrowright_* e'_1\, \texttt{and}\, e'_2} \tag{8.22c}$$

$$\frac{\left\{e_j \curvearrowright e'_j\right\}_{j=1}^{m}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \left\{l_j \Rightarrow e_j\right\}_{j=1}^{m} \curvearrowright_* \texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \left\{l_j \Rightarrow e'_j\right\}_{j=1}^{m}} \tag{8.22d}$$

$$\frac{\left\{e_j \curvearrowright e'_j\right\}_{j=1}^{m}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \left\{k_j \Rightarrow e_j\right\}_{j=1}^{m} \curvearrowright_* \texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \left\{k_j \Rightarrow e'_j\right\}_{j=1}^{m}} \tag{8.22e}$$

$$\frac{e\, \varepsilon \rightsquigarrow e' \quad e' \curvearrowright e''}{e\, \varepsilon \curvearrowright_* e''} \tag{8.22f}$$

$$\frac{e \curvearrowright_* e'}{e : \tau \curvearrowright_* e' : \tau} \tag{8.22g}$$

Special handling is required when e_1 is of the form $\texttt{not}\, e'$, or of the form $e'_1\, \texttt{or}\, e'_2$. These forms have been separated into judgements defined next. Conjunctive and case expressions are converted to CNF by converting their nested expressions. Applications are first head normalized. This will either produce an ANF, which is immediately CNF, or a non-application form, which will be inductively converted. The ascription on an ascribed expression is simply disregarded.

Now consider the conversion of $\texttt{not}\, e$, given by the judgement $\texttt{not}\, e \overset{\text{not}}{\curvearrowright_*} e'$, which depends further on the form of e in $\texttt{not}\, e$. Its definition is

$$\frac{e \curvearrowright e'}{\texttt{not}\,\texttt{not}\, e \overset{\text{not}}{\curvearrowright_*} e'} \tag{8.23a}$$

$$\frac{(\texttt{not}\, e_1)\ \texttt{and}\ (\texttt{not}\, e_2) \curvearrowright e'}{\texttt{not}\ (e_1\, \texttt{or}\, e_2) \overset{\text{not}}{\curvearrowright_*} e'} \tag{8.23b}$$

$$\frac{(\texttt{not}\, e_1)\ \texttt{or}\ (\texttt{not}\, e_2) \curvearrowright e'}{\texttt{not}\ (e_1\, \texttt{and}\, e_2) \overset{\text{not}}{\curvearrowright_*} e'} \tag{8.23c}$$

$$\frac{\texttt{case}_{i\bullet\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow \texttt{not}\, e_j\}_{j=1}^m \curvearrowright e'}{\texttt{not}\left(\texttt{case}_{i\bullet\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow e_j\}_{j=1}^m\right) \overset{\text{not}}{\curvearrowright_*} e'} \tag{8.23d}$$

$$\frac{\texttt{case}_{i\bullet\tau}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow \texttt{not}\, e_j\}_{j=1}^m \curvearrowright e'}{\texttt{not}\left(\texttt{case}_{i\bullet\tau}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow e_j\}_{j=1}^m\right) \overset{\text{not}}{\curvearrowright_*} e'} \tag{8.23e}$$

$$\frac{e\, \varepsilon \rightsquigarrow e' \quad \texttt{not}\, e' \curvearrowright e''}{\texttt{not}\, e\, \varepsilon \overset{\text{not}}{\curvearrowright_*} e''} \tag{8.23f}$$

$$\frac{\texttt{not}\, e \curvearrowright e'}{\texttt{not}\ (e : \tau) \overset{\text{not}}{\curvearrowright_*} e'} \tag{8.23g}$$

In the first rule, we are simply eliminating the double negation. The rules for negating `or` and `and` expressions employ DeMorgan's laws. Negation of case expressions is equivalent to negating each of the case's branches. Negation of an application form is handled by normalizing the application. Negation of an ascribed expression disregards the ascription.

Finally, an `or` expression is converted by the judgement $(e_1\, \texttt{or}\, e_2) \overset{\text{or}}{\curvearrowright_*} e'$. Its definition is

$$\frac{e_1\ \text{CONJ} \quad e_1\, \texttt{or}\, e_2 \overset{\text{or}-\text{conj}}{\curvearrowright_*} e'}{e_1\, \texttt{or}\, e_2 \overset{\text{or}}{\curvearrowright_*} e'} \tag{8.24a}$$

$$\frac{e_2\ \text{CONJ} \quad e_2\, \texttt{or}\, e_1 \overset{\text{or}-\text{conj}}{\curvearrowright_*} e'}{e_1\, \texttt{or}\, e_2 \overset{\text{or}}{\curvearrowright_*} e'} \tag{8.24b}$$

$$\frac{e_1\ \neg\text{CNF} \quad e_1 \curvearrowright e_1' \quad e_1'\, \texttt{or}\, e_2 \curvearrowright e'}{e_1\, \texttt{or}\, e_2 \overset{\text{or}}{\curvearrowright_*} e'} \tag{8.24c}$$

$$\frac{e_2\ \neg\text{CNF} \quad e_2 \curvearrowright e_2' \quad e_1\, \texttt{or}\, e_2' \curvearrowright e'}{e_1\, \texttt{or}\, e_2 \overset{\text{or}}{\curvearrowright_*} e'} \tag{8.24d}$$

When either disjunct is in CONJ, the judgement $\overset{\text{or}-\text{conj}}{\curvearrowright_*}$, defined next, is used. If either is not in CNF, it is first converted to CNF and the judgement called recursively. These are the only possibilities, as given by the following lemma.

Lemma 8.7. *Given* $(e_1\, \texttt{or}\, e_2)$ $\neg$CNF*, one of the following must be true:*

1. e_1 CONJ*, or* e_2 CONJ*, or*

2. e_1 $\neg$CNF, *or* e_2 $\neg$CNF.

Proof. Since $(e_1 \texttt{ or } e_2)$ $\neg$CNF, both e_1 DLF and e_2 DLF cannot be true. The result thus follows directly from Figure 8.1. □

Finally, given e_1 CONJ, let $(e_1 \texttt{ or } e_2) \overset{\text{or}-\text{conj}}{\curvearrowright_*} e'$ be the conversion of $e_1 \texttt{ or } e_2$. Its definition is dependent on the possible forms of e_1, which were given by rules (8.20). The definition is

$$\frac{(e_{11} \texttt{ or } e_2) \texttt{ and } (e_{12} \texttt{ or } e_2) \curvearrowright e'}{(e_{11} \texttt{ and } e_{12}) \texttt{ or } e_2 \overset{\text{or}-\text{conj}}{\curvearrowright_*} e'} \tag{8.25a}$$

$$\frac{\left(\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow e_j \texttt{ or } e_2\}_{j=1}^{m}\right) \curvearrowright e'}{\left(\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow e_j\}_{j=1}^{m}\right) \texttt{ or } e_2 \overset{\text{or}-\text{conj}}{\curvearrowright_*} e'} \tag{8.25b}$$

$$\frac{\left(\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow e_j \texttt{ or } e_2\}_{j=1}^{m}\right) \curvearrowright e'}{\left(\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow e_j\}_{j=1}^{m}\right) \texttt{ or } e_2 \overset{\text{or}-\text{conj}}{\curvearrowright_*} e'} \tag{8.25c}$$

$$\frac{e \texttt{ or } e_2 \overset{\text{or}-\text{conj}}{\curvearrowright_*} e'}{(e : \tau) \texttt{ or } e_2 \overset{\text{or}-\text{conj}}{\curvearrowright_*} e'} \tag{8.25d}$$

The first rule distributes disjunction over conjunction. The second two handle case expressions by replicating the disjunction in each branch.

8.4 Compiling Indexed MP to Indexed MIP

As discussed in Appendix A, the compilation of a mathematical program requires all disjuncts to be bounded. We enforce this by requiring all variables within a disjunct to have known bounds. As in the compilation of unindexed programs, let $\Upsilon \vdash c$ DISJVARSBOUNDED mean Υ contains bounds for all variables free in or existentially introduced within any of the disjuncts in c. The judgement is defined only for c such that $\Upsilon \vdash_\Delta c : \texttt{Prop}$ is true. Its

definition is

$$\frac{}{\Upsilon \vdash \texttt{T}\ \textsc{DisjVarsBounded}} \tag{8.26a}$$

$$\frac{}{\Upsilon \vdash \texttt{F}\ \textsc{DisjVarsBounded}} \tag{8.26b}$$

$$\frac{}{\Upsilon \vdash \texttt{isTrue}\ e\ \textsc{DisjVarsBounded}} \tag{8.26c}$$

$$\frac{}{\Upsilon \vdash e_1\ \texttt{op}\ e_2\ \textsc{DisjVarsBounded}}\ \text{where } \texttt{op} \in \{=, \leq\} \tag{8.26d}$$

$$\frac{\{\rho_j\ \textsc{bounded}\}_{j=1}^{m} \quad \{c_j\ \textsc{ExistVarsBounded}\}_{j=1}^{2}}{\Upsilon \vdash c_1 \vee c_2\ \textsc{DisjVarsBounded}}$$

$$\text{where } FV(c_1 \vee c_2) = \{x_1, \ldots, x_m\}, \text{ and } \Upsilon(x_j) = \rho_j \tag{8.26e}$$

$$\frac{\Upsilon \vdash c_1\ \textsc{DisjVarsBounded} \quad \Upsilon \vdash c_2\ \textsc{DisjVarsBounded}}{\Upsilon \vdash c_1 \wedge c_2\ \textsc{DisjVarsBounded}} \tag{8.26f}$$

$$\frac{\Upsilon, x : \rho \vdash c\ \textsc{DisjVarsBounded}}{\Upsilon \vdash \exists x : \rho\ \texttt{.}\ c\ \textsc{DisjVarsBounded}} \tag{8.26g}$$

$$\frac{\{\rho_j\ \textsc{bounded}\}_{j=1}^{m} \quad c\ \textsc{ExistVarsBounded}}{\Upsilon \vdash \bigvee_{i:\sigma} c\ \textsc{DisjVarsBounded}}$$

$$\text{where } FV(c) = \{x_1, \ldots, x_m\}, \text{ and } \Upsilon(x_j) = \rho_j \tag{8.26h}$$

$$\frac{\Upsilon \vdash c\ \textsc{DisjVarsBounded}}{\Upsilon \vdash \bigwedge_{i:\sigma} c\ \textsc{DisjVarsBounded}} \tag{8.26i}$$

$$\frac{\{\Upsilon \vdash c_j\ \textsc{DisjVarsBounded}\}_{j=1}^{m}}{\Upsilon \vdash \texttt{case}_{i,\zeta}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow c_j\}_{j=1}^{m}\ \textsc{DisjVarsBounded}} \tag{8.26j}$$

$$\frac{\{\Upsilon \vdash c_j\ \textsc{DisjVarsBounded}\}_{j=1}^{m}}{\Upsilon \vdash \texttt{case}_{i,\zeta}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow c_j\}_{j=1}^{m}\ \textsc{DisjVarsBounded}} \tag{8.26k}$$

$$\frac{c\,\varepsilon \rightsquigarrow c' \quad \Upsilon \vdash c'\ \textsc{DisjVarsBounded}}{\Upsilon \vdash c\,\varepsilon\ \textsc{DisjVarsBounded}} \tag{8.26l}$$

$$\frac{\Upsilon \vdash c\ \textsc{DisjVarsBounded}}{\Upsilon \vdash c : \zeta\ \textsc{DisjVarsBounded}} \tag{8.26m}$$

Indexed disjunction is treated like binary disjunction. All free variables in the disjunct must have known bounds in the given context. The judgement c ExistVarsBounded, defined next, checks if variables introduced within the disjunct have bounds. Conjunctive and case propositions inductively check their nested propositions. Applications are handled in the usual way; they are reduced to ANF, and then checked inductively.

Given c such that $\Upsilon \vdash_\Delta c : \texttt{Prop}$, let c ExistVarsBounded mean variables existentially

introduced within c are bounded. Its definition is

$$\frac{}{\texttt{T}\ \textsc{ExistVarsBounded}} \tag{8.27a}$$

$$\frac{}{\texttt{F}\ \textsc{ExistVarsBounded}} \tag{8.27b}$$

$$\frac{}{\texttt{isTrue}\ e\ \textsc{ExistVarsBounded}} \tag{8.27c}$$

$$\frac{}{e_1\ \texttt{op}\ e_2\ \textsc{ExistVarsBounded}}\ \text{where } \texttt{op} \in \{=, \leq\} \tag{8.27d}$$

$$\frac{\{c_j\ \textsc{ExistVarsBounded}\}_{j=1}^{2}}{c_1\ \texttt{op}\ c_2\ \textsc{ExistVarsBounded}}\ \text{where } \texttt{op} \in \{\vee, \wedge\} \tag{8.27e}$$

$$\frac{\rho\ \textsc{bounded} \quad c\ \textsc{ExistVarsBounded}}{\exists x : \rho\ \texttt{.}\ c\ \textsc{ExistVarsBounded}} \tag{8.27f}$$

$$\frac{c\ \textsc{ExistVarsBounded}}{\underset{i:\sigma}{\texttt{OP}}\ c\ \textsc{ExistVarsBounded}}\ \text{for } \texttt{OP} \in \{\bigvee, \bigwedge\} \tag{8.27g}$$

$$\frac{\{c_j\ \textsc{ExistVarsBounded}\}_{j=1}^{m}}{\texttt{case}_{i.\zeta}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow c_j\}_{j=1}^{m}\ \textsc{ExistVarsBounded}} \tag{8.27h}$$

$$\frac{\{c_j\ \textsc{ExistVarsBounded}\}_{j=1}^{m}}{\texttt{case}_{i.\zeta}\ \varepsilon\ \texttt{of}\ \{k_j \Rightarrow c_j\}_{j=1}^{m}\ \textsc{ExistVarsBounded}} \tag{8.27i}$$

$$\frac{c\,\varepsilon \rightsquigarrow c' \quad c'\ \textsc{ExistVarsBounded}}{c\,\varepsilon\ \textsc{ExistVarsBounded}} \tag{8.27j}$$

$$\frac{c\ \textsc{ExistVarsBounded}}{c : \zeta\ \textsc{ExistVarsBounded}} \tag{8.27k}$$

Finally, let p $\textsc{DisjVarsBounded}$ mean all variables in all disjunctions in program p have known bounds. Its definition is

$$\frac{x_1 : \rho_1, \ldots, x_m : \rho_m \vdash c\ \textsc{DisjVarsBounded}}{\delta_{x_1:\rho_1,\ldots,x_m:\rho_m}\ \{e \mid c\}\ \textsc{DisjVarsBounded}} \tag{8.28}$$

Our compiler is defined only for programs satisfying this requirement.

8.4.1 Type Compiler

Let $\tau \overset{\text{TYPE}}{\longmapsto} \tau^{\text{MIP}}$ be a type compiler. Its definition is

$$\frac{}{\texttt{real} \overset{\text{TYPE}}{\longmapsto} \texttt{real}} \tag{8.29}$$

$$\frac{}{\texttt{bool} \overset{\text{TYPE}}{\longmapsto} \texttt{real}} \tag{8.30}$$

$$\frac{\tau \overset{\text{TYPE}}{\longmapsto} \tau'}{(i : \sigma \to \tau) \overset{\text{TYPE}}{\longmapsto} (i : \sigma \to \tau')} \tag{8.31}$$

Let $\rho \overset{\text{RTYPE}}{\longmapsto} \rho^{\text{MIP}}$ be a refined type compiler,

$$\frac{}{\langle r_L, r_U\rangle \overset{\text{RTYPE}}{\longmapsto} \langle r_L, r_U\rangle} \tag{8.32a}$$

$$\frac{}{\langle r_L, \infty) \overset{\text{RTYPE}}{\longmapsto} \langle r_L, \infty)} \tag{8.32b}$$

$$\frac{}{(-\infty, r_U\rangle \overset{\text{RTYPE}}{\longmapsto} (-\infty, r_U\rangle} \tag{8.32c}$$

$$\frac{}{\texttt{real} \overset{\text{RTYPE}}{\longmapsto} \texttt{real}} \tag{8.32d}$$

$$\frac{}{[r_L, r_U] \overset{\text{RTYPE}}{\longmapsto} [r_L, r_U]} \tag{8.32e}$$

$$\frac{}{[r_L, \infty) \overset{\text{RTYPE}}{\longmapsto} [r_L, \infty)} \tag{8.32f}$$

$$\frac{}{(-\infty, r_U] \overset{\text{RTYPE}}{\longmapsto} (-\infty, r_U]} \tag{8.32g}$$

$$\frac{}{\texttt{int} \overset{\text{RTYPE}}{\longmapsto} \texttt{int}} \tag{8.32h}$$

$$\frac{}{\{\texttt{true}\} \overset{\text{RTYPE}}{\longmapsto} [1,1]} \tag{8.32i}$$

$$\frac{}{\{\texttt{false}\} \overset{\text{RTYPE}}{\longmapsto} [0,0]} \tag{8.32j}$$

$$\frac{}{\texttt{bool} \overset{\text{RTYPE}}{\longmapsto} [0,1]} \tag{8.32k}$$

$$\frac{\rho \overset{\text{RTYPE}}{\longmapsto} \rho'}{(i : \sigma \to \rho) \overset{\text{RTYPE}}{\longmapsto} (i : \sigma \to \rho')} \tag{8.32l}$$

8.4.2 Expression Compiler

Only expressions in CONJ and DLF need to be compiled.

8.4.2.1 DLF Expression Compiler

Let $e^{\text{DLF}} \overset{\text{DLF}}{\longmapsto} e^{\text{MIP}}$ be a judgement converting DLF expressions to MIP expressions. Its definition is

$$\frac{}{x \overset{\text{DLF}}{\longmapsto} x} \tag{8.33a}$$

$$\frac{}{\texttt{true} \overset{\text{DLF}}{\longmapsto} 1} \tag{8.33b}$$

$$\frac{}{\texttt{false} \overset{\text{DLF}}{\longmapsto} 0} \tag{8.33c}$$

$$\frac{e \overset{\text{DLF}}{\longmapsto} e'}{\texttt{not}\, e \overset{\text{DLF}}{\longmapsto} 1 - e'} \tag{8.33d}$$

$$\frac{e_1 \overset{\text{DLF}}{\longmapsto} e_1' \quad e_2 \overset{\text{DLF}}{\longmapsto} e_2'}{e_1 \,\texttt{or}\, e_2 \overset{\text{DLF}}{\longmapsto} e_1' + e_2'} \tag{8.33e}$$

$$\frac{\tau \overset{\text{TYPE}}{\longmapsto} \tau' \quad \left\{e_j \overset{\text{DLF}}{\longmapsto} e_j'\right\}_{j=1}^{m}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \left\{l_j \Rightarrow e_j\right\}_{j=1}^{m} \overset{\text{DLF}}{\longmapsto} \texttt{case}_{i.\tau'}\ \varepsilon\ \texttt{of}\ \left\{l_j \Rightarrow e_j'\right\}_{j=1}^{m}} \tag{8.33f}$$

$$\frac{\tau \overset{\text{TYPE}}{\longmapsto} \tau' \quad \left\{e_j \overset{\text{DLF}}{\longmapsto} e_j'\right\}_{j=1}^{m}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \left\{k_j \Rightarrow e_j\right\}_{j=1}^{m} \overset{\text{DLF}}{\longmapsto} \texttt{case}_{i.\tau'}\ \varepsilon\ \texttt{of}\ \left\{k_j \Rightarrow e_j'\right\}_{j=1}^{m}} \tag{8.33g}$$

$$\frac{}{e\,\varepsilon \overset{\text{DLF}}{\longmapsto} e\,\varepsilon} \tag{8.33h}$$

$$\frac{e \overset{\text{DLF}}{\longmapsto} e'}{e : \tau \overset{\text{DLF}}{\longmapsto} e'} \tag{8.33i}$$

8.4.2.2 CONJ Expression Compiler

Let $e^{\text{CONJ}} \overset{\text{CONJ}}{\longmapsto} c^{\text{MIP}}$ be a judgement converting CONJ expressions to MIP propositions. Its definition is

$$\frac{\left\{\emptyset \vdash \texttt{isTrue}\ e_j \overset{\text{PROP}}{\longmapsto} c_j\right\}_{j=1}^{2}}{e_1 \,\texttt{and}\, e_2 \overset{\text{CONJ}}{\longmapsto} c_1 \wedge c_2} \tag{8.34a}$$

$$\frac{\left\{\emptyset \vdash \texttt{isTrue}\ e_j \overset{\text{PROP}}{\longmapsto} c_j\right\}_{j=1}^{m}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \left\{l_j \Rightarrow e_j\right\}_{j=1}^{m} \overset{\text{CONJ}}{\longmapsto} \texttt{case}_{i.\texttt{Prop}}\ \varepsilon\ \texttt{of}\ \left\{l_j \Rightarrow c_j\right\}_{j=1}^{m}} \tag{8.34b}$$

$$\frac{\left\{\emptyset \vdash \texttt{isTrue}\ e_j \overset{\text{PROP}}{\longmapsto} c_j\right\}_{j=1}^{m}}{\texttt{case}_{i.\tau}\ \varepsilon\ \texttt{of}\ \left\{k_j \Rightarrow e_j\right\}_{j=1}^{m} \overset{\text{CONJ}}{\longmapsto} \texttt{case}_{i.\texttt{Prop}}\ \varepsilon\ \texttt{of}\ \left\{k_j \Rightarrow c_j\right\}_{j=1}^{m}} \tag{8.34c}$$

$$\frac{\emptyset \vdash \texttt{isTrue}\ e \overset{\text{PROP}}{\longmapsto} c}{e : \texttt{bool} \overset{\text{CONJ}}{\longmapsto} c : \texttt{Prop}} \tag{8.34d}$$

8.4.3 Proposition Compiler

Let $\Upsilon \vdash c \overset{\text{PROP}}{\longmapsto} c^{\text{MIP}}$ be a proposition compiler.

$$\frac{}{\Upsilon \vdash \mathtt{T} \overset{\text{PROP}}{\longmapsto} \mathtt{T}} \tag{8.35a}$$

$$\frac{}{\Upsilon \vdash \mathtt{F} \overset{\text{PROP}}{\longmapsto} \mathtt{F}} \tag{8.35b}$$

$$\frac{e \curvearrowright e' \quad e' \text{ DLF} \quad e' \overset{\text{DLF}}{\longmapsto} e''}{\Upsilon \vdash \mathtt{isTrue}\, e \overset{\text{PROP}}{\longmapsto} e'' \geq 1} \tag{8.35c}$$

$$\frac{e \curvearrowright e' \quad e' \text{ CONJ} \quad e' \overset{\text{CONJ}}{\longmapsto} c'}{\Upsilon \vdash \mathtt{isTrue}\, e \overset{\text{PROP}}{\longmapsto} c'} \tag{8.35d}$$

$$\frac{}{\Upsilon \vdash e_1 \,\mathtt{op}\, e_2 \overset{\text{PROP}}{\longmapsto} e_1' \,\mathtt{op}\, e_2'} \text{ for } \mathtt{op} \in \{=, \leq\} \tag{8.35e}$$

$$\frac{\Upsilon \vdash c_1 \vee c_2 \overset{\text{DISJ}}{\longmapsto} c'}{\Upsilon \vdash c_1 \vee c_2 \overset{\text{PROP}}{\longmapsto} c'} \tag{8.35f}$$

$$\frac{\Upsilon \vdash c_1 \overset{\text{PROP}}{\longmapsto} c_1' \quad \Upsilon \vdash c_2 \overset{\text{PROP}}{\longmapsto} c_2'}{\Upsilon \vdash c_1 \wedge c_2 \overset{\text{PROP}}{\longmapsto} c_1' \wedge c_2'} \tag{8.35g}$$

$$\frac{\rho \overset{\text{RTYPE}}{\longmapsto} \rho' \quad \Upsilon, x : \rho' \vdash c \overset{\text{PROP}}{\longmapsto} c'}{\Upsilon \vdash \exists x : \rho \,\mathbf{.}\, c \overset{\text{PROP}}{\longmapsto} \exists x : \rho' \,\mathbf{.}\, c'} \tag{8.35h}$$

$$\frac{\Upsilon \vdash \bigvee_{i:\sigma} c \overset{\text{DISJ}}{\longmapsto} c'}{\Upsilon \vdash \bigvee_{i:\sigma} c \overset{\text{PROP}}{\longmapsto} c'} \tag{8.35i}$$

$$\frac{\Upsilon \vdash c \overset{\text{PROP}}{\longmapsto} c'}{\Upsilon \vdash \bigwedge_{i:\sigma} c \overset{\text{PROP}}{\longmapsto} \bigwedge_{i:\sigma} c'} \tag{8.35j}$$

$$\frac{\left\{\Upsilon \vdash c_j \overset{\text{PROP}}{\longmapsto} c_j'\right\}_{j=1}^{m}}{\Upsilon \vdash \mathtt{case}_{i.\zeta}\ \varepsilon\ \mathtt{of}\ \left\{l_j \Rightarrow c_j\right\}_{j=1}^{m} \overset{\text{PROP}}{\longmapsto} \mathtt{case}_{i.\zeta}\ \varepsilon\ \mathtt{of}\ \left\{l_j \Rightarrow c_j'\right\}_{j=1}^{m}} \tag{8.35k}$$

$$\frac{\left\{\Upsilon \vdash c_j \overset{\text{PROP}}{\longmapsto} c_j'\right\}_{j=1}^{m}}{\Upsilon \vdash \mathtt{case}_{i.\zeta}\ \varepsilon\ \mathtt{of}\ \left\{k_j \Rightarrow c_j\right\}_{j=1}^{m} \overset{\text{PROP}}{\longmapsto} \mathtt{case}_{i.\zeta}\ \varepsilon\ \mathtt{of}\ \left\{k_j \Rightarrow c_j'\right\}_{j=1}^{m}} \tag{8.35l}$$

$$\frac{c\,\varepsilon \rightsquigarrow c' \quad \Upsilon \vdash c' \overset{\text{PROP}}{\longmapsto} c''}{\Upsilon \vdash c\,\varepsilon \overset{\text{PROP}}{\longmapsto} c''} \tag{8.35m}$$

$$\frac{\Upsilon \vdash c \overset{\text{PROP}}{\longmapsto} c'}{\Upsilon \vdash c : \zeta \overset{\text{PROP}}{\longmapsto} c' : \zeta} \tag{8.35n}$$

The definition for unindexed syntactic forms are identical to those given in Chapter 4. The indexed disjunction and conjunction, and case propositions are compiled by compiling their

nested constructs. Applications are first put into ANF, the standard technique being used in all judgements.

8.4.4 Disjunctive Proposition Compiler

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

$$\frac{\{x_j : \rho_j \backsimeq c_j\}_{j=1}^{m}}{\Upsilon \vdash c \multimap (c_1 \wedge \cdots \wedge c_m \wedge c)} \tag{8.36}$$

where $FV(c) = \{x_1, \ldots, x_m\}$ and $\Upsilon(x_j) = \rho_j$ for $j = 1, \ldots, m$. The $x : \rho \backsimeq c$ relation was defined in Section 7.3. It provides a proposition c corresponding to the bounds declared by ρ.

Let $e \circledast e_1 \hookrightarrow e_2$ be a judgement which multiplies e to all constant terms in e_1, producing

e_2. Its definition is

$$\frac{}{e \circledast x \hookrightarrow x} \tag{8.37a}$$

$$\frac{}{e \circledast r \hookrightarrow e * r} \tag{8.37b}$$

$$\frac{e \circledast e_1 \hookrightarrow e_2}{e \circledast -e_1 \hookrightarrow -e_2} \tag{8.37c}$$

$$\frac{e \circledast e_1 \hookrightarrow e_1' \quad e \circledast e_2 \hookrightarrow e_2'}{e \circledast (e_1 \,\mathtt{op}\, e_2) \hookrightarrow (e_1' \,\mathtt{op}\, e_2')} \text{ for } \mathtt{op} \in \{+, -\} \tag{8.37d}$$

$$\frac{}{e \circledast (e_1 * e_2) \hookrightarrow e * (e_1 * e_2)} \text{ if } FV\,(e_1 * e_2) = \emptyset \tag{8.37e}$$

$$\frac{}{e \circledast (x_1 * e_2) \hookrightarrow (x_1 * e_2)} \tag{8.37f}$$

$$\frac{}{e \circledast (e_1 * x_2) \hookrightarrow (e_1 * x_2)} \tag{8.37g}$$

$$\frac{e_1 \text{ ANF}}{e \circledast (e_1 * e_2) \hookrightarrow (e_1 * e_2)} \tag{8.37h}$$

$$\frac{e_2 \text{ ANF}}{e \circledast (e_1 * e_2) \hookrightarrow (e_1 * e_2)} \tag{8.37i}$$

$$\frac{e_1 \neg\text{ANF} \quad e \circledast e_1 \hookrightarrow e_1'}{e \circledast (e_1 * e_2) \hookrightarrow (e_1' * e_2)} \text{ if } FV\,(e_1) \neq \emptyset,\ e_1 \text{ not } x \tag{8.37j}$$

$$\frac{e_2 \neg\text{ANF} \quad e \circledast e_2 \hookrightarrow e_2'}{e \circledast (e_1 * e_2) \hookrightarrow (e_1 * e_2')} \text{ if } FV\,(e_2) \neq \emptyset,\ e_2 \text{ not } x \tag{8.37k}$$

$$\frac{e \circledast e_1 \hookrightarrow e_2}{e \circledast \sum_{i:\sigma} e_1 \hookrightarrow \sum_{i:\sigma} e_2} \text{ where } i \notin FV\,(e) \tag{8.37l}$$

$$\frac{\left\{e \circledast e_j \hookrightarrow e_j'\right\}_{j=1}^{m}}{e \circledast \mathtt{case}_{i.\tau}\ \varepsilon\ \mathtt{of}\ \left\{l_j \Rightarrow e_j\right\}_{j=1}^{m} \hookrightarrow \mathtt{case}_{i.\tau}\ \varepsilon\ \mathtt{of}\ \left\{l_j \Rightarrow e_j'\right\}_{j=1}^{m}} \tag{8.37m}$$

$$\frac{\left\{e \circledast e_j \hookrightarrow e_j'\right\}_{j=1}^{m}}{e \circledast \mathtt{case}_{i.\tau}\ \varepsilon\ \mathtt{of}\ \left\{k_j \Rightarrow e_j\right\}_{j=1}^{m} \hookrightarrow \mathtt{case}_{i.\tau}\ \varepsilon\ \mathtt{of}\ \left\{k_j \Rightarrow e_j'\right\}_{j=1}^{m}} \tag{8.37n}$$

$$\frac{e_1\,\varepsilon \text{ ANF}}{e \circledast e_1\,\varepsilon \hookrightarrow e_1\,\varepsilon} \tag{8.37o}$$

$$\frac{e_1\,\varepsilon \neg\text{ANF} \quad e_1\,\varepsilon \rightsquigarrow e' \quad e \circledast e' \hookrightarrow e''}{e \circledast e_1\,\varepsilon \hookrightarrow e''} \tag{8.37p}$$

$$\frac{e \circledast e_1 \hookrightarrow e_2}{e \circledast e_1 : \tau \hookrightarrow e_2 : \tau} \tag{8.37q}$$

In the rule for $e \circledast \sum_{i:\sigma} e_1$, index i must be α-converted if i is free in e.

Let $e \circledast c_1 \hookrightarrow c_2$ be a corresponding judgement on a proposition. Its definition is by induction on the form of c_1 and simply recurses on nested propositions and expressions. Attention must be paid to α-conversion when recursing into index constraints. Its definition

is

$$\frac{}{e \circledast \mathtt{T} \hookrightarrow \mathtt{T}} \tag{8.38a}$$

$$\frac{}{e \circledast \mathtt{F} \hookrightarrow \mathtt{F}} \tag{8.38b}$$

$$\frac{e \circledast e_1 \hookrightarrow e_2}{e \circledast (\mathtt{isTrue}\ e_1) \hookrightarrow (\mathtt{isTrue}\ e_2)} \tag{8.38c}$$

$$\frac{e \circledast e_{11} \hookrightarrow e_{21} \quad e \circledast e_{12} \hookrightarrow e_{22}}{e \circledast (e_{11}\ \mathtt{op}\ e_{12}) \hookrightarrow (e_{21}\ \mathtt{op}\ e_{22})} \text{ for } \mathtt{op} \in \{=, \leq\} \tag{8.38d}$$

$$\frac{e \circledast c_{11} \hookrightarrow c_{21} \quad e \circledast c_{12} \hookrightarrow c_{22}}{e \circledast (c_{11}\ \mathtt{op}\ c_{12}) \hookrightarrow (c_{21}\ \mathtt{op}\ c_{22})} \text{ for } \mathtt{op} \in \{\vee, \wedge\} \tag{8.38e}$$

$$\frac{e \circledast c_1 \hookrightarrow c_2}{e \circledast (\exists x : \rho \centerdot c_1) \hookrightarrow (\exists x : \rho \centerdot c_1)} \text{ where } x \notin FV(e) \tag{8.38f}$$

$$\frac{e \circledast c_1 \hookrightarrow c_2}{e \circledast \left(\underset{i:\sigma}{\mathtt{OP}}\, c_1\right) \hookrightarrow \left(\underset{i:\sigma}{\mathtt{OP}}\, c_2\right)} \text{ for } \mathtt{OP} \in \{\bigvee, \bigwedge\}, \text{ where } i \notin FV(e) \tag{8.38g}$$

$$\frac{\left\{e \circledast c_j \hookrightarrow c'_j\right\}_{j=1}^{m}}{e \circledast \mathtt{case}_{i.\zeta}\ \varepsilon\ \mathtt{of}\ \left\{l_j \Rightarrow c_j\right\}_{j=1}^{m} \hookrightarrow \mathtt{case}_{i.\zeta}\ \varepsilon\ \mathtt{of}\ \left\{l_j \Rightarrow c'_j\right\}_{j=1}^{m}} \tag{8.38h}$$

$$\frac{\left\{e \circledast c_j \hookrightarrow c'_j\right\}_{j=1}^{m}}{e \circledast \mathtt{case}_{i.\zeta}\ \varepsilon\ \mathtt{of}\ \left\{k_j \Rightarrow c_j\right\}_{j=1}^{m} \hookrightarrow \mathtt{case}_{i.\zeta}\ \varepsilon\ \mathtt{of}\ \left\{k_j \Rightarrow c'_j\right\}_{j=1}^{m}} \tag{8.38i}$$

$$\frac{c_1\ \varepsilon \rightsquigarrow c' \quad e \circledast c' \hookrightarrow c_2}{e \circledast c_1\ \varepsilon \hookrightarrow c_2} \tag{8.38j}$$

$$\frac{e \circledast c_1 \hookrightarrow c_2}{e \circledast c_1 : \zeta \hookrightarrow c_2 : \zeta} \tag{8.38k}$$

A variable x of type ρ will be disaggregated into a variable x' of type $i : \sigma \to \rho$, where the disjunction is indexed over σ. Let $x : \rho^{\text{MIP}} \Rrightarrow x'_{i:\sigma} \looparrowright c$ be a judgement providing a proposition c that relates the original and disaggregated variables. Its definition is

$$\frac{x_x : \rho^{\text{MIP}} \Rrightarrow x'_{i:\sigma} \looparrowright^* c}{x : \rho^{\text{MIP}} \Rrightarrow x'_{i:\sigma} \looparrowright c} \tag{8.39}$$

which relies on a helper judgement $e_x : \rho^{\text{MIP}} \Rrightarrow x'_{i:\sigma} \looparrowright^* c$ that involves an extra construct e. Its definition is inductive on the form of ρ^{MIP},

$$\frac{}{e_x : \rho \Rrightarrow x'_{i:\sigma} \looparrowright^* \left(e = \sum_{i:\sigma} \{x'\ i/x\}\, e\right)} \text{ where } \rho \text{ is any numeric type} \tag{8.40a}$$

$$\frac{(e\ i')_x : \rho' \Rrightarrow x'_{i:\sigma} \looparrowright^* c}{e_x : (i' : \sigma' \to \rho') \Rrightarrow x'_{i:\sigma} \looparrowright^* \bigwedge_{i':\sigma'} c} \text{ where } i' \notin FV(e) \tag{8.40b}$$

Let $\Upsilon \vdash c_1 \overset{\text{DISJ}}{\longmapsto} c_2$ be a disjunctive proposition compiler.

$$\frac{\Upsilon \vdash \left(\bigvee_{i:[1,2]} \texttt{case}\ i\ \texttt{of}\ 1 \Rightarrow c_A \mid 2 \Rightarrow c_B \right) \overset{\text{DISJ}}{\longmapsto} c'}{\Upsilon \vdash (c_A \vee c_B) \overset{\text{DISJ}}{\longmapsto} c'} \text{ where } i \notin FV\,(c_A \vee c_B) \tag{8.41a}$$

$$\frac{\begin{array}{c} \Upsilon \vdash c \overset{\text{PROP}}{\longmapsto} c' \quad \Upsilon \overset{\text{CTXT}}{\longmapsto} \Upsilon' \\ \Upsilon' \vdash c' \multimap c'' \quad (y\,i) \circledast \{(\vec{x}'\,i)\,/\vec{x}\}\,c'' \hookrightarrow c''' \\ \left\{x_j : \rho_j \Rrightarrow x'_{j,i:\sigma} \rightarrowtail c^*_j\right\}_{j=1}^{m} \end{array}}{\Upsilon \vdash \bigvee_{i:\sigma} c \overset{\text{DISJ}}{\longmapsto} \left(\begin{array}{c} \exists \vec{x}' : (i : \sigma \to \vec{\rho})\,.\,\exists y : (i : \sigma \to [0,1])\,. \\ (c^*_1 \wedge \cdots \wedge c^*_m) \wedge \left(\sum_{i:\sigma} y\,i = 1 \right) \wedge \bigwedge_{i:\sigma} c''' \end{array} \right)} \tag{8.41b}$$

Binary disjunction could have been handled as in the unindexed theory, but instead we convert it into an indexed disjunction and allow the second rule to produce the result. The compiler for indexed disjunctions generates a disaggregated variable x' to replace each variable x, but x' is of type $i : \sigma \to \rho$. So we can think of $x'\,i$ as the variable associated with the i^{th} disjunct. Similarly, $y : (i : \sigma \to [0,1])$ allows us to think of $y\,i$ as the binary associated with each disjunct.

The preconditions are similar to that in the unindexed theory. The disjunct c is itself compiled first. This produces c' that will satisfy c' MIP. Then $\multimap$ is employed to add bounding propositions, each x is replaced with $x'\,i$, and constant terms are multiplied by $y\,i$. Unlike in the unindexed theory, producing the propositions that relate the disaggregated variables with the original is not so straightforward. The judgement $x : \rho^{\text{MIP}} \Rrightarrow x'_{i:\sigma} \rightarrowtail c$ was defined to do this and is used to produce c^*_j for each variable x_j.

8.4.5 Program Compiler

Let $p \overset{\text{PROG}}{\longmapsto} p^{\text{MIP}}$ be a program compiler. Its definition is

$$\frac{\left\{\rho_j \overset{\text{RTYPE}}{\longmapsto} \rho'_j\right\}_{j=1}^{m} \qquad x_1 : \rho_1, \ldots, x_m : \rho_m \vdash c \overset{\text{PROP}}{\longmapsto} c'}{\delta_{x_1:\rho_1,\ldots,x_m:\rho_m}\,\{e \mid c\} \overset{\text{PROG}}{\longmapsto} \delta_{x_1:\rho'_1,\ldots,x_m:\rho'_m}\,\{e \mid c'\}} \tag{8.42}$$

8.5 Results

Example 8.1 In Example 7.2 on page 124, we demonstrated the use of a disjunctive constraint. We use our software to check that the program satisfies p DisjVarsBounded. It does. Thus, the compiler can be applied to it. We do so and it produces the following mixed-integer program.

```
1 var w:<10.0, 90.0>
2 var x:[{1, ..., 10}] -> <5.0, 75.0>
3
4 min w subject_to
5
6 CONJ i:{1, ..., 10} w <= x[i] + 4.0,
```

```
7
8  exists y:[{1, ..., 10}] -> [0, 1]
9  exists x1:[{1, ..., 10}] -> [{1, ..., 10}] -> <5.0, 75.0>
10 exists w1:[{1, ..., 10}] -> <10.0, 90.0>
11   w = SUM i:{1, ..., 10} w1[i],
12   CONJ d:{1, ..., 10} x[d] = SUM i:{1, ..., 10} x1[i][d],
13   SUM i:{1, ..., 10} y[i] = 1,
14   CONJ i:{1, ..., 10}
15     10.0 * y[i] <= w1[i],
16     w1[i] <= 90.0 * y[i],
17     (CONJ d:{1, ..., 10}
18       5.0 * y[i] <= x1[i][d],
19       x1[i][d] <= 75.0 * y[i]),
20     w1[i] >= x1[i][i] + 4.0 * y[i]
```

Chapter 9

Application: Switched Flow Process

We have provided several modeling frameworks and transformations between them. We now apply these to a small switched flow process. Following the structure of the dissertation, we first present an LCCA model of the system, then convert it into an MP model with Boolean and disjunctive constraints, and then to a MIP model. This is done manually. Following this, we show how the MP model can be expressed formally in the language we defined and finally use our compiler to automatically produce the equivalent MIP model.

The following is a conceptual description of the system we wish to model.

> Figure 9.1 depicts a tank being filled by two hybrid processes, α and β, and being emptied continuously at a rate of $F^{\text{out}} = 1.8$. Initially, the material level in the tank is $M_0 = 20.0$. The tank's maximum capacity $M^{\max} = 150.0$ and the material level should never fall below $M^{\min} = 10.0$.
>
> Process α represents a pump that can be either on or off. When it is on, it provides material to the tank at rate 2.0. There is also an operating cost of 10.0 per unit time for running the pump. Operational constraints on the pump forbid it from being continuously run longer than 30.0 time units; it must be switched off before this time limit is reached. There are no operating costs while it is off, but it must not be switched on again in less than 2.0 time units. When it is switched on again, if at all, a startup cost of 50.0 is incurred.
>
> Process β is similar, but it represents a pump that is always on, either at a high or low setting. In the high setting, material flows to the tank at rate 4.0 and the operating cost is 15.0 per time unit. In the low setting, the material flow rate drops to 0.5, and the operating cost is 2.0. Once set to low, the pump cannot be switched to the high setting again for at least 3.0 time units, and, when it does, a startup cost of 40.0 is incurred.
>
> We wish to study how the material level changes over time and to understand the cost of running the system for $T^{\max} = 500.0$ time units.

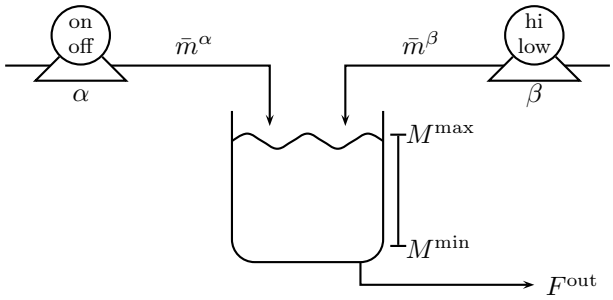


Figure 9.1: Schematic of switched flow process.

9.1 LCCA Model

A model in the LCCA framework, introduced in Chapter 2, can be formulated with relative ease. First, the following variables are defined:

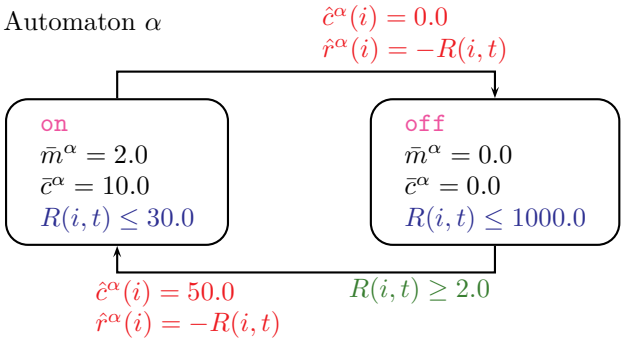
- $M(i,t)$ is material level in tank, and $\bar{m}^a(q)$ is rate at which process a puts material into tank when in mode q.
- $C(i,t)$ is total incurred cost, $\bar{c}^a(q)$ is operating cost incurred for process a in mode q, and $\hat{c}^a(i)$ is instantaneous cost incurred for process a switching modes at event i.
- R and S stand for the amount of time processes α and β, respectively, have been in their current discrete mode, $\hat{r}^a(i)$ and $\hat{s}^a(i)$ are the instantaneous changes to these values at event i.

Then, the LCCA model is

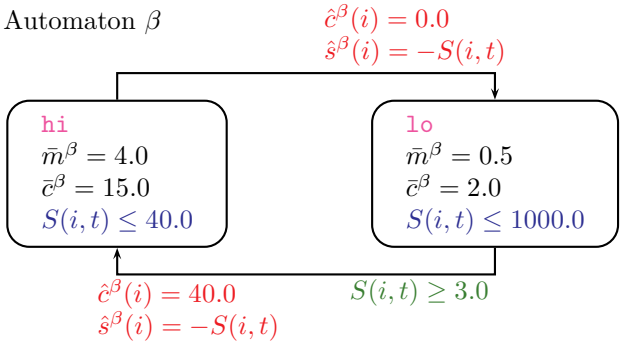
$$n = 10 \tag{9.1}$$

$$t_n^e = T^{\max} \tag{9.2a}$$

$$t_1^s = 0.0 \tag{9.2b}$$



(9.3a)



(9.3b)

$$\frac{\mathrm{d}R\,(i,t)}{\mathrm{d}t} = 1.0 \qquad \forall\,(i,t) \in \mathbb{T} \tag{9.4a}$$

$$R\,(i+1,t) = R\,(i,t) + \hat{r}^{\alpha}\,(i) \qquad \forall i \in \mathbb{N}\backslash\,\{n\} \tag{9.4b}$$

$$R\,(1,t_1^s) = 0.0 \tag{9.4c}$$

$$\frac{\mathrm{d}S\,(i,t)}{\mathrm{d}t} = 1.0 \qquad \forall\,(i,t) \in \mathbb{T} \tag{9.4d}$$

$$S\,(i+1,t) = S\,(i,t) + \hat{s}^{\beta}\,(i) \qquad \forall i \in \mathbb{N}\backslash\,\{n\} \tag{9.4e}$$

$$S\,(1,t_1^s) = 0.0 \tag{9.4f}$$

$$\frac{\mathrm{d}M\,(i,t)}{\mathrm{d}t} = \bar{m}^{\alpha}\left(Q^{\alpha}\,(i)\right) + \bar{m}^{\beta}\left(Q^{\beta}\,(i)\right) - F^{\mathrm{out}} \qquad \forall\,(i,t) \in \mathbb{T} \tag{9.4g}$$

$$M\,(i+1,t) = M\,(i,t) \qquad \forall i \in \mathbb{N}\backslash\,\{n\} \tag{9.4h}$$

$$M\,(1,t_1^s) = M_0 \tag{9.4i}$$

$$M^{\min} \leq M\,(i,t) \leq M^{\max} \qquad \forall\,(i,t) \in \mathbb{T} \tag{9.4j}$$

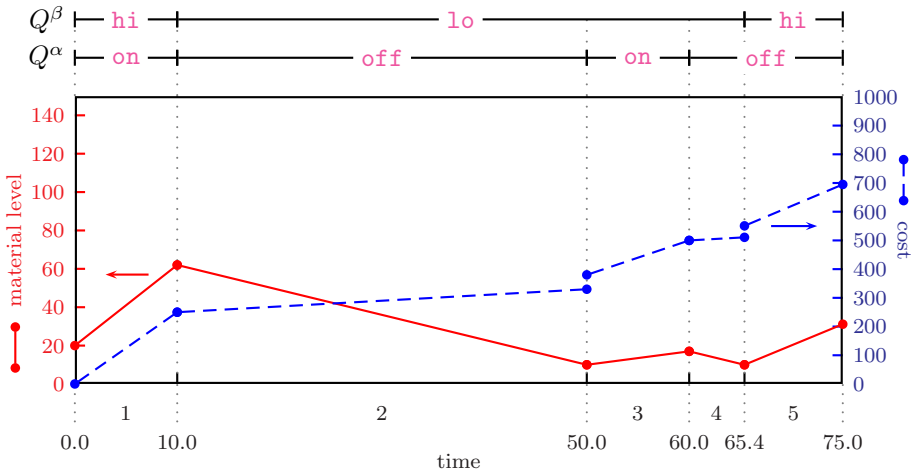
$$\frac{\mathrm{d}C\,(i,t)}{\mathrm{d}t} = \bar{c}^{\alpha}\left(Q^{\alpha}\,(i)\right) + \bar{c}^{\beta}\left(Q^{\beta}\,(i)\right) \qquad \forall\,(i,t) \in \mathbb{T} \tag{9.4k}$$

$$C\,(i+1,t) = C\,(i,t) + \hat{c}^{\alpha}\,(i) + \hat{c}^{\beta}\,(i) \qquad \forall i \in \mathbb{N}\backslash\,\{n\} \tag{9.4l}$$

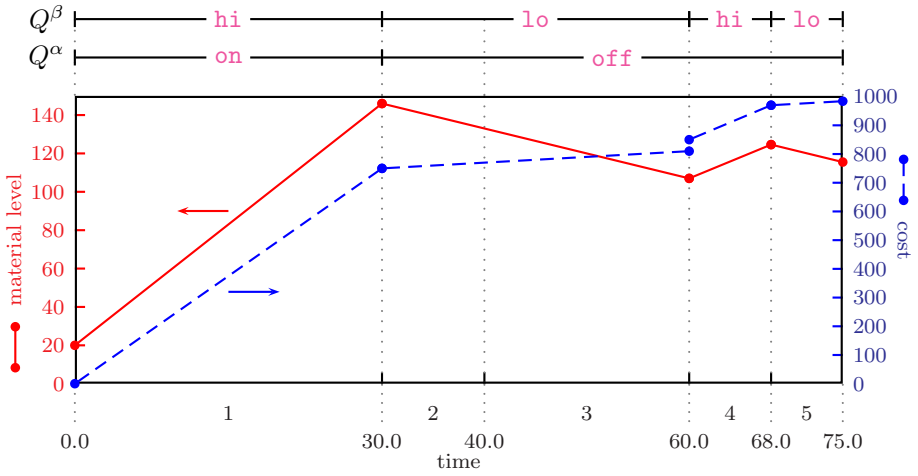
$$C\,(1,t_1^s) = 0.0 \tag{9.4m}$$

All components $(n, G_t, \mathbf{X}, Aut, G_V)$ of an LCCA model have been defined above. The first equation sets n to 10. Equations (9.2) represent constraints on the timeline in the form of G_t. The set of variables $\mathbf{X} = \{R, S, M, C\}$. A component automata is provided for each hybrid process, $Aut = \{\alpha, \beta\}$, and both are declared graphically. The remaining equations constitute constraint G_V. Figure 9.2 shows two feasible trajectories for this system.

Equation (9.4g) states that its rate of change depends on $\bar{m}^{\alpha}$ and $\bar{m}^{\beta}$, and these values themselves depend on the modes of the automata. Consider the trajectory shown in Figure



(a) Lower material levels, more switching. Complete data on page 211.



(b) Higher material levels, less switching. Complete data on page 211.

Figure 9.2: Initial segments of two feasible trajectories for switched flow process.

9.2a during interval 3. Within this interval, $Q^\alpha(i) = \texttt{on}$ and $Q^\beta(i) = \texttt{lo}$. In these modes, the differential equation for M is

$$\frac{\mathrm{d}M(i,t)}{\mathrm{d}t} = 2.0 + 0.5 - 1.8$$

$$= 0.7$$

At $t = 60.0$, automaton α transitions to its off mode, the equation becomes

$$\frac{\mathrm{d}M(i,t)}{\mathrm{d}t} = 0.0 + 0.5 - 1.8$$

$$= -1.3$$

The right-hand-side switches between different constants.

The differential equation for cost C is similar, but costs can also be incurred instantaneously by equation (9.4l). These represent the startup costs. Consider the transition from interval 2 to 3 in Figure 9.2a. Automaton α transitions from `off` to `on`. The reset on this transition includes the equation $\hat{c}^{\alpha}(i) = 50.0$. Automata β remains in its `lo` mode. This is modeled with the dummy transition, which includes the reset $\hat{c}^{\beta}(i) = 0.0$ by definition. Equation (9.4l) becomes

$$\begin{aligned} C(i+1,t) &= C(i,t) + 50.0 + 0.0 \\ &= C(i,t) + 50.0 \end{aligned}$$

which means the cost gets incremented by 50.0. The data is correctly plotted with two filled in circles at $t = 50.0$. Since this is an event point, there are two time points, $(2, 50.0)$ and $(3, 50.0)$, at this position.

R and S are called clock variables because they keep track of time, as indicated by setting their rates to 1.0. Equation (9.4b) increments the value of clock R by $\hat{r}^{\alpha}(i)$ at every event. On both transitions of α, the reset is $\hat{r}^{\alpha}(i) = -R(i,t)$, but this is simply the negation of the clock value at the beginning of the event. In other words, the clock gets reset to 0.0. Dummy transitions are not shown, but their definition is fixed such that $\hat{r}^{\alpha}(i) = 0.0$ on those transitions. Clocks are not reset when an automaton transitions from some mode back to itself.

The guard on the transition from `off` to `on` is $R(i,t) \geq 2.0$. Upon entry into the `off` mode, clock R is set to 0.0 and evolves at rate 1.0 while in this mode. Thus, this guard states that the automaton must remain in the `off` mode for at least 2.0 time units. Finally, there are several invariants regarding the maximum time that each automaton can remain in its discrete modes, and these are satisfied in the trajectory shown.

Finally, there are several invariants stated. In Figure 9.2b, automaton α is in the `on` mode during interval 1. In this mode, the invariant $R(i,t) \leq 30.0$ must hold. Since R was initialized to the value 0.0, the automata had to transition in at most 30.0 time units.

9.2 MP Model

Now, using the transformation methods discussed in Chapter 3, we present a GDP model that is equivalent to the LCCA model of the previous section. Several new variables are required; the naming convention follows that in the description of the general procedure. The model is

$$t_i^s \leq t_i^e \qquad \forall i \in \mathbb{N} \tag{9.5a}$$

$$t_i^e = t_{i+1}^s \qquad \forall i \in \mathbb{N} \backslash \{n\} \tag{9.5b}$$

$$\Delta t_i = t_i^e - t_i^s \qquad \forall i \in \mathbb{N} \tag{9.5c}$$

$$t_n^e = T^{\max} \tag{9.6a}$$

$$t_1^s = 0.0 \tag{9.6b}$$

$$\begin{bmatrix} Y^\alpha(\texttt{on}, i) \\ R^s(i) \leq 30.0 \\ R^e(i) \leq 30.0 \end{bmatrix} \vee \begin{bmatrix} Y^\alpha(\texttt{off}, i) \\ R^s(i) \leq 1000.0 \\ R^e(i) \leq 1000.0 \end{bmatrix} \qquad \forall i \in \mathbb{N} \tag{9.7a}$$

$$\begin{bmatrix} Z^\alpha(\texttt{on}, \texttt{off}, i) \\ \hat{c}^\alpha(i) = 0.0 \\ \hat{r}^\alpha(i) = -R^e(i) \end{bmatrix} \vee \begin{bmatrix} Z^\alpha(\texttt{off}, \texttt{on}, i) \\ R^e(i) \geq 2.0 \\ \hat{c}^\alpha(i) = 50.0 \\ \hat{r}^\alpha(i) = -R^e(i) \end{bmatrix} \vee \begin{bmatrix} YY^\alpha(i) \\ \hat{c}^\alpha(i) = 0.0 \\ \hat{r}^\alpha(i) = 0.0 \end{bmatrix} \qquad \forall i \in \mathbb{N} \backslash \{n\} \tag{9.7b}$$

$$\begin{bmatrix} Y^\beta(\texttt{hi}, i) \\ S^s(i) \leq 40.0 \\ S^e(i) \leq 40.0 \end{bmatrix} \vee \begin{bmatrix} Y^\beta(\texttt{lo}, i) \\ S^s(i) \leq 1000.0 \\ S^e(i) \leq 1000.0 \end{bmatrix} \qquad \forall i \in \mathbb{N} \tag{9.7c}$$

$$\begin{bmatrix} Z^\beta(\texttt{hi}, \texttt{lo}, i) \\ \hat{c}^\beta(i) = 0.0 \\ \hat{s}^\beta(i) = -S^e(i) \end{bmatrix} \vee \begin{bmatrix} Z^\beta(\texttt{lo}, \texttt{hi}, i) \\ S^e(i) \geq 3.0 \\ \hat{c}^\beta(i) = 40.0 \\ \hat{s}^\beta(i) = -S^e(i) \end{bmatrix} \vee \begin{bmatrix} YY^\beta(i) \\ \hat{c}^\beta(i) = 0.0 \\ \hat{s}^\beta(i) = 0.0 \end{bmatrix} \qquad \forall i \in \mathbb{N} \backslash \{n\} \tag{9.7d}$$

$$R^e(i) = R^s(i) + \Delta t_i \qquad \forall i \in \mathbb{N} \tag{9.8a}$$

$$R^s(i+1) = R^e(i) + \hat{r}^\alpha(i) \qquad \forall i \in \mathbb{N} \backslash \{n\} \tag{9.8b}$$

$$R^s(1) = 0.0 \tag{9.8c}$$

$$S^e(i) = S^s(i) + \Delta t_i \qquad \forall i \in \mathbb{N} \tag{9.8d}$$

$$S^s(i+1) = S^e(i) + \hat{s}^\beta(i) \qquad \forall i \in \mathbb{N} \backslash \{n\} \tag{9.8e}$$

$$S^s(1) = 0.0 \tag{9.8f}$$

$$M^e(i) = M^s(i) + \bar{w}^{m,\alpha}(i) + \bar{w}^{m,\beta}(i) - F^{\text{out}}\Delta t_i \qquad \forall i \in \mathbb{N} \tag{9.8g}$$

$$M^s(i+1) = M^e(i) \qquad \forall i \in \mathbb{N}\backslash\{n\} \tag{9.8h}$$

$$M^s(1) = M_0 \tag{9.8i}$$

$$M^{\min} \leq M^s(i) \leq M^{\max} \qquad \forall i \in \mathbb{N} \tag{9.8j}$$

$$M^{\min} \leq M^e(i) \leq M^{\max} \qquad \forall i \in \mathbb{N} \tag{9.8k}$$

$$C^e(i) = C^s(i) + \bar{w}^{c,\alpha}(i) + \bar{w}^{c,\beta}(i) \qquad \forall i \in \mathbb{N} \tag{9.8l}$$

$$C^s(i+1) = C^e(i) + \hat{c}^{\alpha}(i) + \hat{c}^{\beta}(i) \qquad \forall i \in \mathbb{N}\backslash\{n\} \tag{9.8m}$$

$$C^s(1) = 0.0 \tag{9.8n}$$

$$\bigvee_{q \in \mathbb{Q}^a} \begin{bmatrix} Y^a(q,i) \\ \bar{w}^{m,a}(i) = \bar{m}^a(q)\Delta t_i \\ \bar{w}^{c,a}(i) = \bar{c}^a(q)\Delta t_i \end{bmatrix} \qquad \forall i \in \mathbb{N}, \forall a \in Aut \tag{9.8o}$$

$$\bigvee_{q \in \mathbb{Q}^a} Y^a(q,i) \qquad \forall i \in \mathbb{N}, \forall a \in Aut \tag{9.9}$$

$$YYY(i) \Rightarrow YYY(i+1) \qquad \forall i \in \mathbb{N}\backslash\{n-1, n\} \tag{9.10a}$$

$$YYY(i) \Rightarrow (\Delta t_{i+1} = 0.0) \qquad \forall i \in \mathbb{N}\backslash\{n\} \tag{9.10b}$$

$$YY^a(i) \Leftrightarrow \bigvee_{q \in \mathbb{Q}^a} Z^a(q,q,i) \qquad \forall i \in \mathbb{N}\backslash\{n\}, \forall a \in Aut \tag{9.11a}$$

$$YYY(i) \Leftrightarrow \bigwedge_{a \in Aut} YY^a(i) \qquad \forall i \in \mathbb{N}\backslash\{n\} \tag{9.11b}$$

$$Z^a(q,q',i) \Leftrightarrow (Y^a(q,i) \wedge Y^a(q',i+1)) \qquad \forall i \in \mathbb{N}\backslash\{n\}, \forall a \in Aut, \forall q,q' \in \mathbb{Q}^a \tag{9.11c}$$

The first set of constraints involve the timeline variables. Then, there are two disjunctions for each of the two automata. Following these, there is a set of equations for each of the continuous variables R, S, M, and C. Finally, there is a disjunction needed for the auxiliary variable $\bar{w}$, and the symmetry breaking constraints.

This GDP model is rather more complex than the corresponding LCCA model. It would have been difficult to think of this model directly. It requires several variables, e.g. $\bar{w}^{m,\alpha}$, unrelated to the physical conception of the system. Various constraints, such as evolution of mass, have to be defined in terms of these auxiliary variables, making the constraints themselves less intuitive.

9.3 MIP Model

Appendix A reviews methods for converting a GDP to an MIP. We use the convex hull method, which requires introduction of many new variables. The notational convention used employs superscripts of the form 1, 2, or 11, 21. For example, if there is a disjunction with two disjuncts, two disaggregated variables will be needed for each variable. For the variable X, we introduce X^1 and X^2 as its disaggregated variables. If the same variable X is also used in another disjunction, a separate set of disaggregated variables will be needed for it. In that case, double superscripts are used, e.g. X^{21} is the disaggregated variable for the 2nd disjunct of the 1st disjunction. By convention, Boolean variables are capitalized and their corresponding binary $\{0, 1\}$ variables are in lower case.

The full linear MIP model for the example is

$$t_i^s \leq t_i^e \qquad \forall i \in \mathbb{N} \tag{9.12a}$$

$$t_i^e = t_{i+1}^s \qquad \forall i \in \mathbb{N} \backslash \{n\} \tag{9.12b}$$

$$\Delta t_i = t_i^e - t_i^s \qquad \forall i \in \mathbb{N} \tag{9.12c}$$

$$t_n^e = T^{\max} \tag{9.13a}$$

$$t_1^s = 0.0 \tag{9.13b}$$

$$\left[\begin{array}{c} 0.0 \leq R^{s,1}(i) \leq 30.0 y^{\alpha}(\texttt{on}, i) \\ 0.0 \leq R^{e,11}(i) \leq 30.0 y^{\alpha}(\texttt{on}, i) \end{array}\right] \qquad \forall i \in \mathbb{N} \tag{9.14a}$$

$$\left[\begin{array}{c} 0.0 \leq R^{s,2}(i) \leq 1000.0 y^{\alpha}(\texttt{off}, i) \\ 0.0 \leq R^{e,21}(i) \leq 1000.0 y^{\alpha}(\texttt{off}, i) \end{array}\right] \qquad \forall i \in \mathbb{N} \tag{9.14b}$$

$$R^s(i) = R^{s,1}(i) + R^{s,2}(i) \qquad \forall i \in \mathbb{N} \tag{9.14c}$$

$$R^e(i) = R^{e,11}(i) + R^{e,21}(i) \qquad \forall i \in \mathbb{N} \tag{9.14d}$$

$$\begin{bmatrix} \hat{c}^{\alpha,1}(i) = 0.0 \\ \hat{r}^{\alpha,1}(i) = -R^{e,12}(i) \\ 0.0 \leq R^{e,12}(i) \leq 30.0 z^{\alpha}(\texttt{on}, \texttt{off}, i) \end{bmatrix} \quad \forall i \in \mathbb{N} \setminus \{n\} \tag{9.14e}$$

$$\begin{bmatrix} 2.0 z^{\alpha}(\texttt{off}, \texttt{on}, i) \leq R^{e,22}(i) \leq 1000.0 z^{\alpha}(\texttt{off}, \texttt{on}, i) \\ \hat{c}^{\alpha,2}(i) = 50.0 z^{\alpha}(\texttt{off}, \texttt{on}, i) \\ \hat{r}^{\alpha,2}(i) = -R^{e,22}(i) \end{bmatrix} \quad \forall i \in \mathbb{N} \setminus \{n\} \tag{9.14f}$$

$$\begin{bmatrix} \hat{c}^{\alpha,3}(i) = 0.0 \\ \hat{r}^{\alpha,3}(i) = 0.0 \\ 0.0 \leq R^{e,32}(i) \leq 1000.0 yy^{\alpha}(i) \end{bmatrix} \quad \forall i \in \mathbb{N} \setminus \{n\} \tag{9.14g}$$

$$\hat{c}^{\alpha}(i) = \hat{c}^{\alpha,1}(i) + \hat{c}^{\alpha,2}(i) + \hat{c}^{\alpha,3}(i) \quad \forall i \in \mathbb{N} \setminus \{n\} \tag{9.14h}$$

$$\hat{r}^{\alpha}(i) = \hat{r}^{\alpha,1}(i) + \hat{r}^{\alpha,2}(i) + \hat{r}^{\alpha,3}(i) \quad \forall i \in \mathbb{N} \setminus \{n\} \tag{9.14i}$$

$$R^{e}(i) = R^{e,12}(i) + R^{e,22}(i) + R^{e,32}(i) \quad \forall i \in \mathbb{N} \setminus \{n\} \tag{9.14j}$$

$$\begin{bmatrix} 0.0 \leq S^{s,1}(i) \leq 40.0 y^{\beta}(\texttt{hi}, i) \\ 0.0 \leq S^{e,11}(i) \leq 40.0 y^{\beta}(\texttt{hi}, i) \end{bmatrix} \quad \forall i \in \mathbb{N} \tag{9.14k}$$

$$\begin{bmatrix} 0.0 \leq S^{s,2}(i) \leq 1000.0 y^{\beta}(\texttt{lo}, i) \\ 0.0 \leq S^{e,21}(i) \leq 1000.0 y^{\beta}(\texttt{lo}, i) \end{bmatrix} \quad \forall i \in \mathbb{N} \tag{9.14l}$$

$$S^{s}(i) = S^{s,1}(i) + S^{s,2}(i) \quad \forall i \in \mathbb{N} \tag{9.14m}$$

$$S^{e}(i) = S^{e,11}(i) + S^{e,21}(i) \quad \forall i \in \mathbb{N} \tag{9.14n}$$

$$\begin{bmatrix} \hat{c}^{\beta,1}(i) = 0.0 \\ \hat{s}^{\beta,1}(i) = -S^{e,12}(i) \\ 0.0 \leq S^{e,12}(i) \leq 40.0 z^{\beta}(\texttt{hi}, \texttt{lo}, i) \end{bmatrix} \quad \forall i \in \mathbb{N} \setminus \{n\} \tag{9.14o}$$

$$\begin{bmatrix} 3.0 z^{\beta}(\texttt{lo}, \texttt{hi}, i) \leq S^{e,22}(i) \leq 1000.0 z^{\beta}(\texttt{lo}, \texttt{hi}, i) \\ \hat{c}^{\beta,2}(i) = 40.0 z^{\beta}(\texttt{lo}, \texttt{hi}, i) \\ \hat{s}^{\beta,2}(i) = -S^{e,22}(i) \end{bmatrix} \quad \forall i \in \mathbb{N} \setminus \{n\} \tag{9.14p}$$

$$\begin{bmatrix} \hat{c}^{\beta,3}(i) = 0.0 \\ \hat{s}^{\beta,3}(i) = 0.0 \\ 0.0 \leq S^{e,32}(i) \leq 1000.0 yy^{\beta}(i) \end{bmatrix} \quad \forall i \in \mathbb{N} \setminus \{n\} \tag{9.14q}$$

$$\hat{c}^{\beta}(i) = \hat{c}^{\beta,1}(i) + \hat{c}^{\beta,2}(i) + \hat{c}^{\beta,3}(i) \quad \forall i \in \mathbb{N} \setminus \{n\} \tag{9.14r}$$

$$\hat{s}^{\beta}(i) = \hat{s}^{\beta,1}(i) + \hat{s}^{\beta,2}(i) + \hat{s}^{\beta,3}(i) \quad \forall i \in \mathbb{N} \setminus \{n\} \tag{9.14s}$$

$$S^{e}(i) = S^{e,12}(i) + S^{e,22}(i) + S^{e,32}(i) \quad \forall i \in \mathbb{N} \setminus \{n\} \tag{9.14t}$$

$$R^e(i) = R^s(i) + \Delta t_i \qquad \forall i \in \mathbb{N} \tag{9.15a}$$

$$R^s(i+1) = R^e(i) + \hat{r}^\alpha(i) \qquad \forall i \in \mathbb{N}\backslash\{n\} \tag{9.15b}$$

$$R^s(1) = 0.0 \tag{9.15c}$$

$$S^e(i) = S^s(i) + \Delta t_i \qquad \forall i \in \mathbb{N} \tag{9.15d}$$

$$S^s(i+1) = S^e(i) + \hat{s}^\beta(i) \qquad \forall i \in \mathbb{N}\backslash\{n\} \tag{9.15e}$$

$$S^s(1) = 0.0 \tag{9.15f}$$

$$M^e(i) = M^s(i) + \bar{w}^{m,\alpha}(i) + \bar{w}^{m,\beta}(i) - F^{\text{out}}\Delta t_i \qquad \forall i \in \mathbb{N} \tag{9.15g}$$

$$M^s(i+1) = M^e(i) \qquad \forall i \in \mathbb{N}\backslash\{n\} \tag{9.15h}$$

$$M^s(1) = M_0 \tag{9.15i}$$

$$M^{\min} \leq M^s(i) \leq M^{\max} \qquad \forall i \in \mathbb{N} \tag{9.15j}$$

$$M^{\min} \leq M^e(i) \leq M^{\max} \qquad \forall i \in \mathbb{N} \tag{9.15k}$$

$$C^e(i) = C^s(i) + \bar{w}^{c,\alpha}(i) + \bar{w}^{c,\beta}(i) \qquad \forall i \in \mathbb{N} \tag{9.15l}$$

$$C^s(i+1) = C^e(i) + \hat{c}^\alpha(i) + \hat{c}^\beta(i) \qquad \forall i \in \mathbb{N}\backslash\{n\} \tag{9.15m}$$

$$C^s(1) = 0.0 \tag{9.15n}$$

$$\left[\begin{array}{c} w^{m,a,q}(i) = \bar{m}^a(q)\,\Delta t_i^{1,q,a} \\ w^{c,a,q}(i) = \bar{c}^a(q)\,\Delta t_i^{1,q,a} \\ 0.0 \leq \Delta t_i^{1,q,a} \leq T^{\max} y^a(q,i) \end{array}\right] \qquad \forall i \in \mathbb{N}, \forall a \in Aut, \forall q \in \mathbb{Q}^a \tag{9.15o}$$

$$\bar{w}^{m,a}(i) = \sum_{q \in \mathbb{Q}^a} w^{m,a,q}(i) \qquad \forall i \in \mathbb{N}, \forall a \in Aut \tag{9.15p}$$

$$\bar{w}^{c,a}(i) = \sum_{q \in \mathbb{Q}^a} w^{c,a,q}(i) \qquad \forall i \in \mathbb{N}, \forall a \in Aut \tag{9.15q}$$

$$\Delta t_i = \sum_{q \in \mathbb{Q}^a} \Delta t_i^{1,q,a} \qquad \forall i \in \mathbb{N}, \forall a \in Aut \tag{9.15r}$$

$$\sum_{q \in \mathbb{Q}^a} y^a(q,i) = 1 \qquad \forall i \in \mathbb{N}, \forall a \in Aut \tag{9.16}$$

$$1 - yyy(i) + yyy\,(i+1) \geq 1 \qquad \forall i \in \mathbb{N}\backslash\,\{n-1, n\} \tag{9.17a}$$

$$0.0 \leq \Delta t_{i+1} \leq T^{\max}\,(1 - yyy\,(i)) \qquad \forall i \in \mathbb{N}\backslash\,\{n\} \tag{9.17b}$$

$$\left.\begin{array}{c} 1 - yy^a\,(i) + \sum\limits_{q \in \mathbb{Q}^a} z^a\,(q,q,i) \geq 1 \\ 1 - z^a\,(q,q,i) + yy^a\,(i) \geq 1 \qquad \forall q \in \mathbb{Q}^a \end{array}\right\} \quad \forall i \in \mathbb{N}\backslash\,\{n\}\,, \forall a \in Aut \tag{9.18a}$$

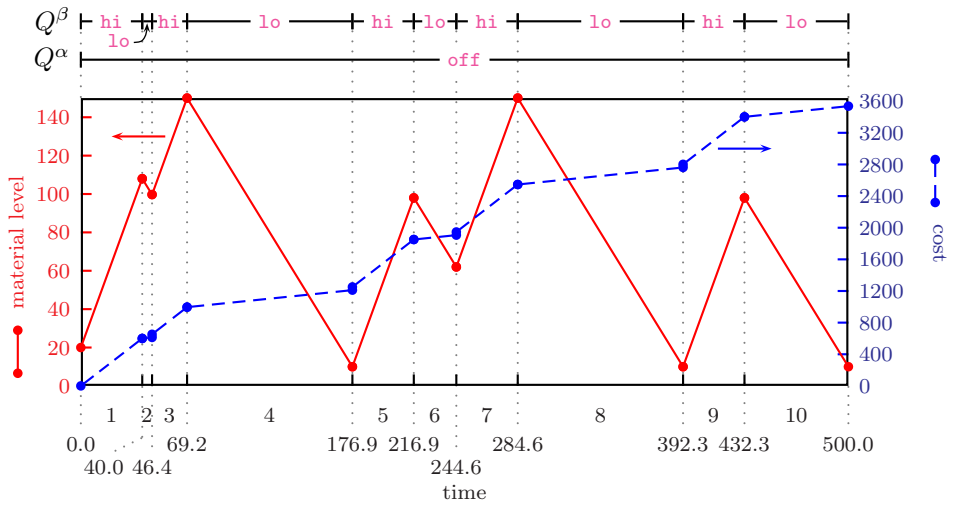
$$\left.\begin{array}{c} yy^a\,(i) + (1 - yyy\,(i)) \geq 1 \qquad \forall a \in Aut \\ \left(\sum\limits_{a \in Aut} (1 - yy^a\,(i))\right) + yyy\,(i) \geq 1 \end{array}\right\} \quad \forall i \in \mathbb{N}\backslash\,\{n\} \tag{9.18b}$$

$$\left.\begin{array}{c} 1 - z^a\,(q,q',i) + y^a\,(q,i) \geq 1 \\ 1 - z^a\,(q,q',i) + y^a\,(q',i+1) \geq 1 \\ 1 - y^a\,(q,i) + 1 - y^a\,(q',i+1) + z^a\,(q,q',i) \geq 1 \end{array}\right\} \quad \forall i \in \mathbb{N}\backslash\,\{n\}\,, \forall a \in Aut, \forall q,q' \in \mathbb{Q}^a \tag{9.18c}$$

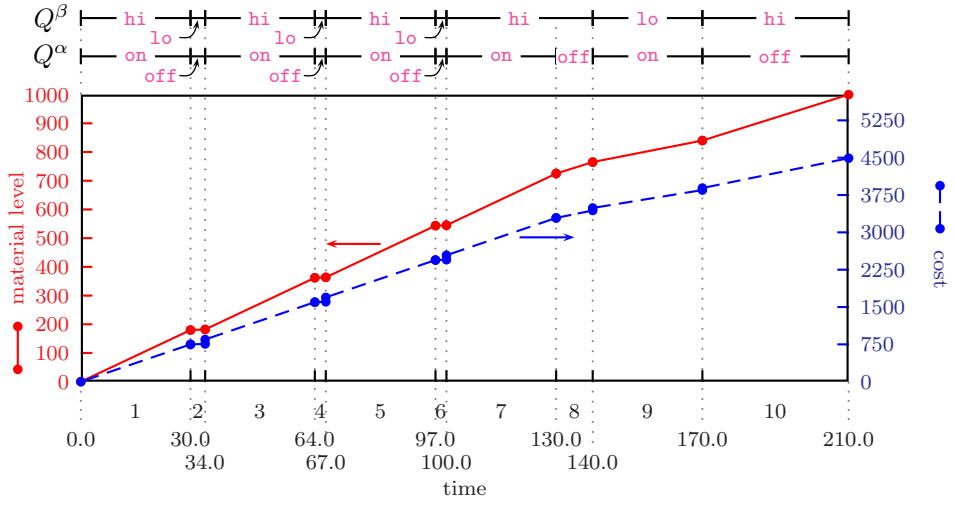
We have now expressed the optimization problem of interest as a linear MIP and thus can apply any of several well-developed algorithms to solve it. For completeness, we present solutions under two different objectives. The above MIP is finally expressed in the GAMS language, distribution 22.0, and solved with the CPLEX algorithm, version 9.1. This requires further modifications to the model, but these are minor syntactic variations and should not be considered model transformations.

Figure 9.3a depicts the optimal trajectory when the objective is to minimize cost, $\Omega = C\,(t_n^e)$. This is easily put into the MIP form $C^e\,(n)$. With this objective, there is essentially no benefit to running the system. However, we are requiring it to run to the time point $(i,t) = (10, 500.0)$. Ideally, processes α and β could remain in their `off` and `lo` modes, where the operating costs are lower. Process α does so, but β has to switch into its `hi` mode because otherwise the material level would fall below 10.0, which is not allowed. The optimal solution obtained is $C\,(t_n^e) = 3537.1$.

Now consider the minimize makespan objective $\Omega = t_n^e$. The idea behind this problem is to complete a job as quickly as possible. This objective is only meaningful if we modify the model a bit. Let us set $F^{\text{out}} = 0.0$, $M^{\min} = 0.0$, $M^{\max} = 1000.0$, $M_0 = 0.0$, and $T^{\max} = 500.0$. The job is to fill the tank, represented by adding the constraint $M\,(t_n^e) = M^{\max}$. This is transformed into the MIP constraint $M^e\,(n) = M^{\max}$. We must still provide an upper bound to the time horizon, but instead of requiring $t_n^e = T^{\max}$, we require only $t_n^e \leq T^{\max}$. This presumes that the job can be fulfilled in at most $T^{\max}$ time units. It can; the optimal solution obtained is $t_n^e = 210.0$. The optimal trajectory, depicted in Figure 9.3b, shows that processes α and β run in their `on` and `hi` modes as much as possible.



(a) Minimize cost. Complete data on page 212.



(b) Minimize makespan. Complete data on page 213.

Figure 9.3: Optimal trajectories under two objectives for switched flow process.

9.4 Formal MP Model

The MP model presented in Section 9.2 is informal. It cannot be understood by a computer. The purpose of Chapters 4, 6, and 7 was to present a formal theory of MP that included modeling constructs required in practice. The following is a formal MP model, i.e. a model expressed in the language we introduced in this work, of the switched flow process.

```
let
  expri n = 5
  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 = 500.0

in

var t : [P * N] -> real
var Matl : [P * N] -> real
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', 1] = 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 .
   (CONJ i:N . clock[c]['e', i] = clock[c]['s', i] + delt[i]),
   (CONJ i:N1 . clock[c]['s', i+1] = clock[c]['e', i] + clockJmp[c][i]),
   clock[c]['s',1] = 0.0
),

(* material level dynamics *)
(CONJ i:N . Matl['e',i] =
   Matl['s',i] + (SUM a:AUT . wmbar[a,i]) + Fout * delt[i]),
(CONJ i:N1 . Matl['s', i+1] = Matl['e', i]),
Matl['s', 1] = Minit,
(CONJ i:N . CONJ p:P . Mmin <= Matl[p,i], Matl[p,i] <= Mmax),
```

```
149
150 (* cost dynamics *)
151 (CONJ i:N . Cost['e',i] = Cost['s',i] + (SUM a:AUT . wcbar[a,i])),
152 (CONJ i:N1 . Cost['s',i+1] = Cost['e',i] + (SUM a:AUT . costJmp[a,i])),
153 Cost['s',1] = 0.0,
154
155 (* definition of wmbar and wcbar *)
156 (CONJ i:N .
157    DISJ q:MODES['alpha'] . (isTrue isInMode['alpha',q,i],
158                         wmbar['alpha',i] = matlFlowa[q] * delt[i],
159                         wcbar['alpha',i] = costFlowa[q] * delt[i])
160 ),
161 (CONJ i:N .
162    DISJ q:MODES['beta'] . (isTrue isInMode['beta',q,i],
163                         wmbar['beta',i] = matlFlowb[q] * delt[i],
164                         wcbar['beta',i] = costFlowb[q] * delt[i])
165 ),
166
167 (* make sure each automaton is only in one mode in each interval *)
168 (CONJ i:N .
169    let expr on = isInMode['alpha', 'on', i]
170        expr off = isInMode['alpha', 'off', i]
171    in isTrue (on or off) and not (on and off)
172    end,
173
174    let expr hi = isInMode['beta', 'hi', i]
175        expr lo = isInMode['beta', 'lo', i]
176    in isTrue (hi or lo) and not (hi and lo)
177    end
178 ),
179
180 (* symmetry breaking *)
181 (CONJ i:N2 . isTrue isDummyEvent[i] ==> isDummyEvent[i+1]),
182 (CONJ i:N1 . (isTrue not isDummyEvent[i]) disj (delt[i+1] = 0.0)),
183
184 (* definition of isDummyTrans *)
185 (CONJ i:N1 . isTrue isDummyTrans['alpha', i] <==>
186    (goesFromTo['alpha','on','on',i] or goesFromTo['alpha','off','off',i])),
187 (CONJ i:N1 . isTrue isDummyTrans['beta', i] <==>
188    (goesFromTo['beta','hi','hi',i] or goesFromTo['beta','lo','lo',i])),
189
190 (* definition of isDummyEvent *)
191 (CONJ i:N1 . isTrue isDummyEvent[i] <==>
192   (isDummyTrans['alpha',i] and isDummyTrans['beta',i])),
193
194 (* definition of goesFromTo *)
195 (CONJ i:N1 . CONJ a:AUT . CONJ q1:MODES[a] . CONJ q2:MODES[a] .
```

```
196     isTrue goesFromTo[a,q1,q2,i] <==> (isInMode[a,q1,i] and isInMode[a,q2,i+1]))
197
198 end
```

9.5 Formal MIP Model

The MIP model of Section 9.3 was derived manually from the general MP model of Section 9.2. Now that we have a formal MP model of the switched flow process, we can apply the compiler defined in Chapter 8 to automatically produce the following MIP.

```
1  let
2    expri n = 5
3    set N = {1, ..., n}
4    set N1 = {1, ..., n-1}
5    set N2 = {1, ..., n-2}
6
7    set CLOCK = {'R','S'}
8    set P = {'s','e'}
9    set AUT = {'alpha','beta'}
10
11   typei funi MODES (a) : [AUT] -> set =
12     case a of
13         'alpha' => {'on','off'}
14       | 'beta'  => {'lo','hi'}
15
16   expr matlFlowa : [MODES['alpha']] -> real =
17     fni q .
18       case _{i.real} q of 'on' => 2.0 | 'off' => 0.0
19
20   expr matlFlowb : [MODES['beta']] -> real =
21     fni q .
22       case _{i.real} q of 'hi' => 4.0 | 'lo' => 0.5
23
24   expr costFlowa : [MODES['alpha']] -> real =
25     fni q .
26       case _{i.real} q of 'on' => 10.0 | 'off' => 0.0
27
28   expr costFlowb : [MODES['beta']] -> real =
29     fni q .
30       case _{i.real} q of 'hi' => 15.0 | 'lo' => 2.0
31 in
32
33 var t:[P * N] -> real
34 var Matl:[P * N] -> real
35 var Cost:[P * N] -> real
36 var isInMode:[a:AUT * MODES[a] * N] -> [0, 1]
```

```
var clock:[CLOCK] -> [P * N] -> <0.0, 1000.0>

min Cost['e',5] 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] -> [0, 1]
exists isDummyTrans:[AUT * N1] -> [0, 1]
exists isDummyEvent:[N1] -> [0, 1]

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',5] = 500.0,
t['s',1] = 0.0,


(* transformation of disjunction over modes of alpha *)
(CONJ i:N
   exists y:[MODES['alpha']] -> [0, 1]
   exists isInMode1:[MODES['alpha']] -> [a:AUT * MODES[a] * N] -> [0, 1]
   exists clock1:[MODES['alpha']] -> [CLOCK] -> [P * N] -> <0.0, 1000.0>

   (CONJ d:CLOCK CONJ d0:(P * N)
      clock[d][d0] = SUM q:MODES['alpha'] clock1[q][d][d0]),
   (CONJ d:(a:AUT * MODES[a] * N)
      isInMode[d] = SUM q:MODES['alpha'] isInMode1[q][d]),
   SUM q:MODES['alpha'] y[q] = 1,
   (CONJ q:MODES['alpha']
      (CONJ d:(a:AUT * MODES[a] * N)
         0 * y[q] <= isInMode1[q][d],
         isInMode1[q][d] <= 1 * y[q]),
      (CONJ d:CLOCK CONJ d0:(P * N)
         0.0 * y[q] <= clock1[q][d][d0],
         clock1[q][d][d0] <= 1000.0 * y[q]),
      isInMode1[q]['alpha',q,i] >= 1 * y[q],
      CONJ p:P clock1[q]['R'][p,i] <=
                  (case _{i.real} q of 'off' => (1000.0 * y[q]) | 'on' => (30.0 * y[q])))
),

```

```
84
85  (* transformation of disjunction over transitions of alpha *)
86  (CONJ i:N1
87     exists y:[{1, ..., 3}] -> [0, 1]
88     exists isDummyTrans1:[{1, ..., 3}] -> [AUT * N1] -> [0, 1]
89     exists goesFromTo1:[{1, ..., 3}] ->
90                           [a:AUT * MODES[a] * MODES[a] * N1] -> [0, 1]
91     exists costJmp1:[{1, ..., 3}] -> [AUT * N] -> <0.0, 50.0>
92     exists clockJmp1:[{1, ..., 3}] -> [{'R','S'}] -> [N] -> <~1000.0, 0.0>
93     exists clock1:[{1, ..., 3}] -> [{'R','S'}] -> [P * N] -> <0.0, 1000.0>
94
95     (CONJ d:CLOCK CONJ d0:(P * N)
96        clock[d][d0] = SUM i0:{1, ..., 3} clock1[i0][d][d0]),
97     (CONJ d:CLOCK CONJ d0:N
98        clockJmp[d][d0] = SUM i0:{1, ..., 3} clockJmp1[i0][d][d0]),
99     (CONJ d:(AUT * N)
100       costJmp[d] = SUM i0:{1, ..., 3} costJmp1[i0][d]),
101    (CONJ d:(a:AUT * MODES[a] * MODES[a] * N1)
102       goesFromTo[d] = SUM i0:{1, ..., 3} goesFromTo1[i0][d]),
103    (CONJ d:(AUT * N1)
104       isDummyTrans[d] = SUM i0:{1, ..., 3} isDummyTrans1[i0][d]),
105    SUM i0:{1, ..., 3} y[i0] = 1,
106    (CONJ i0:{1, ..., 3}
107       (CONJ d:CLOCK CONJ d0:(P * N)
108          0.0 * y[i0] <= clock1[i0][d][d0],
109          clock1[i0][d][d0] <= 1000.0 * y[i0]),
110       (CONJ d:(AUT * N)
111          0.0 * y[i0] <= costJmp1[i0][d],
112          costJmp1[i0][d] <= 50.0 * y[i0]),
113       (CONJ d:CLOCK CONJ d0:N
114          ~1000.0 * y[i0] <= clockJmp1[i0][d][d0],
115          clockJmp1[i0][d][d0] <= 0.0 * y[i0]),
116       (CONJ d:(a:AUT * MODES[a] * MODES[a] * N1)
117          0 * y[i0] <= goesFromTo1[i0][d],
118          goesFromTo1[i0][d] <= 1 * y[i0]),
119       (CONJ d:(AUT * N1)
120          0 * y[i0] <= isDummyTrans1[i0][d],
121          isDummyTrans1[i0][d] <= 1 * y[i0]),
122       (case _{i.prop} i0 of
123             1 => goesFromTo1[i0]['alpha','on','off',i] >= 1 * y[i0],
124                  costJmp1[i0]['alpha',i] = 0.0 * y[i0],
125                  clockJmp1[i0]['R'][i] = ~ clock1[i0]['R']['e',i]
126           | 2 => goesFromTo1[i0]['alpha','off','on',i] >= 1 * y[i0],
127                  clock1[i0]['R']['e',i] >= 2.0 * y[i0],
128                  costJmp1[i0]['alpha',i] = 50.0 * y[i0],
129                  clockJmp1[i0]['R'][i] = ~ clock1[i0]['R']['e',i]
130           | 3 => isDummyTrans1[i0]['alpha',i] >= 1 * y[i0],
```

```
                costJmp1[i0]['alpha',i] = 0.0 * y[i0],
                clockJmp1[i0]['R'][i] = 0.0 * y[i0]
      )
   )
),


(* transformation of disjunction over modes of beta *)
(CONJ i:N
   exists y:[MODES['beta']] -> [0, 1]
   exists isInMode1:[MODES['beta']] -> [a:AUT * MODES[a] * N] -> [0, 1]
   exists clock1:[MODES['beta']] -> [CLOCK] -> [P * N] -> <0.0, 1000.0>

   (CONJ d:CLOCK CONJ d0:(P * N)
      clock[d][d0] = SUM q:MODES['beta'] clock1[q][d][d0]),
   (CONJ d:(a:AUT * MODES[a] * N)
      isInMode[d] = SUM q:MODES['beta'] isInMode1[q][d]),
   SUM q:MODES['beta'] y[q] = 1,
   (CONJ q:MODES['beta']
      (CONJ d:(a:AUT * MODES[a] * N)
         0 * y[q] <= isInMode1[q][d],
         isInMode1[q][d] <= 1 * y[q]),
      (CONJ d:CLOCK CONJ d0:(P * N)
         0.0 * y[q] <= clock1[q][d][d0],
         clock1[q][d][d0] <= 1000.0 * y[q]),
      isInMode1[q]['beta',q,i] >= 1 * y[q],
      CONJ p:P clock1[q]['S'][p,i] <=
                  (case _{i.real} q of 'hi' => (40.0 * y[q]) | 'lo' => (1000.0 * y[q])))
),


(* transformation of disjunction over transitions of beta *)
(CONJ i:N1
   exists y:[{1, ..., 3}] -> [0, 1]
   exists isDummyTrans1:[{1, ..., 3}] -> [AUT * N1] -> [0, 1]
   exists goesFromTo1:[{1, ..., 3}] ->
                         [a:AUT * MODES[a] * MODES[a] * N1] -> [0, 1]
   exists costJmp1:[{1, ..., 3}] -> [AUT * N] -> <0.0, 50.0>
   exists clockJmp1:[{1, ..., 3}] -> [CLOCK] -> [N] -> <~1000.0, 0.0>
   exists clock1:[{1, ..., 3}] -> [CLOCK] -> [P * N] -> <0.0, 1000.0>

   (CONJ d:CLOCK CONJ d0:(P * N)
      clock[d][d0] = SUM i0:{1, ..., 3} clock1[i0][d][d0]),
   (CONJ d:CLOCK CONJ d0:N
      clockJmp[d][d0] = SUM i0:{1, ..., 3} clockJmp1[i0][d][d0]),
   (CONJ d:(AUT * N)
      costJmp[d] = SUM i0:{1, ..., 3} costJmp1[i0][d]),
```

```
   (CONJ d:(a:AUT * MODES[a] * MODES[a] * N1)
      goesFromTo[d] = SUM i0:{1, ..., 3} goesFromTo1[i0][d]),
   (CONJ d:(AUT * N1)
      isDummyTrans[d] = SUM i0:{1, ..., 3} isDummyTrans1[i0][d]),
   SUM i0:{1, ..., 3} y[i0] = 1,
   (CONJ i0:{1, ..., 3}
      (CONJ d:CLOCK CONJ d0:(P * N)
         0.0 * y[i0] <= clock1[i0][d][d0],
         clock1[i0][d][d0] <= 1000.0 * y[i0]),
      (CONJ d:(AUT * N)
         0.0 * y[i0] <= costJmp1[i0][d],
         costJmp1[i0][d] <= 50.0 * y[i0]),
      (CONJ d:CLOCK CONJ d0:N
         ~1000.0 * y[i0] <= clockJmp1[i0][d][d0],
         clockJmp1[i0][d][d0] <= 0.0 * y[i0]),
      (CONJ d:(a:AUT * MODES[a] * MODES[a] * N1)
         0 * y[i0] <= goesFromTo1[i0][d],
         goesFromTo1[i0][d] <= 1 * y[i0]),
      (CONJ d:(AUT * N1)
         0 * y[i0] <= isDummyTrans1[i0][d],
         isDummyTrans1[i0][d] <= 1 * y[i0]),
      (case _{i.prop} i0 of
            1 => goesFromTo1[i0]['beta','hi','lo',i] >= 1 * y[i0],
                 costJmp1[i0]['beta',i] = 0.0 * y[i0],
                 clockJmp1[i0]['S'][i] = ~ clock1[i0]['S']['e',i]
          | 2 => goesFromTo1[i0]['beta','lo','hi',i] >= 1 * y[i0],
                 clock1[i0]['S']['e',i] >= 3.0 * y[i0],
                 costJmp1[i0]['beta',i] = 40.0 * y[i0],
                 clockJmp1[i0]['S'][i] = ~ clock1[i0]['S']['e',i]
          | 3 => isDummyTrans1[i0]['beta',i] >= 1 * y[i0],
                 costJmp1[i0]['beta',i] = 0.0 * y[i0],
                 clockJmp1[i0]['S'][i] = 0.0 * y[i0]
      )
   )
),


(* clock dynamics *)
(CONJ c:CLOCK
   (CONJ i:N clock[c]['e',i] = clock[c]['s',i] + delt[i]),
   (CONJ i:N1 clock[c]['s',(i + 1)] = clock[c]['e',i] + clockJmp[c][i]),
   clock[c]['s',1] = 0.0
),

(* material level dynamics *)
(CONJ i:N Matl['e',i] = Matl['s',i] + SUM a:AUT wmbar[a,i] + (1.8 * delt[i])),
(CONJ i:N1 Matl['s',(i + 1)] = Matl['e',i]),
```

```
Matl['s',1] = 20.0,
(CONJ i:N CONJ p:P
   10.0 <= Matl[p,i],
   Matl[p,i] <= 150.0),

(* cost dynamics *)
(CONJ i:N Cost['e',i] = Cost['s',i] + SUM a:AUT wcbar[a,i]),
(CONJ i:N1 Cost['s',(i + 1)] = Cost['e',i] + SUM a:AUT costJmp[a,i]),
Cost['s',1] = 0.0,

(* definition of wmbar and wcbar *)
(CONJ i:N
   exists y:[MODES['alpha']] -> [0, 1]
   exists wmbar1:[MODES['alpha']] -> [(AUT * N)] -> <0.0, 2000.0>
   exists wcbar1:[MODES['alpha']] -> [(AUT * N)] -> <0.0, 7500.0>
   exists isInMode1:[MODES['alpha']] -> [(a:AUT * MODES[a] * N)] -> [0, 1]
   exists delt1:[MODES['alpha']] -> [N] -> <0.0, 500.0>

   (CONJ d:N
      delt[d] = SUM q:MODES['alpha'] delt1[q][d]),
   (CONJ d:(a:AUT * MODES[a] * N)
      isInMode[d] = SUM q:MODES['alpha'] isInMode1[q][d]),
   (CONJ d:(AUT * N)
      wcbar[d] = SUM q:MODES['alpha'] wcbar1[q][d]),
   (CONJ d:(AUT * N)
      wmbar[d] = SUM q:MODES['alpha'] wmbar1[q][d]),
   SUM q:MODES['alpha'] y[q] = 1,
   (CONJ q:MODES['alpha']
      (CONJ d:(a:AUT * MODES[a] * N)
         0 * y[q] <= isInMode1[q][d],
         isInMode1[q][d] <= 1 * y[q]),
      (CONJ d:N
         0.0 * y[q] <= delt1[q][d],
         delt1[q][d] <= 500.0 * y[q]),
      (CONJ d:(AUT * N)
         0.0 * y[q] <= wmbar1[q][d],
         wmbar1[q][d] <= 2000.0 * y[q]),
      (CONJ d:(AUT * N)
         0.0 * y[q] <= wcbar1[q][d],
         wcbar1[q][d] <= 7500.0 * y[q]),
       isInMode1[q]['alpha',q,i] >= 1 * y[q],
       wmbar1[q]['alpha',i] = matlFlowa[q] * delt1[q][i],
       wcbar1[q]['alpha',i] = costFlowa[q] * delt1[q][i]
   )
),

(CONJ i:N
```

```
   exists y:[MODES['beta']] -> [0, 1]
   exists wmbar1:[MODES['beta']] -> [(AUT * N)] -> <0.0, 2000.0>
   exists wcbar1:[MODES['beta']] -> [(AUT * N)] -> <0.0, 7500.0>
   exists isInMode1:[MODES['beta']] -> [(a:AUT * MODES[a] * N)] -> [0, 1]
   exists delt1:[MODES['beta']] -> [N] -> <0.0, 500.0>

   (CONJ d:N
      delt[d] = SUM q:MODES['beta'] delt1[q][d]),
   (CONJ d:(a:AUT * MODES[a] * N)
      isInMode[d] = SUM q:MODES['beta'] isInMode1[q][d]),
   (CONJ d:(AUT * N)
      wcbar[d] = SUM q:MODES['beta'] wcbar1[q][d]),
   (CONJ d:(AUT * N)
      wmbar[d] = SUM q:MODES['beta'] wmbar1[q][d]),
   SUM q:MODES['beta'] y[q] = 1,
   (CONJ q:MODES['beta']
      (CONJ d:(a:AUT * MODES[a] * N)
         0 * y[q] <= isInMode1[q][d],
         isInMode1[q][d] <= 1 * y[q]),
      (CONJ d:N
         0.0 * y[q] <= delt1[q][d],
         delt1[q][d] <= 500.0 * y[q]),
      (CONJ d:(AUT * N)
         0.0 * y[q] <= wmbar1[q][d],
         wmbar1[q][d] <= 2000.0 * y[q]),
      (CONJ d:(AUT * N)
         0.0 * y[q] <= wcbar1[q][d],
         wcbar1[q][d] <= 7500.0 * y[q]),
      isInMode1[q]['beta',q,i] >= 1 * y[q],
      wmbar1[q]['beta',i] = matlFlowb[q] * delt1[q][i],
      wcbar1[q]['beta',i] = costFlowb[q] * delt1[q][i]
   )
),

(* make sure each automaton is only in one mode in each interval *)
(CONJ i:N
   isInMode['alpha','on',i] + isInMode['alpha','off',i] >= 1,
   1 - isInMode['alpha','on',i] + 1 - isInMode['alpha','off',i] >= 1,
   isInMode['beta','hi',i] + isInMode['beta','lo',i] >= 1,
   1 - isInMode['beta','hi',i] + 1 - isInMode['beta','lo',i] >= 1
),

(* symmetry breaking *)
(CONJ i:N2 1 - isDummyEvent[i] + isDummyEvent[i + 1] >= 1),
(CONJ i:N1
   exists y:[{1, ..., 2}] -> [0, 1]
   exists isDummyEvent1:[{1, ..., 2}] -> [N1] -> [0, 1]
```

```
319    exists delt1:[{1, ..., 2}] -> [N] -> <0.0, 500.0>
320
321    (CONJ d:N
322       delt[d] = SUM i0:{1, ..., 2} delt1[i0][d]),
323    (CONJ d:N1
324       isDummyEvent[d] = SUM i0:{1, ..., 2} isDummyEvent1[i0][d]),
325    SUM i0:{1, ..., 2} y[i0] = 1,
326    (CONJ i0:{1, ..., 2}
327       (CONJ d:N
328          0.0 * y[i0] <= delt1[i0][d],
329          delt1[i0][d] <= 500.0 * y[i0]),
330       (CONJ d:N1
331          0 * y[i0] <= isDummyEvent1[i0][d],
332          isDummyEvent1[i0][d] <= 1 * y[i0]),
333       (case _{i.prop} i0 of
334             1 => 1 * y[i0] - isDummyEvent1[i0][i] >= 1 * y[i0]
335           | 2 => delt1[i0][i + 1] = 0.0 * y[i0]
336       )
337    )
338 ),
339
340 (* definition of isDummyTrans *)
341 (CONJ i:N1
342    1 - isDummyTrans['alpha',i] +
343       goesFromTo['alpha','on','on',i] + goesFromTo['alpha','off','off',i] >= 1,
344    1 - goesFromTo['alpha','on','on',i] + isDummyTrans['alpha',i] >= 1,
345    1 - goesFromTo['alpha','off','off',i] + isDummyTrans['alpha',i] >= 1
346 ),
347 (CONJ i:N1
348    1 - isDummyTrans['beta',i] +
349       goesFromTo['beta','hi','hi',i] + goesFromTo['beta','lo','lo',i] >= 1,
350    1 - goesFromTo['beta','hi','hi',i] + isDummyTrans['beta',i] >= 1,
351    1 - goesFromTo['beta','lo','lo',i] + isDummyTrans['beta',i] >= 1
352 ),
353
354 (* definition of isDummyEvent *)
355 (CONJ i:N1
356    isDummyTrans['alpha',i] + 1 - isDummyEvent[i] >= 1,
357    isDummyTrans['beta',i] + 1 - isDummyEvent[i] >= 1,
358    isDummyEvent[i] +
359       1 - isDummyTrans['alpha',i] + 1 - isDummyTrans['beta',i] >= 1
360 ),
361
362 (* definition of goesFromTo *)
363 (CONJ i:N1 CONJ a:AUT CONJ q1:MODES[a] CONJ q2:MODES[a]
364    isInMode[a,q1,i] + 1 - goesFromTo[a,q1,q2,i] >= 1,
365    isInMode[a,q2,(i + 1)] + 1 - goesFromTo[a,q1,q2,i] >= 1,
```

```
366     goesFromTo[a,q1,q2,i] +
367         1 - isInMode[a,q1,i] + 1 - isInMode[a,q2,(i + 1)] >= 1
368 )
369
370 end
```

9.6 Results

The example in this chapter demonstrates the value of the novel modeling frameworks introduced in this work. It is clear that the LCCA model is more intuitive than the general MP model and much simpler than the MIP model. At a glance, this evident from the fact the MIP model is much longer, but the true difficulty is even greater than this suggests. The constraints of MIP are much more complex, and even experts would find it difficult to think of them directly. Our MP language supports more intuitive constraint forms, and our transformation procedure automatically generates the equivalent MIP. This is a large savings in labor, does not require expertise in the transformation methods, and minimizes the chance of error.

Chapter 10

Conclusions

We conclude by providing a summary of the main results from each chapter. Following that we provide an assessment of our contributions, and, lastly, discuss directions for future research.

10.1 Summary

Chapter 2 Modeling Hybrid Systems

- Defined the linear coupled component automata (LCCA) modeling framework.
- Finite domain constraints supported. Allows easily expressing resource constraints between different processes.
- Explicit support for modeling dynamics. For example, guards make it easy to restrict future behavior based on current state.
- Enforcement of variable localization encourages modular modeling. Change to any process requires changing only corresponding component automata. Other automata guaranteed to be unaffected.
- Showed that LCCA models are significantly easier to pose than MIP models, the current standard for modeling discrete-continuous process systems. LCCA models are smaller, easier to modify, and more intuitive.

Chapter 3 Optimizing Hybrid Systems

- Defined a procedure for systematically transforming LCCA models to MIP constraints.
- Methods provided include elimination of infinite quantifiers, removal of variable arguments, and transformation of finite domain constraints to Boolean constraints.
- It is shown that automata can be represented with disjunctive constraints.
- Reformulation is likely efficient. Symmetry breaking constraints added to remove redundant solutions. Convex hull method used for translating disjunctive constraints.

- It is now possible to formulate the LCCA model more easily, mechanically generate the equivalent but complex MIP model, and solve the problem with well-developed existing MIP algorithms.

Chapter 4 Logical Formulation of Mathematical Programs

- Employed type theory to define a logic of mathematical programming.
- Constraints and whole programs can be treated as mathematical objects, which allows defining operations on constraint and program spaces.
- Theory leads directly to software implementation.
- Definition of MP is simultaneously a language for naturally expressing an MP, in contrast to matrix-based definitions.
- Boolean expressions, disjunctive constraints, and existential quantifiers supported.
- Software is trustworthy because its implementation is defined mathematically.
- Software development is greatly simplified.
- A clearer understanding is provided for what an MP algorithm should be. The solution should be a proof describing the program's execution. Explained that current practice seems to follow a mixture of the classical and constructive traditions of proof.
- Certain error messages are shown to be explanations of the progress or lack thereof of theorem provers, which allows rigorous treatment of the seemingly non-mathematical matter of messaging.

Chapter 5 Compiling Mathematical Programs

- A binary relation between MP and MIP is defined. Interpreted algorithmically, this is a procedure for transforming an MP to a MIP.
- Two main procedures defined are: conversion to conjunctive normal form, and reformulation of disjunctive constraints.
- Our definitions formalize methods previously defined in less mathematical terms and on only canonical forms.
- Precondition on compiler defined. Only MPs that can be expressed as MIPs are transformed.
- Theory leads directly to automation of a procedure that has long been performed manually.

Chapter 6 Index Sets

- Defined a finitary logic within which index sets can be defined in intuitive ways.
- Allows index sets to be treated as mathematical objects, rather than mere notational conveniences eliminated at parse time.
- Type system defined semantically. More complex index sets are accepted as well-formed.
- Advanced features supported include functions mapping to the space of all index sets, dependent functions and products, case expressions and case types.

Chapter 7 Indexed Mathematical Programs

- Combined theories of Chapter 6 and 7 to provide a logic of indexed mathematical programs.
- All contributions of Chapter 7 extended to indexed programs.
- Indexing allows retention of problem structure and maintains small program size.
- Theory makes it possible to develop genuinely novel algorithms, which are parametric on the indexed structure.

Chapter 8 Compiling Indexed Mathematical Programs

- All contributions of Chapter 5 extended to indexed programs.
- Definition of conjunctive normal form defined in the presence of higher-order types.
- Compiler operates over indices. This both makes it faster and returns a program with knowledge of the problem structure retained.

10.2 Assessment

In the Introduction chapter, we critiqued current definitions of mathematical programming from both a mathematical and software perspective. Our claim was that a type theoretic formulation improves on both and unifies the two efforts.

Our software firstly provides a computer language for expressing programs in a natural way. We compare our language with four of the leading MP languages in use: GAMS, AMPL, OPL, and Mosel. We do not compare to AIMMS because, although it provides several enhancements, its modeling language is based on GAMS. Also, SIMPL (Aron et al., 2004) provides several interesting language features, but they regard coupling modeling and algorithmic constructs, which is outside the scope of our language.

Let us call the language developed in this work TyL, which stands for "typed language" to emphasize its type theoretic formulation. TyL is the language whose abstract syntax p, given by definition (7.5), must satisfy the judgement p PROG, defined by rule (7.15), and whose meaning is defined in Section 7.4. TyL also refers to the concrete syntax features

discussed in Appendix C, and the various operations we have defined for it, most notably the compiler (8.42).

Table 10.1 lists language features in the first column, and a check mark indicates the languages in which that feature is available. The list is biased towards modeling features addressed in our work. The languages we compare with are all parts of commercial products and have many features not listed. Nonetheless, the table shows that TyL provides many features not available in them. Often we credited the other languages with features that are not supported as thoroughly or rigorously as in TyL.

Language Feature	GAMS	AMPL	OPL[a]	Mosel[b]	TyL
Boolean expressions				✓	✓
disjunctive constraints			✓	✓	✓
index type functions		✓	✓		✓
index set operations		✓	✓	✓	
expression functions			✓	✓	✓
proposition functions					✓
functions returning tuples					✓
dependent products		✓	✓		✓
dependent functions					✓
case expressions		✓	✓	✓	✓
index case types		✓			✓
case propositions					✓
conditional index sets	✓	✓	✓		
syntax checker	✓	✓	✓	✓	✓
type system					✓
computation on reals					
Boolean to IP compiler					✓
disjunction to MIP compiler			✓		✓

[a]Based on OPL Studio 3.7.1. Version 4.0 has reduced features.
[b]Including XPress-Kalis.

Table 10.1: Comparison of features in several modeling languages.

Of the previous languages, only Mosel provides the Boolean data type. It does not however distinguish between Boolean and propositional truth, as we do in TyL. Since only Boolean expressions can be converted to pure integer inequalities, the Boolean expression compiler available in TyL would be difficult to provide in Mosel. It has no mechanism for even knowing which class of expressions the procedure could be applied to.

The situation is similar in OPL, which treats truth and falsehood with the integers 1 and 0. Even propositions are treated as integers. If $x = 3$ is true, the whole equation is treated as the integer 1. The expression $1 + (x = 3)$ is thus legal in OPL. Such a design decision probably stems from two widely held misconceptions: the integers 1 and 0 can be used to represent truth and falsehood and that there is only one notion of truth. Even if one manages to work out a consistent theory with these decisions, it is not beneficial to do so. The distinction between expressions and propositions is relied upon by many mathematical methods. It is not clear to us how the disjunctive proposition compiler provided in TyL could be defined within a framework that does not make this distinction.

AMPL and OPL have a limited form of the dependent product type. Interestingly,

only flat products are available, meaning there cannot be a product of products. Placing this restriction in a language formulated type theoretically would actually require additional work. Such generalizations are obtained for free. The basic reason is neither language treats products as a type. What they call types are only the base types, such as real. Higher types, those composed of other types, are formulated separately. Our formulation of TyL makes it clear that these ideas can be unified, simultaneously simplifying the theory and producing a more powerful language.

Although functions of various forms are available in several of the previous languages, it is clear that none have provided a robust functional theory. Functions are not first class entities; they cannot be passed as arguments to or returned by other functions. There cannot be tuples of functions, nor can functions return tuples. These are elementary features in TyL.

We formalized indexed expressions as being of function type, where the domain is an index set and the codomain an MP type. The statement x_i is thus interpreted as a function x being applied to the argument i. We are also able to easily translate this idea to the level of propositions, providing indexed propositions.

Two important indexing features are not available in TyL. There are no set operations, such as union and intersection, and it is not possible to define sets conditionally. An example of a conditional set is $J = \{i \in I \mid i \leq 10\}$. There should not be any significant challenges in extending TyL with these features. Dependent types do already provide some equivalent features. The second component of the TyL index set $i : I \times [i, 10]$ is effectively the set J.

Neither TyL nor any of the other languages have resolved the issue of computing on the reals. A straightforward implemenation of the semantics of TyL is not possible. At present, the practical option is to treat reals as floats in an algorithm. There is then no guarantee that the algorithm is giving correct answers, but this is the situation in all software. Correct real computations are a subject of much research by others.

All language implementations have some kind of syntax checker. This is a basic requirement. However, we hesitate to claim that the previous languages have defined a type system. Typing judgements are formal set relations, which have not been provided for the previous languages. The judgement p PROG does not simply check that an input syntax is well-formed; it is the definition of what a program is.

The broad point is that TyL is more trustworthy and more expressive (in those features we addressed), while simultaneously having a less complex software design. The reason is we have provided a theoretical basis for software development by treating it as a problem in type theory.

This equates design of a language with invention of a mathematical system. Thus, our formulation of TyL is simultaneously an alternative definition for mathematical programs. Table 10.2 compares the previous numerical algebraic formulation of MP, as in definition (1.4) on page 9, with our type theoretic formulation.

Definition (1.4) is compact and simple, and it of course makes it readily apparent how all the results of numerical mathematics can be employed to study and solve an MP. Our definition of TyL spans a few chapters. This is partly because we take care of many details that are taken for granted in the compact notation of linear algebra, but largely it is because

	Definition Style	
	Matrix\Numerical	Type Theory\Logical
simple compact definition	✓	
numerical methods applicable	✓	✓
knowledge retention mechanisms		✓
extensible		✓
non-numerical operations		✓
practical modeling system		✓
varied algorithms easily employed		✓
leads to computer implementation		✓

Table 10.2: Comparison of definitions of mathematical programs.

many more features are provided.

The logical formulation in no way excludes the application of any existing analyses. Definition (1.4) can be thought of as a special case of TyL. It is a canonical form, and we could (but did not yet) provide a method for transforming a TyL program to this form. Expressions of type $\sigma \to \tau$ and $\sigma_1 \times \sigma_2 \to \tau$ can be interpreted as vectors and matrices.

TyL has many more types than this. These provide the ability to express programs more naturally, and more efficient representations are possible. Dependent index sets allow representing non-rectangular matrices; so only the necessary elements need be declared. Also, they extend the theory of MP in ways that a matrix based definition could not, for example to include Boolean propositions. It should be clear that further extensions to TyL would be straightforward. Enumerated types, of the form available in the indexing logic, could be readily added to the MP types. (The enumerated type is available in OPL, and AMPL has plans to introduce it.)

A richer syntax serves algorithms also. Each syntactic form represents knowledge in a distinct manner. The syntax of TyL maintains all the knowledge provided by a modeler. A constraint indexed over a set S is a single constraint in TyL, and an algorithm could operate on it as such. In contrast, indexing has till now been treated as a notational convenience. An indexed constraint is expanded out to numerous unindexed constraints. This discards the knowledge that all these constraints actually have an identical structure, and now there are $|S|$ constraints that must be operated on, instead of one.

Although we did not provide algorithms for solving an MP, we did provide some other operations on MP, the compiler for example. Since indices are retained, the generated programs are compact and the compiler runs faster. Expanding an indexed disjunction or Boolean expression would cause the same operations to be performed many more times.

Above, we stated that a type theoretic formulation of a computer language is simultaneously a definition of a mathematical system. Conversely, a type theoretic definition of MP immediately leads to a practical modeling language. The compact matrix definition of an MP clearly does not provide this. Models are never written in this form.

The reason for several of the above benefits is that our formulation of TyL treats indices, expressions, propositions, and programs as rigorously as current methods treat numbers. It is a central notion of mathematical logic that a standard of rigor can be established for all manner of constructs.

This is necessary to define the compilation of MP to MIP. A compiler is an operation whose domain and codomain are program spaces. A mathematical definition of this operation requires these spaces to be defined first. The conversion of Boolean expressions to integer inequalities is considered established knowledge. Similarly, the convex hull transformation of a disjunctive constraint was provided many years ago. It is not surprising however that none of the MP software we have reviewed provide these transformations, although their importance is well known. Automation requires a mathematical definition of these methods on programs as written in practice. The binary relation $p^{\text{MP}} \overset{\text{PROG}}{\longmapsto} p^{\text{MIP}}$ is the first such definition.

Our definition of the semantics of MP is a first step towards providing an algorithmic theory of MP. Current concepts of what an algorithm is refer to the result of an algorithm, i.e. that the value returned should have a certain property. This however does not classify the methods that might be used to get to this result. We discussed that the methods sought are often constructive proofs.

10.3 Future Work

There are some important features our language would benefit from. As mentioned above, set operations, such as union and intersection, are certainly important and should be provided. Also, conditional index sets should be supported, which requires introducing propositions into the indexing logic.

Indexed versions of most operators were provided in the logic of indexed MP, but Boolean conjunction and disjunction were excluded. Adding these into the language is actually straightforward, but the compiler becomes difficult to define. Consider the conversion of $\mathtt{OR}_{i:\sigma}\left(\mathtt{AND}_{i':\sigma'}\, e\right)$ to conjunctive normal form. One possible answer is

$$\mathtt{AND}_{f:(i:\sigma\to\sigma')}\left(\mathtt{OR}_{i:\sigma}\ \left\{f\left(i\right)/i'\right\} e\right)$$

However, this result is not expressible in our language because we do not support index sets of the form $i:\sigma\to\sigma'$. Another possible answer is

$$\mathtt{AND}_{i:\sigma}\left(s_i \vee \mathtt{AND}_{i':\sigma'}\, e\right)$$

where the constraint $\mathtt{not}\left(\mathtt{AND}_{i:\sigma}\, s_i\right)$ is also required. However, this statement is incomplete because the slack variable s_i was never introduced; our language has no mechanism for introducing variables into Boolean expressions. In either case, some extensions are necessary to transform indexed Boolean operators.

Our treatment of semantics was not as formal as that of the MP language. Much work remains to be done here. Firstly, a language of proof terms is needed. This would serve as a definition of the space of algorithms that could be applied to solve an MP. An implementation will only be possible when real numbers are treated constructively. Absent that, it will be important to develop a theory of approximate program execution, allowing a quantitative assessment of how close a found solution is to the true solution.

Differential equations would enhance the language's utility well beyond the present focus

of mathematical programming. The compiler technology that becomes available with a type theoretic formulation would be very valuable because discretization of differential equations can be thought of as a program transformation. It is a mapping from the space of differential equations to the space of algebraic equations. Also, a logic of differential equations is a prerequisite to a logic of hybrid systems.

Finally, a very promising application of type theory is to define domain specific logics. For example, it should be possible to design a logic in which "molecule", "atom", and other chemical concepts are intrinsic notions. Then, we would have a mathematical system specifically designed to express chemical phenomena. The gap between physics and mathematics would be truly narrowed.

Appendix A

Reformulating Mathematical Programs

We use the term mathematical program (MP) to refer to the most general class of problems studied under this name. There are commonly used names referring to frameworks with only some features supported. In Figure A.1 we depict the relationship between these. A linear program (LP) allows equations and inequalities on the reals (also an objective function but our focus is on the constraints). When the constraints are allowed to be nonlinear, the term nonlinear program (NLP) is used. A mixed-integer program (MIP) allows requiring certain variables to be integer. If only linear constraints are allowed, it is called a mixed-integer linear program (MILP).

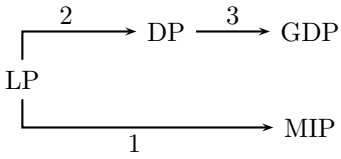


Figure A.1: Relationships between specialized MP frameworks. $A \xrightarrow{n} B$ means language B enhances language A with feature n, where n can be (1) integer variables, (2) disjunctive constraints, (3) Boolean logic.

In LP, we have a system of inequalities, by which is meant a conjunction of inequalities. A disjunctive program (DP) also allows disjunctive constraints. These are in the form

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

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. Superscripts are used to denote multiple entities of a similar type, e.g. b^1 and b^2 are two different $m \times 1$ vectors, and subscripts to access components of matrices or vectors, e.g. b^2_1 is the first element of b^2, and b^1_2 is the second element of b^1. Instead of requiring all inequalities to hold, as in LP, a disjunctive constraint requires just one of several to hold.

Finally, a generalized disjunctive program (GDP) also allows Boolean expressions. Each disjunct of a disjunction can also be marked with a Boolean variable indicating that that disjunct is satisfied only if the corresponding Boolean variable is.

By general mathematical program, we refer to the system in which all of these features are supported in complete generality. Nested disjunctions are allowed, as are integer variables, and Boolean expressions of any form can be used within disjunctions and elsewhere.

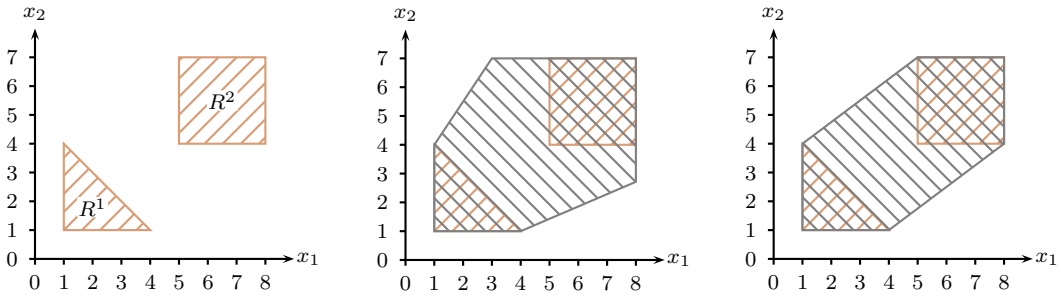
In this Appendix, we review methods for converting certain constraint forms. We focus on programs with linear constraints, but some of the methods are applicable to the non-linear case. Our review is conceptual. Formal definitions of MP frameworks and of the transformation of a general MP to a pure MIP is the subject of Chapters 4–8.

A.1 Simple Reformulations

Consider the disjunctive constraint

$$\underbrace{\begin{bmatrix} x_1 \geq 1 \\ x_2 \geq 1 \\ x_1 + x_2 \leq 5 \end{bmatrix}}_{R^1} \vee \underbrace{\begin{bmatrix} 5 \leq x_1 \leq 8 \\ 4 \leq x_2 \leq 7 \end{bmatrix}}_{R^2} \tag{A.2}$$

which we use as an example throughout. The feasible space of this constraint is depicted in Figure A.2a. A conjunction of inequalities defines a polyhedral region, but a disjunction of inequalities defines a (possibly) disjoint union of polyhedra. There are few algorithms for solving a DP directly. Thus, our goal is to convert this constraint into a MIP form.



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

Figure A.2: Properties of disjunctive constraint (A.2).

An obvious but poor solution is to multiply the inequalities of each disjunct by a unique binary $\{0,1\}$ variable. For the i^{th} disjunct, introduce $\lambda_i \in \{0,1\}$ and allow exactly one of the λ_i's to equal 1. The idea is that if $\lambda_k = 1$, the k^{th} disjunct is enforced, and the i^{th}

disjunct is disregarded for all $i \neq k$. For the example, we have

$$\begin{bmatrix} x_1\lambda_1 \geq 1\lambda_1 \\ x_2\lambda_1 \geq 1\lambda_1 \\ (x_1 + x_2)\,\lambda_1 \leq 5\lambda_1 \end{bmatrix} \tag{A.3a}$$

$$\begin{bmatrix} 5\lambda_2 \leq x_1\lambda_2 \leq 8\lambda_2 \\ 4\lambda_2 \leq x_2\lambda_2 \leq 7\lambda_2 \end{bmatrix} \tag{A.3b}$$

$$\lambda_1 + \lambda_2 = 1 \tag{A.3c}$$

The disjunction has been eliminated, and we have purely a conjunction of inequalities, with some variables integer. One possible solution would set $\lambda_1 = 1$ and $\lambda_2 = 0$. In this case, the first set of inequalities reduces to the original disjunct R^1, and the second set degenerates into trivially satisfied constraints $0 \leq 0$, effectively disregarding disjunct R^2. Alternatively, the solution could be $\lambda_1 = 0$ and $\lambda_2 = 1$, in which case the second disjunct would get enforced. Although simple, this reformulation is nonlinear. We can do better—a linear reformulation can be obtained when the inequalities comprising the disjuncts are themselves linear.

The big-M method is one way to accomplish this. Again, binary variables λ_i are introduced, one for each disjunct. Constraint (A.1) is reformulated into

$$A^1x - b^1 \leq M^1\,(1 - \lambda_1) \tag{A.4a}$$

$$A^2x - b^2 \leq M^2\,(1 - \lambda_2) \tag{A.4b}$$

$$\lambda_1 + \lambda_2 = 1 \tag{A.4c}$$

where M^i are the so called big-M parameters. These are known upper bounds on $A^ix - b^i$. Consider $\lambda_1 = 1$ and $\lambda_2 = 0$. The second inequality reduces to $A^2x - b^2 \leq M^2$, which is trivially satisfied because, by definition, M^2 is the maximum value the left-hand-side could take. Effectively, disjunct R^2 is disregarded. The first inequality reduces to $A^1x - b^1 \leq 0$, which is the original disjunct R^1. So these MILP constraints are seen to be equivalent to constraint (A.1). The big-M reformulation of example (A.2) is

$$\begin{bmatrix} -x_1 + 1 \leq 0\,(1 - \lambda_1) \\ -x_2 + 1 \leq 0\,(1 - \lambda_1) \\ x_1 + x_2 - 5 \leq 10\,(1 - \lambda_1) \end{bmatrix} \tag{A.5a}$$

$$\begin{bmatrix} -x_1 + 5 \leq 4\,(1 - \lambda_2) \\ x_1 - 8 \leq 0\,(1 - \lambda_2) \\ -x_2 + 4 \leq 3\,(1 - \lambda_2) \\ x_2 - 7 \leq 0\,(1 - \lambda_2) \end{bmatrix} \tag{A.5b}$$

$$\lambda_1 + \lambda_2 = 1 \tag{A.5c}$$

Most algorithms for solving MILP algorithms first solve a relaxation of the problem.

Relaxation means the binary $\{0,1\}$ variables are instead allowed to be reals within the interval $[0,1]$. The size of the region defined by the relaxation can affect the performance of the algorithm significantly. In Figure A.2b, we show the feasible space of the above constraints, after relaxing the binaries, projected onto the (x_1, x_2) space.

Determining this projection is not so straightforward. Hooker (2000, p. 250) provides a general method, but an ad hoc procedure is sufficient for this small example. Consider constraints (A.5) as parametric on λ_1 and λ_2, which effectively means parametric on λ_1 because $\lambda_2 = 1 - \lambda_1$. Several of the inequalities are independent of λ_1 because it is being multiplied by 0. From these, we can infer that the region is at most the square defined by $1 \leq x_1 \leq 7$ and $1 \leq x_2 \leq 8$. The three inequalities that do depend on λ_1 define a triangle. It is not too difficult to determine that the points of this triangle are

$$A = (5 - 4\lambda_1, 4 - 3\lambda_1) \tag{A.6a}$$

$$B = (5 - 4\lambda_1, 10 - 6\lambda_1) \tag{A.6b}$$

$$C = (11 - 7\lambda_1, 4 - 3\lambda_1)\,. \tag{A.6c}$$

Each choice of $\lambda_1 \in [0,1]$ determines a specific triangle. The relaxed big-M region is the union of all these triangles intersected with the aforementioned square. The union of the triangles can be determined by considering the locus of points A for each choice of λ_1, and similarly for B and C. Each locus of points is a line. The vertices of the final polygon can be determined by calculating where these lines intersect the lines of the square. They are $(1.0, 1.0)$, $(4.0, 1.0)$, $(8.0, 2.7)$, $(8.0, 7.0)$, $(3.0, 7.0)$, and $(1.0, 4.0)$.

It is possible to provide MILP constraints whose relaxation provides a tighter convex region, indeed the tightest possible.

A.2 Convex Hull Reformulation

Balas (1974) defined a reformulation that provides the convex hull of the region defined by a disjunctive constraint. Theorem 2.1 of this work states that the convex hull of (A.1) is

$$A^1 \bar{x}^1 \leq b^1 \lambda_1 \tag{A.7a}$$

$$A^2 \bar{x}^2 \leq b^2 \lambda_2 \tag{A.7b}$$

$$\lambda_1 + \lambda_2 = 1 \tag{A.7c}$$

$$x = \bar{x}^1 + \bar{x}^2 \tag{A.7d}$$

where $\lambda_i \in [0,1]$. 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. As we have presented it, the assumption of Balas (1974, Corollary 2.1.1) must be satisfied, which essentially requires each disjunct to be bounded. This also allows including disjuncts that are infeasible, i.e.

define an empty region. We will shortly discuss how to handle disjunctive constraints not satisfying this condition.

Note that the λ_i's are not binary, and constraints (A.7) are pure LP constraints. They define the convex hull of (A.1), not the equivalent feasible space. Nonetheless, Balas (1974, Corollary 2.1.2) states that the LP (assuming a linear objective) is equivalent to the DP. Interestingly, it also states that this is so when the LP's solution gives $\lambda_k = 1$ and $\lambda_i = 0$ for all $i \neq k$. In other words, equivalence requires the λ_i's to be integral. Certain algorithms may satisfy this requirement.

However, we wish to provide an equivalent model without referring to any properties of algorithms that may are may not be used to optimize the model. Thus, we require $\lambda_i \in \{0, 1\}$, not $\lambda_i \in [0, 1]$. Then, constraints (A.7) are MILP constraints, and they define a region identical to the DP. Consider the solution with $\lambda_1 = 1$ and $\lambda_2 = 0$. The inequality corresponding to the second disjunct becomes $A^2\bar{x}^2 \leq 0$. Coupled with the precondition that the disjunct be bounded, this forces $\bar{x}^2$ to equal 0 (as will be demonstrated in the following example). This then means $x = \bar{x}^1$, and the first inequality becomes identical to the first disjunct, as desired.

The convex hull reformulation of example (A.2) is

$$\begin{bmatrix} \bar{x}_1^1 \geq 1\lambda_1 \\ \bar{x}_2^1 \geq 1\lambda_1 \\ \bar{x}_1^1 + \bar{x}_2^1 \leq 5\lambda_1 \end{bmatrix} \tag{A.8a}$$

$$\begin{bmatrix} 5\lambda_2 \leq \bar{x}_1^2 \leq 8\lambda_2 \\ 4\lambda_2 \leq \bar{x}_2^2 \leq 7\lambda_2 \end{bmatrix} \tag{A.8b}$$

$$\lambda_1 + \lambda_2 = 1 \tag{A.8c}$$

$$x = \bar{x}^1 + \bar{x}^2 \tag{A.8d}$$

where we require $\lambda_i \in \{0, 1\}$. Consider the solution with $\lambda_1 = 1$ and $\lambda_2 = 0$. The second set of inequalities degenerate into

$$\begin{bmatrix} 0 \leq \bar{x}_1^2 \leq 0 \\ 0 \leq \bar{x}_2^2 \leq 0 \end{bmatrix}$$

which forces $\bar{x}^2$ to equal 0. This makes $x = \bar{x}^1$, and the first set of inequalities become equivalent to disjunct R^1.

Relaxing to $\lambda_i \in [0, 1]$ gives the convex hull, depicted in Figure A.2c. The convex hull is easy to determine visually, so we have drawn it without actually calculating the projection of the constraints. Notice that this region is smaller than that of Figure A.2b, which was obtained with the big-M method.

Even though we require the λ_i's to be binary, relaxing them gives the convex hull, and we can expect this to be a good reformulation. This is often true but not definite. As will be discussed next, it is frequently necessary to add additional constraints into the disjuncts. Generally, the convex hull reformulation requires many more constraints than the big-M

reformulation, which can offset its benefits. Another motivation for choosing to implement the convex hull method is that the big-M method requires calculating the big-M parameters, making it somewhat more difficult to automate. Eventually, our software should allow both, and users should be able to select the method they prefer.

We have assumed thus far that the region defined by each disjunct is bounded, and the example we have been working with satisfied this requirement. The constraint

$$[x \leq 1] \vee [x \geq 4] \tag{A.9}$$

is reasonable, but it entails two unbounded regions. It would be useful to allow it. First, let us observe that the convex hull reformulation is not applicable. If we mistakenly apply it, we get

$$\left[\bar{x}^1 \leq 1\lambda_1\right] \tag{A.10a}$$

$$\left[\bar{x}^2 \geq 4\lambda_2\right] \tag{A.10b}$$

$$\lambda_1 + \lambda_2 = 1 \tag{A.10c}$$

$$x = \bar{x}^1 + \bar{x}^2. \tag{A.10d}$$

Now consider $\lambda_1 = 1$ and $\lambda_2 = 0$. The second inequality becomes $\bar{x}^2 \geq 0$, but this does not force $\bar{x}^2 = 0$ as needed. The system is satisfied with $\bar{x}^1 = 1$ and $\bar{x}^2 = 2$. This makes $x = 1 + 2 = 3$, which is infeasible for the disjunctive constraint.

Balas's requirement is that there is a system of inequalities $A'x \leq b'$, external to the disjunctive constraint, which is bounded. The overall DP is then

$$A'x \leq b' \tag{A.11a}$$

$$\left[A^1x \leq b^1\right] \vee \left[A^2x \leq b^2\right], \tag{A.11b}$$

where $A^ix \leq b^i$ need not be bounded. The reformulation is done by first inserting the bounded constraint into each disjunct, giving

$$\left[\begin{array}{c} A'x \leq b' \\ A^1x \leq b^1 \end{array}\right] \vee \left[\begin{array}{c} A'x \leq b' \\ A^2x \leq b^2 \end{array}\right]. \tag{A.12}$$

Each disjunct is now bounded.

In our automation of the convex hull method, we take a somewhat brute force approach to providing $A'x \leq b'$. We require that every variable x_i involved in a disjunction have known lower L_i and upper U_i bounds. The inequalities $L_i \leq x_i \leq U_i$ serve the role of $A'x \leq b'$. This is more restrictive than necessary, but the more flexible approach requires determining whether an arbitrary inequality $A'x \leq b'$ is bounded. This should not be too difficult, but we have not yet implemented this procedure.

We have also been assuming that there is only one disjunctive constraint. Of course, one

might want to write the model

$$R^1 \vee R^2 \tag{A.13a}$$

$$R^3 \vee R^4 \tag{A.13b}$$

where R^i stands for $A^i x \leq b^i$. There are two disjunctive constraints, or a conjunction of disjunctions. Simply applying the convex hull method to each disjunction separately does not give the convex hull of this problem. Rather, the constraints must be put into disjunctive normal form (DNF) to produce the single disjunction

$$\begin{bmatrix} R^1 \\ R^3 \end{bmatrix} \vee \begin{bmatrix} R^1 \\ R^4 \end{bmatrix} \vee \begin{bmatrix} R^2 \\ R^3 \end{bmatrix} \vee \begin{bmatrix} R^2 \\ R^4 \end{bmatrix}. \tag{A.14}$$

This is a much bigger system, and it is not clear that obtaining the convex hull this way is advantageous. Balas (1974, Section 5) provides another method known as sequential convexification to handle multiple disjunctions. Our software does not implement either of these methods yet. We simply apply the reformulation to each disjunction separately. Thus, we cannot call our method a convex hull reformulation overall, but the convex hull of each disjunction is obtained.

A.3 Adding Boolean Propositions

LP entails purely continuous systems; models are a conjunction of inequalities on reals. DP allows representing discrete phenomena by enriching the logical operators to conjunction and disjunction. MILP also allows representing discrete phenomena, but extends LP differently. Instead of enriching the operators, it enriches the data types. Variables can either be real or integer. Another useful extension is to include Boolean propositions because much discrete phenomena is most naturally expressed in this form. We first introduce Boolean propositions by themselves and discuss how they can be transformed into a system of linear inequalities on binary $\{0, 1\}$ variables. This will be followed by a discussion of their incorporation into disjunctive programs.

Let Y be a set of Boolean variables. A Boolean proposition allows connecting these variables with the operators negation $\neg$, conjunction $\wedge$, and disjunction $\vee$. Other operators such as implication $\Rightarrow$ can be defined in terms of these. The conversion of Boolean propositions into integer constraints has been discussed by several authors (e.g. Mitra et al., 1994; Bemporad and Morari, 1999; Hooker, 2000), and Sasao (1999) provides a general review of Boolean algebra.

A simple procedure is to replace each Boolean variable Y with a binary $\{0, 1\}$ variable y. A negated variable $\neg Y$ is replaced with the expression $1 - y$, disjunction $\vee$ is replaced with addition $+$, and conjunction $\wedge$ is replaced with multiplication $*$. Finally, the resulting expression is required to be ≥ 1, corresponding to requiring it to be true. For example,

$$Y_1 \vee (Y_2 \wedge \neg Y_3) \tag{A.15}$$

becomes

$$y_1 + (y_2 * (1 - y_3)) \geq 1. \tag{A.16}$$

The resulting constraint is nonlinear. It is possible to do better—a linear system can always be obtained. This requires first putting the Boolean proposition into conjunctive normal form (CNF), a procedure discussed in many places; see for example Visser (2001) or Chang and Lee (1987, p. 12–15). Then, instead of replacing conjunction with multiplication, each conjunct is separately transformed into an inequality. The CNF of (A.15) is

$$(Y_1 \vee Y_2) \wedge (Y_1 \vee \neg Y_3)\,. \tag{A.17}$$

Each conjunct must contain only disjunctions and variables, possibly negated. This is converted into the two inequalities

$$y_1 + y_2 \geq 1 \tag{A.18a}$$

$$y_1 + (1 - y_3) \geq 1 \tag{A.18b}$$

which are linear.

Interestingly, Raman and Grossmann (1994) introduce Booleans into DP in such a way that the convex hull method can still be employed. Their extended framework is called generalized disjunctive programming (GDP). Their definition of a disjunctive constraint, simplified to the two disjunct case with linear inequalities, is

$$\begin{bmatrix} Y_1 \\ A^1 x \leq b^1 \end{bmatrix} \vee \begin{bmatrix} Y_2 \\ A^2 x \leq b^2 \end{bmatrix} \tag{A.19}$$

where $Y_i \in \{\texttt{true}, \texttt{false}\}$. The interpretation is that if $Y_k = \texttt{true}$, then $A^k x \leq b^k$ is enforced and $A^i x \leq b^i$ is disregarded for all $i \neq k$ (we will see shortly that an implicit requirement of their reformulation is that exactly one Y_i be true). Separately, they allow a Boolean proposition Ω, which can involve the Boolean variables Y. This has the benefit of allowing additional logical conditions to govern which disjunct is active.

Their insight was that the λ variables introduced as part of the convex hull reformulation could be useful in the original model. In Balas's definition of a DP (A.1), the model cannot make any use of λ. These variables arise only during the transformation. In (A.19), each disjunct is labeled with a Boolean Y_i, allowing this variable to be used elsewhere in the model, and Y_i is associated with λ_i. Their reformulation of (A.19) is exactly (A.7), but now it is understood that λ_i is the binary variable replacing Y_i. The proposition $\Omega(Y)$ is replaced with inequalities involving λ_i's.

In addition to the constraints on λ resulting from $\Omega(Y)$, the constraint $\lambda_1 + \lambda_2 = 1$ is also present because it is required as part of the convex hull reformulation. Thus, a mutual exclusivity constraint over the Y_i's is implicit in Raman and Grossmann's method, i.e. exactly one Y_i must be true.

The disjunctive constraints we support our work is more flexible. Amongst other extensions, we allow any Boolean proposition within disjunctions, rather than a single Boolean

variable. For example, one could write

$$\begin{bmatrix} \Omega_1(Y) \\ A^{12}x \leq b^{12} \end{bmatrix} \vee \begin{bmatrix} \Omega_2(Y) \\ A^{22}x \leq b^{22} \end{bmatrix}. \tag{A.20}$$

This is reformulated by first converting the Boolean propositions into integer inequalities, giving

$$\begin{bmatrix} A^{11}y \leq b^{11} \\ A^{12}x \leq b^{12} \end{bmatrix} \vee \begin{bmatrix} A^{21}y \leq b^{21} \\ A^{22}x \leq b^{22} \end{bmatrix}. \tag{A.21}$$

We can rewrite this as

$$\left[A^1 z \leq b^1\right] \vee \left[A^2 z \leq b^2\right] \tag{A.22}$$

where A^i combines A^{i1} and A^{i2}, b^i combines b^{i1} and b^{i2}, and z is the vector of integer variables y and real variables x. This is in the DP form (A.1), albeit with some variables integer. Balas's standard convex hull reformulation can be applied. This will provide the convex hull of the disjunctive constraint with all z relaxed to reals. With some variables integer, ideally we would like the integer hull, but that is not obtained. Balas (1979, Section 7) discusses possible improvements in this setting.

In this method, there is no correspondence between the λ's generated as part of the convex hull method and the Booleans Y. In each disjunct, one could always replace $\Omega_i(Y)$ with a new Boolean variable Y_i' and separately require $Y_i' \Leftrightarrow \Omega_i(Y)$. Thus, the additional flexibility in this method over Raman and Grossmann's is not that arbitrary Boolean propositions are allowed in the disjuncts. It is that these need not be mutually exclusive.

Turning the discussion around, Raman and Grossmann's method can be viewed as a specialization of this general method. If each Ω_i is in fact an individual Boolean variable Y_i and exactly one can be true, then the binary y replacing Y can play the role of the λ needed for the convex hull reformulation. If these conditions are not met, both sets of variables y and λ must be introduced, making the program larger.

A.4 Conclusions

We have discussed the basic methods for converting disjunctive constraints and Boolean propositions into pure MILP constraints. These are considered established techniques. But notice that our discussion here has always presented the models in a canonical form and has not employed index sets. Even the simplest systems can rarely be modeled this way. Furthermore, the transformations have been discussed in English. Consider the statement "disjunctive constraint (A.1) can be reformulated into constraints (A.7)". One might consider this a rather precise statement since (A.1) and (A.7) are. It is not however a mathematical definition of a transformation. A mathematical definition would provide a function $f : \mathbb{D} \to \mathbb{M}$, where $\mathbb{D}$ is the set of all DP constraints and $\mathbb{M}$ is the set of all MILP constraints. These sets would include all constraints as one would want to write in practice, not just

canonical forms. We would then be able to apply this function as

$$f\left(\begin{bmatrix} x_1 \geq 1 \\ x_2 \geq 1 \\ x_1 + x_2 \leq 5 \end{bmatrix} \vee \begin{bmatrix} 5 \leq x_1 \leq 8 \\ 4 \leq x_2 \leq 7 \end{bmatrix}\right)$$

where the argument is literally constraint (A.2), and the result would be literally constraint (A.8). Providing definitions at this level of rigor is the subject of Chapters 4–8 of this thesis and is required for computers to understand our models and execute the transformations automatically.

Appendix B

Variable Binding Meta-Logic

For the theories of un-indexed programs and index sets, Chapters 4 and 6, we presented procedures for determining the set of free variables and substituting into various constructs. Also, we assumed there was a method for α-converting bound variable names. In Section 7.1.2, we discussed that the number of such operations needed for indexed programs (augmented with the let constructs of the concrete syntax) is combinatorial in the number of syntactic categories. We now present a meta-logic within which the various logics discussed in this dissertation can be defined. Operations regarding variables are defined once in this meta-logic and become available automatically for any logic implemented within it. This saves us the trouble of writing hundreds of functions.

We first discuss some relevant terminology. A *binder* is a syntactic item that introduces a variable. For example, the λ in $\lambda i : \sigma \centerdot e$ and the $\sum$ in $\sum_{i:\sigma} e$ are binders. The full terms $\lambda i : \sigma \centerdot e$ and $\sum_{i:\sigma} e$ are called *binding terms* because they introduce variables that are bound within them, i.e. the variables do not have any significance outside the terms. Sometimes there is no symbol associated with a binder. For example, $i : \sigma \to \tau$ is a binding term that introduces the variable i, but there is no symbol prefixing the i. The *scope* of a variable is its range of significance. In $\lambda i : \sigma \centerdot e$, the scope of i is e. Any i that might happen to occur in σ is unrelated to the i being introduced in this term. Binding terms can be *α-converted*. Since the variable introduced in a binding term has no significance outside the term, its name can be changed without altering the meaning of the term. For example, $\lambda i : [1, i) \centerdot i + 1$ can be α-converted to $\lambda j : [1, i) \centerdot j + 1$. The introduced i is renamed to j, and all uses of it are correspondingly changed. The i in $[1, i)$ remains unchanged.

The meta-logic internalizes these concepts. The abstract syntax of programs captured the structure of arithmetic expressions. For example, the rules for constructing expressions tell us whether an expression is an addition, one of whose sub-expressions is a multiplication, or vice-versa (Harper, 2005, Chp. 5). The notion of a bound variable and scoping rules were however not captured. For example, there is nothing intrinsic in the term $\lambda i : \sigma \centerdot e$ that requires the scope of i to be e. We could have also declared σ to be in its scope. These decisions are external to the abstract syntax. Internalizing these concepts in the meta-logic allows us to define, in an abstract way, the operations that depend on them.

B.1 Syntax

The syntax of the meta-logic is parameterized on syntactic categories c and term-producing operators o, which are dependent on the logic of interest. Say our intention is to implement the indexed programming language within the meta-logic. Then, we would define

$$c ::= \text{expri} \mid \text{typei} \mid \text{kindi} \mid \text{expr} \mid \text{type} \mid \text{prop} \mid \text{prop\_type} \mid \text{prog} \tag{B.1}$$

Each c refers to one of the syntactic categories of the language. For example, expri refers to the set of all index expressions ε, and expr refers to all program expressions e. In the chapters, we did not give names to such sets. The notation ε referred to a single index expression, and it referred generically to any element adhering to the syntax of index expressions. Now we give the name expri to the set of all index expressions.

Every specific syntactic form can be thought of as being generated by a term-producing operator. For example, $\varepsilon_1 + \varepsilon_2$ is generated by the operator plus_ε. This operator takes two terms ε_1 and ε_2 and returns $\varepsilon_1 + \varepsilon_2$. The operator applyi_e takes two arguments, e_1 and e_2, and returns the application $e_1\, e_2$. As one more example, "$\text{int}_\varepsilon\ k$" is a family of operators, one for each integer k. So "$\text{int}_\varepsilon\ 3$" is an operator that takes zero arguments and returns a term representing the index expression 3. We will also define the family of operators "$\text{int}_e\ r$", which would return a program expression representing the number r, for any r.

Operators generating binding terms require arguments of two forms: terms, as in the above examples, and bound variables. For example, the operator lambda_e takes in the arguments x, τ, and e and produces $\lambda x : \tau \,.\, e$. The x is not a term; it is the name of a bound variable.

The full set of operators for the indexed programming language is defined at the end of this appendix. One operator o must be defined for every form, except variables, of every syntactic category. Variables are a primitive notion of the binding logic and a special construct is made available for them. Assuming that c and o have been provided, we now define the syntax of the meta-logic.

A variable

$$v := (x\ \texttt{as}\ c) \tag{B.2}$$

in the meta-logic is an identifier as usual but also states the category to which it belongs. With this design, there can be no confusion between variables of different categories. For example (x as expri) and (x as expr) represent an index and program expression variable, respectively. These are distinct variables even though they have the same name.

A term in the meta-logic is an abstraction of all constructs belonging to any syntactic category. Its syntax is

$$t ::= \texttt{VAR}\, v \mid o\, p \tag{B.3}$$

The first form $\texttt{VAR}\, v$ turns a variable into a term. The second form $o\, p$ represents an operator o applied to its arguments p. The argument to an operator is a list of terms and bound variables. Precisely, its syntax is

$$p ::= \emptyset \mid p, v. \mid p, t \tag{B.4}$$

The $v_\bullet$ denotes the introduction of a variable v. All terms t occurring after some $v_\bullet$ are in the scope of v. A variable v is shadowed by a variable v' if $v_\bullet'$ occurs to the right of $v_\bullet$ and $v = v'$. For clarity, angle brackets $\langle\rangle$ will be used to enclose a list that denotes a specific p.

The lone variable x used as an expression would be represented by the meta-logical term $\texttt{VAR}\ (x\ \texttt{as}\ \text{expr})$. The same variable used as an index expression would be represented by the term $\texttt{VAR}\ (x\ \texttt{as}\ \text{expri})$. All constructs other than variables are represented in the form $o\,p$. For example, $\lambda x : \tau \bullet e$ is represented by the term $\text{lambda}_e\,\langle \ulcorner\tau\urcorner, (x\ \texttt{as}\ \text{expr})_\bullet, \ulcorner e\urcorner\rangle$, where lambda_e is a term-producing operator, $\ulcorner\tau\urcorner$ is the term representing τ, and $\ulcorner e\urcorner$ is the term representing e.

$\texttt{VAR}\,v$ is a family of operators. For example, $\texttt{VAR}\ (x\ \texttt{as}\ \text{expr})$ takes zero arguments and returns a term representing the variable $(x\ \texttt{as}\ \text{expr})$. In that sense, the forms $\texttt{VAR}\,v$ and $o\,p$ are similar. They differ only because the operators o have been abstracted out while $\texttt{VAR}\,v$ has not.

Operator argument lists p are categorized into the argument type

$$\theta ::= \emptyset \mid \theta, c_\bullet \mid \theta, c \tag{B.5}$$

where $c_\bullet$ denotes the set of bound variables of category c, and c (without the "$\bullet$") represents the set of terms of category c. Square brackets [] will be used around a list denoting an argument type θ.

Continuing the above example, the operator lambda_e expects an argument of type $[\text{type}, \text{expr}_\bullet, \text{expr}]$. This indicates that lambda_e takes three arguments: a type, an introduced expression variable, and an expression.

B.2 Judgements

Just as a type checker for the mathematical programming language detects ill-formed programs, we can define a type checker for the meta-logic to detect ill-formed logics. For example, the operator lambda_e is supposed to be applied to arguments of type $[\text{type}, \text{expr}_\bullet, \text{expr}]$. However, we might make a mistake when writing the mathematical programming logic in the meta-logic. We might accidentally form the meta-logical term $\text{lambda}_e\,\langle \ulcorner\tau\urcorner, \ulcorner e\urcorner\rangle$ for the expression $\lambda x : \tau \bullet e$, forgetting the introduced variable. The type checker we define here actually did catch a few such errors in our software. This pinpointed the bug precisely, and it could be fixed easily. Without it, the error would have shown up in other ways that might not have been tracked down easily.

Let $o : \theta$ mean operator o is supposed to be applied to an argument of type θ. Also, an operator produces terms of a specific category, given by $o \ltimes c$. Finally, let $c_1 \equiv c_2$ be a category equivalence relation. These judgements must be provided for the logic that will be implemented within the meta-logic. Given these, we provide the following judgements.

Variable Equality

Let $v = v'$ mean variable v is identical to variable v'. Its definition is

$$\frac{c_1 \equiv c_2}{(x_1 \,\mathtt{as}\, c_1) = (x_2 \,\mathtt{as}\, c_2)} \text{ where } x_1 = x_2 \tag{B.6}$$

It is assumed that there an equality test on alphanumeric strings x is available.

Category of Variable

For convenience, we let $v \ltimes c$ mean v belongs to category c. Its definition is simply

$$\frac{}{(x \,\mathtt{as}\, c) \ltimes c} \tag{B.7}$$

Operator Type Equality

Let $\theta_1 \equiv \theta_2$ mean the two types are equivalent. The definition is inductive on the construction of θ,

$$\frac{}{\emptyset \equiv \emptyset} \tag{B.8a}$$

$$\frac{\theta_1 \equiv \theta_2 \quad c_1 \equiv c_2}{[\theta_1, c_1\centerdot] \equiv [\theta_2, c_2\centerdot]} \tag{B.8b}$$

$$\frac{\theta_1 \equiv \theta_2 \quad c_1 \equiv c_2}{[\theta_1, c_1] \equiv [\theta_2, c_2]} \tag{B.8c}$$

Type of Operator Argument List

Let $p : \theta$ mean p is of type θ. Its definition is inductive on the form of p,

$$\frac{}{\langle\emptyset\rangle : [\emptyset]} \tag{B.9a}$$

$$\frac{p : \theta}{\langle p, (x \,\mathtt{as}\, c)\centerdot\rangle : [\theta, c\centerdot]} \tag{B.9b}$$

$$\frac{p : \theta \quad t \ltimes c}{\langle p, t\rangle : [\theta, c]} \tag{B.9c}$$

An empty list of empty type. An introduced variable $(x \,\mathtt{as}\, c)\centerdot$ belongs in the category $c\centerdot$. A term t belongs in the category c if $t \ltimes c$, defined next.

Category of Term

Let $t \ltimes c$ mean t is a well-formed c term. Its definition is inductive on the form of t,

$$\frac{}{\mathtt{VAR}\ (x \,\mathtt{as}\, c) \ltimes c} \tag{B.10a}$$

$$\frac{o : \theta \quad p : \theta' \quad \theta \equiv \theta' \quad o \ltimes c}{o\,p \ltimes c} \tag{B.10b}$$

A variable term $\texttt{VAR}\ (x\ \texttt{as}\ c)$ is always well-formed and is of category c. A term $o\,p$ is well-formed if the type of the applied operator is θ' is equivalent to the type θ expected by the operator o. The term belongs to category c if $o \ltimes c$.

B.3 Meta-Operations

We now define some operations on terms t of the meta-logic. The availability of these operations is what frees us from defining numerous corresponding operations in the indexing and mathematical programming languages.

B.3.1 Free Variables

Let $FV\,(c, t)$ denote the free c variables of term t. For example, $FV\,(\text{expr}, t)$ gives the free expression variables in t, while $FV\,(\text{expri}, t)$ gives the free index expression variables in t. Its definition is mutually recursive with $FV_{\#}\,(c, p)$, the free c variables in p. The definition of $FV\,(c, t)$ depends on the form of t,

1. $FV\,(c, \texttt{VAR}\,v) = \begin{cases} \{v\} \text{ if } v \ltimes c \\ \emptyset \text{ otherwise} \end{cases}$
2. $FV\,(c, o\,p) = FV_{\#}\,(c, p)$

The definition of $FV_{\#}\,(c, p)$ is inductive on the construction of p in reverse,

1. $FV_{\#}\,(c, \langle\emptyset\rangle) = \emptyset$
2. $FV_{\#}\,(c, \langle v_{\bullet}, p\rangle) = \begin{cases} FV_{\#}\,(p) \setminus \{v\} \text{ if } v \ltimes c \\ FV_{\#}\,(p) \text{ otherwise} \end{cases}$
3. $FV_{\#}\,(c, \langle t, p\rangle) = FV\,(c, t) \cup FV_{\#}\,(c, p)$

B.3.2 Substitution

Let $\{t/v\}\,t'$ denote the substitution of term t for v in t'. Similarly, $\{t/v\}_{\#}\,p$ is the substitution of t for v in p. Substitution is only defined if the category of t is the same as that of v. Substitution into terms is defined by the rules

1. $\{t/v\}\,(\texttt{VAR}\,v') = \begin{cases} \text{if } v = v' \text{ then } t \\ \text{else } \texttt{VAR}\,v' \end{cases}$
2. $\{t/v\}\,(o\,p) = o\ \{t/v\}_{\#}\,p$

Substitution into operator argument lists is defined by the rules

1. $\{t/v\}_{\#}\,\langle\emptyset\rangle = \emptyset$

2. $\{t/v\}_{\#}\langle v'\centerdot, p'\rangle = \begin{cases} \text{if } v \notin FV_{\#}(c, \langle v'\centerdot, p'\rangle)\text{, where } v \ltimes c\text{, then} \\ \quad \langle v'\centerdot, p'\rangle \\ \text{else if } v' \notin FV(c', t)\text{, where } v' \ltimes c'\text{, then} \\ \quad \left\langle v'\centerdot, \{t/v\}_{\#} p' \right\rangle \\ \text{else} \\ \quad \{t/v\}_{\#} \left\langle v''\centerdot, \{\mathtt{VAR}\, v''/v'\}_{\#} p' \right\rangle \end{cases}$

   where in the final else branch, v'', with $v'' \ltimes c'$, chosen such that

$$v'' \notin (FV(c', t) \cup FV_{\#}(c', p'))\,.$$

3. $\{t/v\}_{\#}\langle t', p\rangle = \left\langle \{t/v\}\, t', \{t/v\}_{\#} p \right\rangle$

The second rule is similar to substitution into the existential proposition; see rule 6 on page 52.

B.3.3 Alpha Conversion

Given two terms, we would like to rename the bound variables in them so that they match. Let $(t_1, t_2) \rightarrowtail^c (t_1', t_2')$ be the judgement relating two terms t_1 and t_2 to their α-converted terms t_1' and t_2', such that any introduced variables, of category c, in t_1' and t_2' match. Most of the work is done by the corresponding judgement on operator argument lists $(p_1, p_2) \rightarrowtail^c_{\#} (p_1', p_2')$. The definition of $\rightarrowtail^c$ is

$$\frac{}{(\mathtt{VAR}\, v_1, o\, p_2) \rightarrowtail^c (\mathtt{VAR}\, v_1, o\, p_2)} \tag{B.11a}$$

$$\frac{}{(o\, p_1, \mathtt{VAR}\, v_2) \rightarrowtail^c (o\, p_1, \mathtt{VAR}\, v_2)} \tag{B.11b}$$

$$\frac{}{(\mathtt{VAR}\, v_1, \mathtt{VAR}\, v_2) \rightarrowtail^c (\mathtt{VAR}\, v_1, \mathtt{VAR}\, v_2)} \tag{B.11c}$$

$$\frac{(p_1, p_2) \rightarrowtail^c_{\#} (p_1', p_2')}{(o\, p_1, o\, p_2) \rightarrowtail^c (o\, p_1', o\, p_2')} \tag{B.11d}$$

There are no introduced variables in the term $\mathtt{VAR}\, v$. Thus, if either t_1 or t_2 are of this form, no matching needs to be done. If both are of the form $o\, p$, then $\rightarrowtail^c_{\#}$ is employed.

The definition of $(p_1, p_2) \rightarrowtail^c_\# (p'_1, p'_2)$ is inductive on the construction of p_1 and p_2 in reverse,

$$\frac{(p_2, p_1) \rightarrowtail^c_\# (p'_2, p'_1)}{(p_1, p_2) \rightarrowtail^c_\# (p'_1, p'_2)}\text{symmetry} \tag{B.12a}$$

$$\frac{}{(\emptyset, p_2) \rightarrowtail^c_\# (\emptyset, p_2)} \tag{B.12b}$$

$$\frac{v \ltimes c \quad (\langle v_\blacksquare, p_1\rangle, p_2) \rightarrowtail^c_\# (p'_1, p'_2)}{(\langle v_\blacksquare, p_1\rangle, \langle t, p_2\rangle) \rightarrowtail^c_\# (p'_1, \langle t, p'_2\rangle)} \tag{B.12c}$$

$$\frac{v \not\ltimes c \quad (p_1, p_2) \rightarrowtail^c_\# (p'_1, p'_2)}{(\langle v_\blacksquare, p_1\rangle, \langle t, p_2\rangle) \rightarrowtail^c_\# (\langle v_\blacksquare, p'_1\rangle, \langle t, p'_2\rangle)} \tag{B.12d}$$

$$\frac{(p_1, p_2) \rightarrowtail^c_\# (p'_1, p'_2)}{(\langle t_1, p_1\rangle, \langle t_2, p_2\rangle) \rightarrowtail^c_\# (\langle t_1, p'_1\rangle, \langle t_2, p'_2\rangle)} \tag{B.12e}$$

$$\frac{v_1 \not\ltimes c \quad v_2 \not\ltimes c \quad (p_1, p_2) \rightarrowtail^c_\# (p'_1, p'_2)}{(\langle v_{1\blacksquare}, p_1\rangle, \langle v_{2\blacksquare}, p_2\rangle) \rightarrowtail^c_\# (\langle v_{1\blacksquare}, p'_1\rangle, \langle v_{2\blacksquare}, p'_2\rangle)} \tag{B.12f}$$

$$\frac{v_1 \ltimes c \quad v_2 \not\ltimes c \quad (\langle v_{1\blacksquare}, p_1\rangle, p_2) \rightarrowtail^c_\# (p'_1, p'_2)}{(\langle v_{1\blacksquare}, p_1\rangle, \langle v_{2\blacksquare}, p_2\rangle) \rightarrowtail^c_\# (p'_1, \langle v_{2\blacksquare}, p'_2\rangle)} \tag{B.12g}$$

$$\frac{v_1 \ltimes c \quad v_2 \ltimes c \quad (\{\mathtt{VAR}\, v/v_1\}\, p_1, \{\mathtt{VAR}\, v/v_2\}\, p_2) \rightarrowtail^c_\# (p'_1, p'_2)}{(\langle v_{1\blacksquare}, p_1\rangle, \langle v_{2\blacksquare}, p_2\rangle) \rightarrowtail^c_\# (\langle v_\blacksquare, p'_1\rangle, \langle v_\blacksquare, p'_2\rangle)} \tag{B.12h}$$

where in the last rule v, with $v \ltimes c$, is chosen fresh such that $v \notin (FV_\# (c, p_1) \cup FV_\# (c, p_2))$. Actually, to maintain the same names as much as possible, v is chosen to be v_1 if $v_1 \notin FV_\# (c, p_2)$, or v_2 if $v_2 \notin FV_\# (c, p_1)$. The matching is done in the last rule. If both lists begin with an introduced variable v_1 and v_2, respectively, the names of these two are changed to a common v. The lists p_1 and p_2 are appropriately modified to reflect the name change. All other rules simply recurse into their sub-lists.

A ternary matcher $(t_1, t_2, t_3) \rightarrowtail (t'_1, t'_2, t'_3)$ could be provided in analogy to the above binary matcher. Our software implementation provides a general n-ary matcher.

B.4 Example: Indexed Program Logic

The meta-logic serves as a primitive logical framework, a framework within which languages could be developed. Had we used it from the onset, the variables operations such as free variable calculation, substitution, and α-conversion, would simply be available for use. However, we formulated our languages independent of the meta-logic. Thus, we must map our original formulation, those in the chapters, to the syntax of the meta-logic.

The syntax of the meta-logic is parametric on o and c, and the judgements on $o : \theta$, $o \ltimes c$, and $c_1 \equiv c_2$. The syntactic categories c were previously defined by rules (B.1). Category equivalence is a reflexivity relation,

$$\frac{}{c \equiv c} \tag{B.13}$$

An operator o is required for every syntactic construct of the indexed mathematical

programming language, except for variables. The operators with their expected argument types, i.e. the judgement $o : \theta$, are stated in the table below. Subscripts have been used in the naming scheme to emphasize which category the term operator generates a term for. So these provide the judgement $o \ltimes c$.

Nothing really has been done in defining o. It is merely a change of notation. Instead of denoting a plus expression as $e_1 + e_2$, we now write it as $\text{plus}_e\,(e_1, e_2)$. The latter uses a name for the operator rather than a symbol.

Table B.1: Formulating indexed programs within meta-logic.

Operator o	Argument Type θ
$\text{label}_\varepsilon\ l$	$\emptyset$
$\text{int}_\varepsilon\ k$	$\emptyset$
$\text{tuple}_\varepsilon\ k$	$[\text{expri}_1, \ldots, \text{expri}_k]$
$\text{proj}_\varepsilon\ k$	$[\text{expri}]$
plus_ε	$[\text{expri}, \text{expri}]$
minus_ε	$[\text{expri}, \text{expri}]$
mult_ε	$[\text{expri}, \text{expri}]$
$\text{case}_\varepsilon\ \{l_1, \ldots, l_m\}$	$[\text{expri}, \text{expri}_1, \ldots, \text{expri}_m]$
$\text{case}_\varepsilon\ [k_L, k_U]$	$[\text{expri}, \text{expri}_1, \ldots, \text{expri}_{k_U - k_L + 1}]$
neg_ε	$[\text{expri}]$
$\text{labels}_\sigma\ \{l_1, \ldots, l_m\}$	$\emptyset$
interval_σ	$[\text{expri}, \text{expri}]$
$\text{product}_\sigma\ k$	$[\text{typei}_1, \text{expri}_{\blacksquare 1}, \ldots, \text{typei}_k, \text{expri}_{\blacksquare k}]$
$\text{case}_\sigma\ \{l_1, \ldots, l_m\}$	$[\text{expri}, \text{typei}_1, \ldots, \text{typei}_m]$
$\text{case}_\sigma\ [k_L, k_U]$	$[\text{expri}, \text{typei}_1, \ldots, \text{typei}_{k_U - k_L + 1}]$
ascription_σ	$[\text{typei}, \text{kindi}]$
lambda_σ	$[\text{expri}_\blacksquare, \text{typei}]$
apply_σ	$[\text{typei}, \text{expri}]$
IndexSet_κ	$\emptyset$
arrow_κ	$[\text{typei}, \text{expri}_\blacksquare, \text{kindi}]$
real_τ	$\emptyset$
bool_τ	$\emptyset$
product_τ	$[\text{type}, \ldots, \text{type}]$
arrow_τ	$[\text{type}, \text{type}]$
arrowi_τ	$[\text{typei}, \text{expri}_\blacksquare, \text{type}]$
$\text{real}_e\ r$	$\emptyset$
true_e	$\emptyset$
false_e	$\emptyset$
neg_e	$[\text{expr}]$
plus_e	$[\text{expr}, \text{expr}]$
minus_e	$[\text{expr}, \text{expr}]$
mult_e	$[\text{expr}, \text{expr}]$
not_e	$[\text{expr}]$

Table B.1: (cont.)

Operator o	Argument Type θ
or_e	$[\text{expr}, \text{expr}]$
and_e	$[\text{expr}, \text{expr}]$
$\text{tuple}_e\ k$	$[\text{expr}_1, \ldots, \text{expr}_k]$
$\text{proj}_e\ k$	$[\text{expr}]$
lambda_e	$[\text{type}, \text{expr}_\bullet, \text{expr}]$
apply_e	$[\text{expr}, \text{expr}]$
SUM_e	$[\text{typei}, \text{expri}_\bullet, \text{expr}]$
$\text{case}_e\ \{l_1, \ldots, l_m\}$	$[\text{expri}, \text{expr}_1, \ldots, \text{expr}_m, \text{expri}_\bullet, \text{type}]$
$\text{case}_e\ [k_L, k_U]$	$\left[\text{expri}, \text{expr}_1, \ldots, \text{expr}_{k_U - k_L + 1}, \text{expri}_\bullet, \text{type}\right]$
lambdai_e	$[\text{typei}, \text{expri}_\bullet, \text{expr}]$
applyi_e	$[\text{expr}, \text{expri}]$
ascription_e	$[\text{expr}, \text{type}]$
isTrue_c	$[\text{expr}]$
equal_c	$[\text{expr}, \text{expr}]$
lte_c	$[\text{expr}, \text{expr}]$
exists_c	$[\text{type}, \text{expr}_\bullet, \text{prop}]$
disj_c	$[\text{prop}, \text{prop}]$
conj_c	$[\text{prop}, \text{prop}]$
exists_c	$[\text{type}, \text{expr}_\bullet, \text{prop}]$
DISJ_c	$[\text{typei}, \text{expri}_\bullet, \text{prop}]$
CONJ_c	$[\text{typei}, \text{expri}_\bullet, \text{prop}]$
$\text{case}_c\ \{l_1, \ldots, l_m\}$	$[\text{expri}, \text{prop}_1, \ldots, \text{prop}_m, \text{expri}_\bullet, \text{prop\_type}]$
$\text{case}_c\ [k_L, k_U]$	$\left[\text{expri}, \text{prop}_1, \ldots, \text{prop}_{k_U - k_L + 1}, \text{expri}_\bullet, \text{prop\_type}\right]$
T_c	$\emptyset$
F_c	$\emptyset$
$\text{prog}_p\ (\delta, k)$	$[\text{type}_1, \text{expr}_{1\bullet}, \ldots, \text{type}_k, \text{expr}_{k\bullet}, \text{expr}, \text{prop}]$
lambdai_c	$[\text{typei}, \text{expri}_\bullet, \text{prop}]$
lambda_c	$[\text{type}, \text{expr}_\bullet, \text{prop}]$
applyi_c	$[\text{prop}, \text{expri}]$
apply_c	$[\text{prop}, \text{expr}]$
ascription_c	$[\text{prop}, \text{prop\_type}]$
Prop_ζ	$\emptyset$
arrowi_ζ	$[\text{typei}, \text{expri}_\bullet, \text{prop\_type}]$
arrow_ζ	$[\text{type}, \text{prop\_type}]$

Appendix C

Concrete Syntax

Definitions of logics use what is called abstract syntax. For example, we state that $e_1 + e_2$ is an expression. It is immaterial to the theory whether the + actually occurs in between the two expressions. What is important is that there is such a thing as addition in the logic, and that addition involves two sub-expressions. Concrete syntax, on the other hand, is concerned with such details. This is the syntax that is actually typed into an input file accepted by the software implementation. Concrete syntax varies from the abstract syntax for a few reasons.

Firstly, the parser is generated with the popular tools Lex and Yacc (Lesk and Schmidt, 1978; Johnson, 1979; Levine et al., 1995), and the syntax must be compatible with the requirements of these tools. (Our code is written in ML. So we use the ML variants of these, called ML-Lex and ML-Yacc.) These tools parse only ASCII text symbols. The LaTeX symbol $\sum$, for example, must be written `SUM`. A summary of the relationship between the ASCII and LaTeX notation is provided at the end of this appendix.

An effort is made to provide the simplest syntax possible. For example, the dot used in lambda abstractions is not required. It is sufficient to write $\lambda x\ e$ instead of $\lambda x \,.\, e$.

The dot "." is known as Peano's dot, and can be used for clarity when desired. It is actually an alternative to parentheses (Andrews, 2002, p. 15). It is customary to allow parentheses to delimit expressions. One can write $3 * (2 + 4)$ to indicate that the addition should take precedence over the multiplication. We provide Peano's dot in the concrete syntax. It often leads to cleaner programs. The previous expression can instead be written $3 * .2 + 4$. Peano's dot effectively stands for a left parenthesis. The right parenthesis is inferred by placing it as far to the right as possible. As a result, the notation $\lambda x \,.\, e$ is still valid, but the dot is not part of the function notation. Rather it is part of the expression. And of course, plain ASCII text must be used. So "." is written as a plain period ".".

Sometimes, the concrete syntax is more cumbersome than the abstract syntax. Projections are denoted $e.k$, which could have been supported, but we instead require `#k e` because this coincides with the convention of ML. Similarly, $\varepsilon.k$ must be written `#k epsilon`. We also have several application forms, $\sigma\,\varepsilon$, $e\,\varepsilon$, and $c\,\varepsilon$, where the argument is an index expression. In these cases, the argument must be surrounded with square brackets.

The main benefit of functions is that they can be defined once and reused, but our core

theory actually does not provide a mechanism for reusing functions. The concrete syntax supports a "let" construct. This allows naming a syntactic entity and subsequently referring to it by its name. One could write

```
let
  expr f = fni i . x[i] + 1
in
  y + f[5] = 3
end
```

which is a proposition employing a locally defined abbreviation. The line

```
expr f = fni i . x[i] + 1
```

declares that the name `f` refers to the expression `fni i . x[i] + 1`. Names for any syntactic construct can be provided, but the parser is not able to look at the item to determine what syntactic category it belongs to. The keyword `expr` is required to let the parser know that the name `f` being introduced is for an expression, as opposed to say an index type. At parse time, we eliminate all let constructs by substituting in the constructs that the names stand for. The above becomes

```
y + (fni i . x[i] + 1)[5] = 3
```

In general, any number of abbreviations can be introduced for any syntactic construct. Another example is

```
let
  setf S = {'A', 'B', 'C'}
  propf p_F = fni j . x[j] + 1 <= y[j]
in
  CONJ i:S . p_F[i]
end
```

where a few more keywords have been introduced. `setf` refers to an index type σ, `propf` refers to a proposition c, and `CONJ` is the ASCII notation used in place of $\bigwedge$. Instead of writing the index range as a subscript of $\bigwedge$, it is written after the `CONJ`. At parse time, this entire statement is converted to

```
CONJ i:{'A', 'B', 'C'} . (fni j . x[j] + 1 <= y[j])[i]
```

which is in the form a proposition c as allowed in the core theory (modulo conversion between ASCII and LaTeX symbols).

The general form of an abbreviation that can be written in between the `let` and `in` is

```
s x = a
```

where `s` stands for a category name (so far we have introduced `expr`, `setf`, and `propf`), `x` is any alphanumeric symbol, and `a` is the item the name is being defined for. A few additional features are useful in some cases.

It would be convenient to use the keyword `set` instead of `setf`. However, it would be misleading to allow `set` to refer to any σ because some σ are not sets. Somehow the keyword `set` should refer only to those σ that are of kind `IndexSet`. We do this be treating the statement `set X = sigma` as a synonym for `setf X = sigma : IndexSet`. Similarly, `prop p_X = c` is a synonym for `propf p_X = c : Prop`. The keywords `set` and `prop` are specializations of the general keywords `setf` and `propf`, respectively.

An important convenience for function notation is the ability to pass what appear to be multiple arguments, even though formally functions accept only a single argument. For example, $\lambda i.(x[i.1] + x[i.2])$ effectively takes in two arguments. In the concrete syntax, this would be written

```
fni i . (x[#1 i] + x[#2 i])
```

It would be even more convenient to write

```
fni (j, k) . (x[j] + x[k]
```

We allow this by treating it as

```
fni i . let
            j = #1 i
            k = #2 i
        in
            x[i] + x[j]
        end
```

In other words, multiple arguments are a shorthand for declaring abbreviations with a let construct.

The general idea is to allow a pattern in place of the single variable as the argument to a function. The syntax for a pattern is

$$\pi ::= x \mid \_ \mid (\pi_1, \ldots, \pi_m) \tag{C.1}$$

A pattern can either be a single variable, an underscore, or a list of patterns. An underscore is useful because a function accepting a tuple might not need all components of that tuple. For example,

```
fni (j, _) . (x[j] * y[j])
```

is a function that uses only its first argument. The second argument could have been named and not used, but an underscore is more clear.

Finally, the following tables summarize the correspondence between the notation used in the theory, and that admitted in the concrete syntax. Only the label version of case constructs are shown; case constructs with integer handles are analogous.

Index Expressions	
LaTeX	ASCII
x	`x`
l	`'l'`
k	`k`
$(\varepsilon_1, \ldots, \varepsilon_m)$	`(epsilon1, ..., epsilonm)`
$\varepsilon.k$	`#k epsilon`
$-\varepsilon$	`-epsilon`
$\varepsilon_1 + \varepsilon_2$	`epsilon1 + epsilon2`
$\varepsilon_1 - \varepsilon_2$	`epsilon1 - epsilon2`
$\varepsilon_1 * \varepsilon_2$	`epsilon1 * epsilon2`

$\mathtt{case}\ \varepsilon\ \mathtt{of}\ \{l_j \Rightarrow \varepsilon_j\}_{j=1}^{m}$	`case epsilon of`
	`'l1' epsilon1`
	`| ...`
	`| 'lm' epsilonm`

Index Types	
LaTeX	ASCII
$\{l_1, \ldots, l_m\}$	`{'l1', ..., 'lm'}`
$[\varepsilon_L, \varepsilon_U]$	`{epsilonL, ..., epsilonU}`
$x_1 : \sigma_1 \times \cdots \times x_m : \sigma_m$	`(x1:sigma1 * ... * xm:sigmam)`

$\mathtt{case}\ \varepsilon\ \mathtt{of}\ \{l_j \Rightarrow \sigma_j\}_{j=1}^{m}$	`case epsilon of`
	`'l1' sigma1`
	`| ...`
	`| 'lm' sigmam`

$i : \sigma_1 \rightarrow \sigma_2$	`i:[sigma1] -> sigma2`
$\lambda i\,.\,\sigma$	`fni i sigma`
$\sigma\,\varepsilon$	`sigma[epsilon]`
$\sigma :: \kappa$	`sigma:kappa`

Index Kinds	
LaTeX	ASCII
`IndexSet`	`set`
$i : \sigma \rightarrow \kappa$	`i:[sigma] -> kappa`

Expressions	
LaTeX	ASCII
x	`x`
r	`r`
`true`	`true`
`false`	`false`
$-e$	`~e`
$e_1 + e_2$	`e1 + e2`
$e_1 - e_2$	`e1 - e2`
$e_1 * e_2$	`e1 * e2`
`not` e	`not e`
e_1 `or` e_2	`e1 or e2`
e_1 `and` e_2	`e1 and e2`
`not` e_1 `or` e_2	`e1 implies e2`
$\sum_{i:\sigma} e$	`SUM i:sigma e`

`case`$_{i.\tau}$ ε `of` $\{l_j \Rightarrow e_j\}_{j=1}^{m}$	`case_{i.tau} epsilon of`
	`'l1' e1`
	`| ...`
	`| 'lm' em`

λi`.` e	`fni i e`
$e\,\varepsilon$	`e[epsilon]`
$e : \tau$	`e:tau`

Types	
LaTeX	ASCII
`real`	`real`
`bool`	`bool`
$i : \sigma \to \tau$	`i:[sigma] -> tau`

Propositions	
LaTeX	ASCII
`T`	`truth`
`F`	`falsehood`
`isTrue` e	`isTrue e`
$e_1 = e_2$	`e1 = e2`
$e_1 \leq e_2$	`e_1 <= e_2`
$e_1 \leq e_2$	`e_2 >= e_1`
$\vee$	`disj`
$\wedge$	`conj`
$\exists$	`exists`
$\bigvee_{i:\sigma} c$	`DISJ i:sigma  c`
$\bigwedge_{i:\sigma} c$	`CONJ i:sigma  c`

$\texttt{case}_{i.\zeta}\ \varepsilon\ \texttt{of}\ \{l_j \Rightarrow c_j\}_{j=1}^{m}$	`case_{i.zeta} epsilon of`
	`  'l1'  c1`
	`  |  ...`
	`  | 'lm'  cm`

$\lambda i \,.\, c$	`fni i  c`
$c\,\varepsilon$	`c[epsilon]`
$c : \zeta$	`c:zeta`

Propositional Types	
LaTeX	ASCII
`Prop`	`prop`
$i : \sigma \to \zeta$	`i:[sigma] -> zeta`

Programs	
LaTeX	ASCII

$\delta_{x_1:\tau_1,\ldots,x_m:\tau_m}\ \{e \mid c\}$	`var x1:tau1`
	`...`
	`var xm:taum`
	`dir e subject_to`
	`c`

Appendix D

Data

Disjunctive Constraint in OPL

Input C++ file to OPL

```
#include <ilcplex/ilocplex.h>

main(int argc, char **argv) {

  IloEnv env;
  IloModel model(env);

  IloNumVar x1(env,"x1");
  IloNumVar x2(env,"x2");

  model.add((x1 + x2 <= 900000) || (x1 + x2 >= 1000000));

  IloCplex cplex(model);
  cplex.exportModel("OPLDisj.out.lp");
  env.end();
}
```

Output LP file generated by OPL

```
\Problem name: CPLEX solver

Minimize
 obj:
Subject To
 id10: x1 + x2 + 100000 id8 <= 1000000
 id12: x1 + x2 + 2000000 id8 >= 1000000
Bounds
      x1 Free
      x2 Free
```

```
 0 <= id8 <= 1
Generals
 id8
End
```

Disjunctive Constraint in LogMIP

Input LogMIP file

```
BINARY VARIABLES y;
VARIABLES x1,x2;
VARIABLE z;

equations eq1,eq2,eq3,eq4;
eq1.. x1 + x2 =l= 5.0;
eq2.. x1 + x2 =g= 10.0;
eq3.. y =g= 0;
eq4.. z =e= x1 + x2;

x1.lo = 1.0;
x1.up = 100.0;

option mip=osl;
option limcol=0;
option limrow=0;
option optcr=0.;
option optca=0.;


$ONTEXT BEGIN LOGMIP
DISJUNCTION D1;

D1 IS
IF Y THEN
        eq1;
ELSE
        eq2;
ENDIF;

$OFFTEXT END LOGMIP

OPTION MIP=LOGMIPC;
MODEL TEST /ALL/;
SOLVE TEST USING MIP MINIMIZING Z;
display z.l, x1.l, x2.l, y.l;
```

Output GAMS file generated by LogMIP

```
*** VARIABLES DEFINITION CORRESPONDING TO THE ORIGINAL MINLP ***

 BINARY Variables
y;
 Variables
x1, x2, z;
 VARIABLE LGM(*,*,*);


*** EQUATIONS independentof discrete choices ***

 EQUATIONS
  eq1,  eq2,



*** EQUATIONS corresponding to disjunction terms ***

 eqdis1,  eqdis2,


*** EQUATIONS added to generate de CONVEX HULL ***


 vdisag1, vdisag2,
 bound1, bound2, bound3, bound4, bound5, bound6, bound7, bound8 ;
$inlinecom { }
*//* WRITING INDEPENDENT CONSTRAINTS
 eq1 {eq3} ..  + y =G= 0;

eq2 {eq4} ..  - x1 - x2 + z =E= 0;

*//* WRITING DISJUNCTIONS'S CONSTRAINTS
 eqdis1 {eq1} ..  + lgm('1','1','1') + lgm('1','1','2') =L= 5.00000 * y ;

eqdis2 {eq2} ..  + lgm('1','2','1') + lgm('1','2','2') =G= 10.00000 * (1-y );

vdisag1..x1 =E= + lgm('1','1','1')+ lgm('1','2','1');
vdisag2..x2 =E= + lgm('1','1','2')+ lgm('1','2','2');
bound1..lgm('1','1','1') =G= x1.LO * y ;
bound2..lgm('1','1','1') =L= x1.UP * y ;
bound3..lgm('1','2','1') =G= x1.LO * (1-y );
bound4..lgm('1','2','1') =L= x1.UP * (1-y );
bound5..lgm('1','1','2') =G= x2.LO * y ;
bound6..lgm('1','1','2') =L= x2.UP * y ;
bound7..lgm('1','2','2') =G= x2.LO * (1-y );
```

```
47 bound8..lgm('1','2','2') =L= x2.UP * (1-y );
48  SCALAR UPB/10000/;
49  SCALAR LOB /0.01/;
50  x1.UP = 100;
51  x1.LO = 1;
52  x2.UP = UPB;
53  x2.LO = LOB;
54  OPTION ITERLIM = 10000;
55  OPTION OPTCA = 0;
56  OPTION OPTCR = 0;
57  OPTION RESLIM =  1000.0000;
58  OPTION LIMCOL = 0;
59  OPTION LIMROW = 0;
60  OPTION MIP = CPLEX     ;
61 MODEL ConvexHull /ALL/;
62
63   SOLVE ConvexHull MINIMIZING z USING MIP;
64  FILE RESULTS /C:\WINDOWS\gamsdir\225a\PUTfile.scr/;
65  PUT RESULTS;
66  PUT Convexhull.MODELSTAT;
67  PUT Convexhull.SOLVESTAT;
68  PUT Convexhull.ITERUSD;
69  PUT Convexhull.RESUSD;
70  PUT z.L /;
71  PUT eqdis1.L; PUT eqdis1.M /;
72  PUT eqdis2.L; PUT eqdis2.M /;
73  PUT eq1.L; PUT eq1.M /;
74  PUT eq2.L; PUT eq2.M /;
75  PUT y.L ; PUT y.M /;
76  PUT x1.L ; PUT x1.M /;
77  PUT x2.L ; PUT x2.M /;
78  PUT z.L ; PUT z.M /;
```

Data for Figures of Switched Flow Process

Figure 9.2a

i	t	Q^α	Q^β	M	C	R	S
1	0.0	on	hi	20.0	0.0	0.0	0.0
1	10.0	on	hi	62.0	250.0	10.0	10.0
2	10.0	off	lo	62.0	250.0	0.0	0.0
2	50.0	off	lo	10.0	330.0	40.0	40.0
3	50.0	on	lo	10.0	380.0	0.0	40.0
3	60.0	on	lo	17.0	500.0	10.0	50.0
4	60.0	off	lo	17.0	500.0	0.0	50.0
4	65.4	off	lo	10.0	510.8	5.4	55.4
5	65.4	off	hi	10.0	550.8	5.4	0.0
5	75.0	off	hi	31.2	695.0	15.0	9.6

Figure 9.2b

i	t	Q^α	Q^β	M	C	R	S
1	0.0	on	hi	20.0	0.0	0.0	0.0
1	30.0	on	hi	146.0	750.0	30.0	30.0
2	30.0	off	lo	146.0	750.0	0.0	0.0
2	40.0	off	lo	133.0	770.0	10.0	10.0
3	40.0	off	lo	133.0	770.0	10.0	10.0
3	60.0	off	lo	107.0	810.0	30.0	30.0
4	60.0	off	hi	107.0	850.0	30.0	0.0
4	68.0	off	hi	124.6	970.0	38.0	8.0
5	68.0	off	lo	124.6	970.0	38.0	0.0
5	75.0	off	lo	115.5	984.0	45.0	7.0

Figure 9.3a

i	t	Q^{α}	Q^{β}	M	C	R	S
1	0.0	off	hi	20.0	0.0	0.0	0.0
1	40.0	off	hi	108.0	600.0	40.0	40.0
2	40.0	off	lo	108.0	600.0	40.0	0.0
2	46.38	off	lo	99.7	612.7	46.4	6.4
3	46.38	off	hi	99.7	652.7	46.4	0.0
3	69.2	off	hi	150.0	995.6	69.2	22.9
4	69.2	off	lo	150.0	995.6	69.2	0.0
4	176.9	off	lo	10.0	1211.0	176.9	107.7
5	176.9	off	hi	10.0	1251.0	176.9	0.0
5	216.9	off	hi	98.0	1851.0	216.9	40.0
6	216.9	off	lo	98.0	1851.0	216.9	0.0
6	244.6	off	lo	62.0	1906.4	244.6	27.7
7	244.6	off	hi	62.0	1946.4	244.6	0.0
7	284.6	off	hi	150.0	2546.4	284.6	40.0
8	284.6	off	lo	150.0	2546.4	284.6	0.0
8	392.3	off	lo	10.0	2761.8	392.3	107.7
9	392.3	off	hi	10.0	2801.8	392.3	0.0
9	432.3	off	hi	98.0	3401.8	432.3	40.0
10	432.3	off	lo	98.0	3401.8	432.3	0.0
10	500.0	off	lo	10.0	3537.1	500.0	67.7

Figure 9.3b

i	t	Q^α	Q^β	M	C	R	S
1	0.0	on	hi	0.0	0.0	0.0	0.0
1	30.0	on	hi	180.0	750.0	30.0	30.0
2	30.0	off	lo	180.0	750.0	0.0	0.0
2	34.0	off	lo	182.0	758.0	4.0	4.0
3	34.0	on	hi	182.0	848.0	0.0	0.0
3	64.0	on	hi	362.0	1598.0	30.0	30.0
4	64.0	off	lo	362.0	1598.0	0.0	0.0
4	67.0	off	lo	363.5	1604.0	3.0	3.0
5	67.0	on	hi	363.5	1694.0	0.0	0.0
5	97.0	on	hi	543.5	2444.0	30.0	30.0
6	97.0	off	lo	543.5	2444.0	0.0	0.0
6	100.0	off	lo	545.0	2450.0	3.0	3.0
7	100.0	on	hi	545.0	2540.0	0.0	0.0
7	130.0	on	hi	725.0	3290.0	30.0	30.0
8	130.0	off	hi	725.0	3290.0	0.0	30.0
8	140.0	off	hi	765.0	3440.0	10.0	40.0
9	140.0	on	lo	765.0	3490.0	0.0	0.0
9	170.0	on	lo	840.0	3850.0	30.0	30.0
10	170.0	off	hi	840.0	3890.0	0.0	0.0
10	210.0	off	hi	1000.0	4490.0	40.0	40.0

Notation

Chapter 2

$\mathbb{N}$	the set$\{1, \ldots, n\}$ where n is a known finite constant
t_i^s	time at start of interval i
t_i^e	time at end of interval i
Δt_i	length of timeline interval i
$\mathcal{T}$	set of timeline intervals
(i, t)	an integer-real time point
$\mathbb{T}$	set of time points, i.e. a timeline
$\preceq$	total order relation on $\mathbb{T}$
$\mathbb{Q}^a$	set of finite domain constants for automaton a
$\mathbf{Q}$	set of finite domain variables
$\mathbf{X}$	set of real-valued variables
$\mathbf{t}$	set of timeline variables
$\mathbf{Q}(i)$	set of each variable in $\mathbf{Q}$ applied to i
$\mathbf{X}(i, t)$	set of each variable in $\mathbf{X}$ applied to (i, t)
$\mathcal{L}(\mathbf{Q}(i))$	set of finite domain constraints involving any of the variables in $\mathbf{Q}$ applied to i
$\mathcal{L}(\mathbf{X})$	set of (in)equations involving any of the variables in $\mathbf{X}$ applied to (i, t)
$\mathcal{L}(\mathbf{t})$	set of (in)equations on timeline variables
$(n, G_t, \mathbf{X}, Aut, G_V)$	LCCA model
n	number of intervals in timeline
G_t	constraints on timeline variables
Aut	set of component automata
G_V	constraints involving $\mathbf{Q}$ and $\mathbf{X}$
$(\mathbb{Q}, \bar{\mathbf{x}}, \hat{\mathbf{x}}, F, Arc)$	component automata
$\bar{\mathbf{x}}$	set of given rates
$\hat{\mathbf{x}}$	set of jump variables
$F(q)$	invariant for mode q
Arc	set of transitions
$\gamma_{(q,q')}$	guard on transition from q to q'
$\rho_{(q,q')}$	reset on transition from q to q'
$\chi(i, t)$	point in the state space

$[\chi(i, t_i^s) \longrightarrow \chi(i, t_i^e)]$	continuous trajectory
$\langle \chi(i,t) \mapsto \chi(i+1,t) \rangle$	discrete step
ξ	hybrid trajectory
$\Xi_{(n,G_t,\mathbf{X},Aut,G_V)}$	set of hybrid trajectories for model $(n, G_t, \mathbf{X}, Aut, G_V)$

Chapter 3

$X^s(i)$	value of variable X at start of interval i
$X^e(i)$	value of variable X at end of interval i
$\bar{w}(i)$	variable equal to $\bar{x}(Q(i))$
$Y(q,i)$	Boolean variable corresponding to $Q(i) = q$
$YY^a(i)$	true if automata a makes dummy transition at event i
$YYY(i)$	true if all automata make dummy transition at event i
$Z^a(q, q', i)$	true if automaton a transitions from q in interval i to q' in interval $i+1$

Chapter 4

τ	type
ρ	refined type
e	expression
c	proposition
p	mathematical program
Γ	context of mathematical programming variables
Υ	refined context, list of variables with declared refined types
$FV(s)$	free variables of s, where s is e, c, or p
s CLOSED	s has no free variables, where s is e, c, or p
$\{e/x\}\, s'$	substitution of e for x in s', where s' is e, c, or p
τ TYPE	τ is a well-formed type
Γ CTXT	Γ is a well-formed context
$\Gamma \vdash e : \tau$	in context Γ, e is of type τ
$\Gamma \vdash c$ PROP	in context Γ, c is a well-formed proposition
p MP	p is a well-formed mathematical program
$\rho \subseteq \tau$	refined type ρ is a subset of type τ
$x : \rho \backsimeq c$	the declaration $x : \rho$ corresponds to the proposition c
$\Gamma(\Upsilon)$	context corresponding to refined context Υ
ρ BOUNDED	refined type ρ is bounded
e CANONICAL	e is an irreducible expression
v	a canonical expression
$e \searrow v$	e evaluates to v
c TRUE	proposition c is true
r_{option}	an optional real number
$p \twoheadrightarrow r_{\text{option}}$	the solution to program p is r_{option}

Ψ	valuation, list of variables with assigned values
$\Psi \vdash e \searrow v$	under valuation Ψ, e evaluates to v
$\Psi \vdash c$ TRUE	under valuation Ψ, c is true

Chapter 5

s^{MIP}	s is a MIP construct, where s is any of τ, ρ, e, c, p, Υ, Ψ, or v
e LINEAR	expression e can be transformed into a linear expression
c LINEAR	expressions within proposition c can be transformed to linear expressions
p LINEAR	expressions within program p can be transformed to linear expressions
p MILP	p is a mixed-integer linear program
e LITERAL	e is a literal expression
e DLF	e is in disjunctive literal form
e CNF	e is in conjunctive normal form
e CONJ	e is in CNF but not in DLF
$e_1 \curvearrowright e_2$	e_1 can be converted to the CNF expression e_2
$e_1 \curvearrowright_* e$	non-CNF expression e_1 can be converted to the CNF expression e_2
$\Upsilon \vdash c$ DISJVARSBOUNDED	Υ contains bounds for all variables free in or existentially introduced within any of the disjuncts in c
c EXISTVARSBOUNDED	all variables existentially introduced within c are bounded
p DISJVARSBOUNDED	all variables in all disjunctions in program p have known bounds
$\tau \overset{\text{TYPE}}{\longmapsto} \tau^{\text{MIP}}$	type compiler
$\rho \overset{\text{RTYPE}}{\longmapsto} \rho^{\text{MIP}}$	refined type compiler
$e \overset{\text{DLF}}{\longmapsto} e^{\text{MIP}}$	DLF expression compiler
$e \overset{\text{CONJ}}{\longmapsto} c^{\text{MIP}}$	CONJ expression compiler
$\Upsilon \vdash c \overset{\text{PROP}}{\longmapsto} c^{\text{MIP}}$	proposition compiler
$\Upsilon \vdash c \multimap c'$	add bounding propositions to c for all variables free in c
$e \circledast e_1 \hookrightarrow e_2$	multiply e to constant terms in e_1
$e \circledast c_1 \hookrightarrow c_2$	multiply e to constant terms in c_1
$\Upsilon \vdash c \overset{\text{DISJ}}{\longmapsto} c^{\text{MIP}}$	disjunctive proposition compiler
$p \overset{\text{PROG}}{\longmapsto} p^{\text{MIP}}$	program compiler
$\Upsilon \overset{\text{CTXT}}{\longmapsto} \Upsilon^{\text{MIP}}$	refined context compiler
$\Psi \overset{\text{VAL}}{\longmapsto} \Psi^{\text{MIP}}$	valuation compiler

Chapter 6

ε	index expression
σ	index types

κ	index kinds
Δ	context of index variables
$FV\,(s)$	free variables of s, where s is ε, σ, or κ
s CLOSED	s has no free variables, where s is ε, σ, or κ
$\{\varepsilon/x\}\,s$	substitution of ε for x in s, where s is ε, σ, κ, or Δ
s CANONICAL	s is irreducible, where s is ε, σ, or κ
η	canonical index expression
$s_1 \searrow s_2$	s_1 evaluatess to s_2, where s is ε, σ, or κ
Φ	index valuation, list of index variables with assigned values
$\Phi \vdash s_1 \searrow s_2$	under valuation Φ, s_1 evaluates to s_2, where s is ε, σ, or κ
$\vdash^c \kappa$ KIND	κ is a well-formed canonical kind
$\vdash^c \sigma :: \kappa$	σ is a canonical type of canonical kind κ
$\vdash^c \varepsilon : \sigma$	ε is a canonical element of canonical type σ
$\vdash^c \sigma_1 \leq: \sigma_2$	canonical type σ_1 is a subtype of canonical type σ_2
$\vdash^c \sigma_1 \equiv \sigma_2 :: \kappa$	canonical type σ_1 is equivalent to canonical type σ_2
$\vdash^c \kappa_1 \leq:: \kappa_2$	κ_1 is a subkind of κ_2
$\vdash^c \kappa_1 \equiv \kappa_2$	kinds κ_1 and κ_1 are equivalent
$\vdash^c \varepsilon_1 \leq \varepsilon_2$	ε_1 is less than or equal to ε_2
$\vdash^c \varepsilon_1 \equiv \varepsilon_2 : \sigma$	expressions ε_1 and ε_2 are equivalent at type σ
$\vdash^q J$	corresponding judgement on closed forms
$\Delta \vdash J$	corresonding judgement on open forms
$S^q\,(\sigma)$	set of canonical elements of closed type σ
$S^c\,(\sigma)$	set of canonical elements of canonical type σ

Chapter 7

τ	indexed type
ρ	indexed refined type
e	indexed expression
c	indexed proposition
ζ	indexed propositional type
p	indexed mathematical program
Γ	context of mathematical programming variables
Υ	refined context, list of variables with declared refined types
s CLOSED	s has no free variables, where s is any syntactic construct
$\{s/x\}\,s'$	substitution of s for x in s', where s and s' are any syntactic constructs
$\vdash_\Delta \tau$ TYPE	τ is a well-formed type
$\vdash_\Delta \Gamma$ CTXT	Γ is a well-formed context
$\vdash_\Delta \tau_1 \equiv \tau_2$	types τ_1 and τ_1 are equivalent
$\Gamma \vdash_\Delta e : \tau$	e is of type τ
$\Gamma \vdash_\Delta e \downarrow \tau$	type analysis
$\Gamma \vdash_\Delta e \uparrow \tau$	type synthesis

$\vdash_\Delta \zeta$ PROP_TYPE	propositional type ζ is well-formed
$\Gamma \vdash_\Delta c : \zeta$	c is of propositional type ζ
p MP	p is a well-formed indexed mathematical program
$\rho \subseteq \tau$	refined type ρ is a subset of type τ
$x : \rho \backsimeq c$	the declaration $x : \rho$ corresponds to the proposition c
$\Gamma(\Upsilon)$	context corresponding to refined context Υ
ρ BOUNDED	refined type ρ is bounded
e CANONICAL	e is an irreducible expression
v	a canonical expression
$e \searrow v$	e evaluates to v
c CANONICAL	c is a canonical proposition
$c_1 \searrow c_2$	c_1 evaluates to c_2
c TRUE	proposition c is true
r_{option}	an optional real number
$p \twoheadrightarrow r_{\text{option}}$	the solution to program p is r_{option}
Ψ	valuation, list of variables with assigned values
$\Psi \vdash_\Phi e \searrow v$	under valuation Ψ and Φ, e evaluates to v
$\Psi \vdash_\Phi c$ TRUE	under valuation Ψ and Φ, c is true

Chapter 8

e^{ANF}	e is in application normal form
$e_1 \rightsquigarrow_{\text{hr}} e_2$	one-step head reduction
$e_1 \not\rightsquigarrow_{\text{hr}}$	e_1 does not head reduce
$e_1 \rightsquigarrow e_2$	convert e_1 into application normal form e_2
s^{MIP}	s is a MIP construct,
	where s is any of τ, ρ, e, c, ζ, p, Υ, Ψ, or v
e LINEAR	expression e can be transformed into a linear expression
c LINEAR	expressions within proposition c can be transformed to linear expressions
p LINEAR	expressions within program p can be transformed to linear expressions
e LITERAL	e is a literal expression
e DLF	e is in disjunctive literal form
e CNF	e is in conjunctive normal form
e CONJ$'$	e is in CNF but not in DLF
e CONJ	e is in CNF but not in DLF
$e_1 \curvearrowright e_2$	e_1 can be converted to the CNF expression e_2
$e_1 \curvearrowright_* e$	non-CNF expression e_1 can be converted to the CNF expression e_2
$\texttt{not}\, e \overset{\text{not}}{\curvearrowright_*} e'$	CNF of `not` expressions
$(e_1 \,\texttt{or}\, e_2) \overset{\text{or}}{\curvearrowright_*} e'$	CNF of `or` expressions
$(e_1 \,\texttt{or}\, e_2) \overset{\text{or-conj}}{\curvearrowright_*} e'$	CNF of $e_1 \,\texttt{or}\, e_2$, where e_1 CONJ

$\Upsilon \vdash c$ DisjVarsBounded	Υ contains bounds for all variables free in or existentially introduced within any of the disjuncts in c
c ExistVarsBounded	all variables existentially introduced within c are bounded
p DisjVarsBounded	all variables in all disjunctions in program p have known bounds
$\tau \overset{\text{TYPE}}{\longmapsto} \tau^{\text{MIP}}$	type compiler
$\rho \overset{\text{RTYPE}}{\longmapsto} \rho^{\text{MIP}}$	refined type compiler
$e \overset{\text{DLF}}{\longmapsto} e^{\text{MIP}}$	DLF expression compiler
$e \overset{\text{CONJ}}{\longmapsto} c^{\text{MIP}}$	CONJ expression compiler
$\Upsilon \vdash c \overset{\text{PROP}}{\longmapsto} c^{\text{MIP}}$	proposition compiler
$\Upsilon \vdash c \multimap c'$	add bounding propositions to c for all variables free in c
$e \circledast e_1 \hookrightarrow e_2$	multiply e to constant terms in e_1
$e \circledast c_1 \hookrightarrow c_2$	multiply e to constant terms in c_1
$x : \rho^{\text{MIP}} \Rrightarrow x'_{i:\sigma} \looparrowright c$	relate original variables x and disaggregated variables x' in c
$\Upsilon \vdash c \overset{\text{DISJ}}{\longmapsto} c^{\text{MIP}}$	disjunctive proposition compiler
$p \overset{\text{PROG}}{\longmapsto} p^{\text{MIP}}$	program compiler
$\Upsilon \overset{\text{CTXT}}{\longmapsto} \Upsilon^{\text{MIP}}$	refined context compiler
$\Psi \overset{\text{VAL}}{\longmapsto} \Psi^{\text{MIP}}$	valuation compiler

Appendix B

c	syntactic category
v	variable
t	term
o	term producing operator
p	list of arguments to term producing operator
θ	type of p
$v = v'$	variable equality
$v \ltimes c$	variable v is of syntactic category c
$\theta_1 \equiv \theta_2$	θ_1 and θ_2 are equivalent
$p : \theta$	p is of type θ
$t \ltimes c$	term t is well-formed and of syntactic category c
$FV\,(c, t)$	free c variables of term t
$FV_{\#}\,(c, p)$	free c variables of p
$\{t/v\}\,t'$	substitution of t for v in t'
$\{t/v\}_{\#}\,p$	substitution of t for v in p
$(t_1, t_2) \rightarrowtail^{c} (t'_1, t'_2)$	α-match the c variables of t_1 and t_2
$(p_1, p_2) \rightarrowtail^{c}_{\#} (p'_1, p'_2)$	α-match the c variables of p_1 and p_2

Acronyms

LCCA	linear coupled component automata
CP	constraint program(ming)
DAE	differential-algebraic equation
STN	state-task network
CNF	conjunctive normal form
DLF	disjunctive literal form
DNF	disjunctive normal form
IH	inductive hypothesis
MP	mathematical program(ming)
LP	linear program(ming)
MIP	mixed-integer program(ming)
MILP	mixed-integer linear program(ming)
MINLP	mixed-integer nonlinear program(ming)
DP	disjunctive program(ming)
GDP	generalized disjunctive program(ming)
HA	hybrid automata
ANF	application normal form

Bibliography

Abdeddaim, Y. and Maler, O. (2001). Job-shop scheduling using timed automata, *in* G. Berry, H. Comon and A. Finkel (eds), *Proceedings of the13th International Conference on Computer Aided Verification, CAV 2001.*, Vol. 2102 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 478–492. 7

Abdeddaim, Y. and Maler, O. (2002). Preemptive job-shop scheduling using stopwatch automata, *Tools and Algorithms for the Construction and Analysis of Systems. 8th International Conference, TACAS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002*, Vol. 2280 of *Lecture Notes in Computer Science*, Springer-Verlag, Grenoble, France, pp. 113–126. 7

Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T. A., Ho, P. H., Nicollin, X., Olivero, A., Sifakis, J. and Yovine, S. (1995). The algorithmic analysis of hybrid systems, *Theoretical Computer Science* **138**(1): 3–4. 6, 7, 21

Alur, R. and Dill, D. L. (1994). A theory of timed automata, *Theoretical Computer Science* **126**(2): 183–235. 6

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. 15, 48, 201

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, Berlin, pp. 21–36. 176

Asarin, E. and Maler, O. (1999). Optimal control for timed automata, *in* H. F. Chen, D. Z. Cheng and J. F. Zhang (eds), *14th World Congress of the International Federation of Automatic Control*, Vol. 6, Elsevier Science, Beijing, China, pp. 1–6. 7

Asarin, E., Maler, O. and Pnueli, A. (1995). Reachability analysis of dynamical systems having piecewise-constant derivatives, *Theoretical Computer Science* **138**(1): 35–65. 7

Aubin, J. P., Lygeros, J., Quincampoix, M., Sastry, S. and Seube, N. (2002). Impulse differential inclusions: A viability approach to hybrid systems, *IEEE Transactions on Automatic Control* **47**(1): 2–20. 21

Avraam, M. P., Shah, N. and Pantelides, C. C. (1998). Modelling and optimisation of general hybrid systems in the continuous time domain, *Computers & Chemical Engineering* **22**: S221–S228. 34

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). 4, 185, 186, 187, 188, 189, 190, 222

Balas, E. (1979). Disjunctive programming, *Annals of Discrete Mathematics* **5**: 3–51. 190

Balas, E. (1985). Disjunctive programming and a hierarchy of relaxations for discrete optimization problems, *Siam Journal on Algebraic and Discrete Methods* **6**(3): 466–486. 4

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). See also the foreword by Cornuéjols and Pulleyblank (1998). 222, 223

Barendregt, H. P. (1984). *The lambda calculus: its syntax and semantics*, Vol. 103 of *Studies in logic and the foundations of mathematics*, rev. edn, Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co. 17, 127

Barton, P. I. (1992). *The modelling and simulation of combined discrete/continuous processes*, PhD thesis, Imperial College. 7

Barton, P. I. and Pantelides, C. C. (1994). Modeling of combined discrete-continuous processes, *AIChE Journal* **40**(6): 966–979. 6

Bemporad, A. and Morari, M. (1999). Control of systems integrating logic, dynamics, and constraints, *Automatica* **35**(3): 407–427. 6, 188

Bisschop, J. and Meeraus, A. (1979). Selected aspects of a general algebraic modeling language, *in* K. Iracki, K. Malanowski and S. Walukiewicz (eds), *9th IFIP Conference on Optimization Techniques: Part 2*, Vol. 23 of *Lecture Notes in Control and Information Sciences*, Springer-Verlag, Warsaw, pp. 223–233. 12

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. 11

Bixby, R. (2002). Solving real-world linear programs: a decade and more of progress, *Operations Research* **50**(1): 3–15. 7

Brooke, A., Kendrick, D., Meeraus, A. and Raman, R. (1998). *GAMS: A User's Guide*, GAMS Development Corporation, Washington, D.C. 12

Brouwer, L. E. J. (1907). *Over de Grondslagen der Wiskunde*, PhD thesis, University of Amsterdam. 16

Cassez, F. and Larsen, K. (2000). The impressive power of stopwatches, *in* C. Palamidessi (ed.), *Proceedings of the 11th International Conference on Concurrency Theory, CONCUR 2000.*, Vol. 1877 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, pp. 138–152. 6

Chang, C.-L. and Lee, R. C.-T. (1987). *Symbolic logic and mechanical theorem proving*, Computer science classics, Academic Press, San Diego. 71, 189

Chutinan, A. and Krogh, B. H. (2003). Computational techniques for hybrid system verification, *IEEE Transactions on Automatic Control* **48**(1): 64–75. 7

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. 11, 17

Cornuéjols, G. and Pulleyblank, W. R. (1998). Foreword to Balas (1998), *Discrete Applied Mathematics* **89**(1–3): 1–2. 222

Curry, H. B. (1963). *Foundations of mathematical logic*, McGraw-Hill series in higher mathematics, McGraw-Hill. 14

DeCarlo, R. A., Branicky, M. S., Pettersson, S. and Lennartson, B. (2000). Perspectives and results on the stability and stabilizability of hybrid systems, *Proceedings of the IEEE* **88**(7): 1069–1082. 7

Dummett, M. A. E. (1977). *Elements of intuitionism*, Oxford logic guides, Clarendon Press, Oxford. With the assistance of Roberto Minio. 16

Fourer, R. and Gay, D. M. (2002). Extending an algebraic modeling language to support constraint programming, *Informs Journal on Computing* **14**(4): 322–344. 12

Fourer, R., Gay, D. M. and Kernighan, B. W. (1990). A modeling language for mathematical programming, *Management Science* **36**(5): 519–554. 11

Fourer, R., Gay, D. M. and Kernighan, B. W. (2003). *AMPL: a modeling language for mathematical programming*, 2nd edn, Thomson/Brooks/Cole, Pacific Grove, CA. 12

Frege, F. L. G. (1893). *Grundgesetze der Arithmetik, Band I*, Verlag Herman Pohle, Jena. Partial translation available as Frege (1964). 15, 223

Frege, F. L. G. (1903). *Grundgesetze der Arithmetik, Band II*, Verlag Herman Pohle, Jena. Translation of epilogue available in Frege (1964). 223

Frege, F. L. G. (1964). *The basic laws of arithmetic: Exposition of the system*, University of California Press, Berkeley. Translation of the introductory portions of Frege (1893) and an epilogue appended to Frege (1903). 223

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. 14, 17

Harper, R. W. (2005). Programming languages: Theory and practice. Unpublished text. Available online at http://www-2.cs.cmu.edu/∼rwh. 17, 192

Heemels, W. P. M. H., De Schutter, B. and Bemporad, A. (2001). On the equivalence of classes of hybrid dynamical models, *Proceedings of the 40th IEEE Conference on Decision and Control*, Vol. 1, IEEE, Orlando, FL, USA, pp. 364–369. 8

Henzinger, T. A. (1996). The theory of hybrid automata, *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, IEEE Comput. Soc. Press, pp. 278–292. 7

Hooker, J. (2000). *Logic-based methods for optimization: combining optimization and constraint satisfaction*, Wiley-Interscience series in discrete mathematics and optimization, John Wiley & Sons. 185, 188

Hooker, J. N. and Osorio, M. A. (1999). Mixed logical-linear programming, *Discrete Applied Mathematics* **97**: 395–442. 4

Ierapetritou, M. G. and Floudas, C. A. (1998). Effective continuous-time formulation for short-term scheduling. 1. multipurpose batch processes, *Industrial & Engineering Chemistry Research* **37**(11): 4341–4359. 5

Jaffar, J. and Lassez, J.-L. (1987). Constraint logic programming, *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, ACM, Munich, West Germany, pp. 111–119. 15

Johnson, S. C. (1979). Yacc: yet another compiler-compiler, *UNIX Programmer's Manual*, Vol. 2, Holt, Rinehart, and Winston, New York, NY, USA, pp. 353–387. 201

Kallrath, J. (2000). Mixed integer optimization in the chemical process industry - experience, potential and future perspectives, *Chemical Engineering Research & Design* **78**(A6): 809–822. 5

Kallrath, J. (2004). *Modeling languages in mathematical optimization*, Vol. 88 of *Applied optimization*, Kluwer Academic Publishers, Boston. 11

Kondili, E., Pantelides, C. C. and Sargent, R. W. H. (1993). A general algorithm for short-term scheduling of batch operations. I. MILP formulation, *Computers & Chemical Engineering* **17**(2): 211–227. 5

Krogh, B. H. (2000). Hybrid systems: State of the art and perspectives, *18th Brazilian Congress of Automatic Control*, Florianopolis, SC, Brazil. 7

Lee, C. K., Singer, A. B. and Barton, P. I. (2004). Global optimization of linear hybrid systems with explicit transitions, *Systems & Control Letters* **51**(5): 363–375. 8

Lesk, M. E. and Schmidt, E. (1978). Lex: a lexical analyzer generator, *UNIX Programmer's Manual*, 7 edn, Bell Labs, Murray Hill, N.J. 201

Levine, J. R., Mason, T. and Brown, D. (1995). *Lex & yacc*, A Nutshell handbook;, 2nd minor corrections edn, O'Reilly & Associates, Sebastopol, CA. 201

Lygeros, J., Pappas, G. J. and Sastry, S. (1999). An introduction to hybrid system modeling, analysis, and control, *Preprints of the First Nonlinear Control Network Pedagogical School*, Athens, Greece, pp. 307–329. 21

Maravelias, C. T. and Grossmann, I. E. (2003). New general continuous-time state-task network formulation for short-term scheduling of multipurpose batch plants, *Industrial & Engineering Chemistry Research* **42**(13): 3056–3074. 5

Martin-Löf, P. (1984). *Intuitionistic type theory*, Vol. 1 of *Studies in proof theory. Lecture notes; Variation: Studies in proof theory*, Bibliopolis, Napoli. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980. 16, 84

Martin-Löf, P. (1987). Truth of a proposition, evidence of a judgment, validity of a proof, *Synthese* **73**(3): 407–420. 60

Martin-Löf, P. (1996). On the meanings of the logical constants and the justifications of the logical laws, *Nordic Journal of Philosophical Logic* **1**(1): 11–60. 60

Mitra, G., Lucas, C., Moody, S. and Hadjiconstantinou, E. (1994). Tools for reformulating logical forms into zero-one mixed integer programs, *European Journal of Operational Research* **72**(2): 262–276. 188

Nemhauser, G. L. and Wolsey, L. A. (1999). *Integer and combinatorial optimization*, Wiley-Interscience series in discrete mathematics and optimization, Wiley, New York. 4, 9, 70

Nicollin, X., Olivero, A., Sifakis, J. and Yovine, S. (1991). An approach to the description and analysis of hybrid systems, *in* R. L. Grossman, A. Nerode, A. P. Ravn and H. Rischel (eds), *Hybrid Systems Conference*, Springer-Verlag, Lyngby, Denmark, pp. 149–178. 7

Pierce, B. C. (2002). *Types and programming languages*, MIT Press, Cambridge, Mass. 12, 17

Raghunathan, A. U. and Biegler, L. T. (2003). Mathematical programs with equilibrium constraints (MPECs) in process engineering, *Computers & Chemical Engineering* **27**(10): 1381–1392. 7

Raman, R. and Grossmann, I. E. (1994). Modelling and computational techniques for logic based integer programming, *Computers & Chemical Engineering* **18**(7): 563–578. 4, 9, 35, 38, 189, 190

Russell, B. (1903). *The principles of mathematics*, University Press, Cambridge. The material on the subject originally intended to form the 2d volume was later developed into an independent work: Principia Mathematica, by A. N. Whitehead and B. Russell, published in 3 vols., Cambridge, 1910-13. 15

Saraswat, V. A. (1989). *Concurrent Constraint Programming Languages*, PhD thesis, Carnegie Mellon University. 15

Sasao, T. (1999). *Switching theory for logic synthesis*, Kluwer Academic Publishers, Boston, Mass. 188

Stursberg, O., Panek, S., Till, J. and Engell, S. (2002). Generation of optimal control policies for systems with switched hybrid dynamics, *Modelling, Analysis, and Design of Hybrid Systems*, Vol. 279 of *Lecture Notes in Control and Information Sciences*, Springer-Verlag, Berlin, pp. 337–352. 7

Torrisi, F. D., Bemporad, A. and Mignone, D. (2000). Hysdel — a tool for generating hybrid models, *Technical Report AUT00-03*, Automatic Control Lab, ETH. 8

Troelstra, A. S. and van Dalen, D. (1988). *Constructivism in mathematics: an introduction*, Vol. 1 of *Studies in logic and the foundations of mathematics v. 121*, Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co., Amsterdam, North-Holland; New York, N.Y. 16

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. 12

Vecchietti, A. and Grossmann, I. E. (2000). Modeling issues and implementation of language for disjunctive programming, *Computers & Chemical Engineering* **24**(9–10): 2143–2155. 13

Visser, E. (2001). A survey of rewriting strategies in program transformation systems, *1st International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, Vol. 57 of *Electronic Notes in Theoretical Computer Science*, Elsevier, Utrecht, Netherlands. 71, 189

Westerberg, A. W., Hutchison, H. P., Motard, R. L. and Winter, P. (1979). *Process flowsheeting*, Cambridge University Press. 5

Whitehead, A. N. and Russell, B. (1910). *Principia Mathematica*, Vol. 1, University Press, Cambridge. 15