The Inez Mathematical Programming Modulo Theories Framework

Panagiotis Manolios, Jorge Pais, and Vasilis Papavasileiou

Northeastern University *
{pete,jpais,vpap}@ccs.neu.edu

Abstract. Our Mathematical Programming Modulo Theories (MPMT) constraint solving framework extends Mathematical Programming technology with techniques from the field of Automated Reasoning, *e.g.*, solvers for first-order theories. In previous work, we used MPMT to synthesize system architectures for Boeing's Dreamliner and we studied the theoretical aspects of MPMT by means of the Branch and Cut Modulo T (BC(T)) transition system. BC(T) can be thought of as a blueprint for MPMT solvers. This paper provides a more practical and algorithmic view of BC(T). We elaborate on the design and features of Inez, our BC(T) constraint solver. Inez is an open-source, freely available superset of the OCaml programming language that uses the SCIP Branch and Cut framework to extend OCaml with MPMT capability. Inez allows users to write programs that arbitrarily interweave general computation with MPMT constraint solving.

1 Introduction

The ILP (or, more generally, Mathematical Programming) Modulo Theories (IMT or MPMT) framework accommodates Mathematical Programming (MP) instances, where some variable symbols have meaning in background first-order theories [21]. In previous work, we used this approach to solve systems architectural synthesis problems with hard real-time constraints for Boeing and we introduced the Branch and Cut Modulo T (BC(T)) architecture for solving MPMT [17, 21]. BC(T) combines Branch and Cut (B&C) with theory reasoning. B&C is the most established family of algorithms for solving ILP instances, empowering such powerful solvers as CPLEX [2], Gurobi [3], and SCIP [8].

We have formalized BC(T) as a highly non-deterministic transition system [21]. By abstracting away solver implementation details, the BC(T) transition system captures a wide range of possible implementations, and facilitates theoretical analysis. BC(T) can be thought of a design space for MPMT solvers. Implementing an MPMT solver involves zooming in on a region of this design space, with assorted performance trade-offs. To inform efficient solver design, this paper provides an algorithmic (and more deterministic) view of BC(T).

Inez [1] extends the SCIP [8] solver and we show how to implement MPMT on top of a B&C-based solver. We explain as much of the operation of the B&C

* This research was supported in part by DARPA under AFRL Cooperative Agreement No. FA8750-10-2-0233 and by NSF grants CCF-1117184 and CCF-1319580.

[1] https://github.com/vasilisp/inez

core as needed to demonstrate where theory solvers fit, with an emphasis on the interface between theory solvers and B&C. We do not cover purely internal operations of either side. For example, we treat Simplex (which handles real relaxations within B&C) purely as a black box. We use congruence closure (CC) as an example of a background procedure. Given that the core operations of CC are well-known [27], our discussion only covers the BC(T)-specific aspects. Our choice of CC is motivated by its wide applicability and by the relatively simple (but not trivial) constraints and algorithms involved.

We provide an overview of the features of Inez. Notably, Inez provides database techniques for reasoning in the presence of data [22]. Inez additionally supports user-provided axioms through *local theory extensions* [30]. Inez is implemented in OCaml, and makes extensive use of OCaml language constructs and technologies. In fact, the standard way of interacting with the solver is via scripts in a superset of OCaml. Programming with Inez is qualitatively different from programming in a standard programming language because Inez allows us to write programs that arbitrarily interleave general computation with MPMT constraint solving. To our knowledge, Inez is the first system that allows expressing constraints over uninterpreted functions within a programming language, with minimal syntactic overhead, while providing type-safety.

The rest of the paper is organized as follows. Section 2 introduces our superset of the OCaml language through a worked example, and explains how OCaml facilitates our implementation efforts. Section 3 describes the core BC(T) setup as a set of algorithms, while Section 4 discusses extensions on top of this setup. Section 5 provides an overview of related work. We conclude with Section 6. In the interest of space, we do not provide experimental results. We refer the interested readers to our previous work for a comparison of Inez against SMT solvers on instances from the SMT-LIB [21], and also on database problems [22].

2 The Inez Language

In this section, we introduce some of the most notable features of Inez by means of a worked example. We focus on the user-facing aspects of Inez, *i.e.*, on its input language, which is a superset of OCaml. Our extensions over OCaml are language constructs (and supporting APIs) for easily expressing logical constraints and seamlessly integrating with the underlying constraint solver. Building on top of OCaml allows us to provide a mixed functional and constraint language that users can utilize to express their models in a compact and self-contained way.

Inez utilizes the Camlp4 framework [1] to extend OCaml by assigning meaning to programs that are *syntactically valid* (*i.e.*, recognized by the unmodified OCaml grammar) but *semantically invalid*. The semantics of programs accepted by unmodified OCaml do not change under Inez. This design decision has multiple benefits. First, the syntax of Inez programs is natural, given that these programs are syntactically valid OCaml anyway. Also, there are no additional syntactic constructs to cause trouble for editors and other tools. Finally, the

implementation is cleaner, because all that it does is transform Abstract Syntax Trees (ASTs) produced by the Camlp4 parser.

Our integration of OCaml and constraints has great impact from a user perspective. For instance, consider a problem that depends on raw data defined and stored in a different location than the problem code, *e.g.*, in a plain text file, spreadsheet, database or web service. With Inez, data retrieval, data processing, and constraint solving can all happen side by side, in the same environment. We present concrete Inez code that obtains data from a database, defines data structures to store and manipulate this data, and finally produces and solves an MPMT instance. In the interest of succinctness, we omit the data retrieval code. A complete implementation (based on MySQL [4] and the `mysql_protocol` [5] library) can be found online.[2]

The example is based on a facility location problem [10]. We are given a finite set of locations and a finite set of cities. Each city requires a certain number of units of some product. We have to decide where to place facilities in order to satisfy the needs of the cities, while maximizing our earnings.

```
1 open Script ;;
2 open Core.Std ;;
3
4 let n_cities = db_get_n_cities() ;;
5 let locations = db_get_locations() ;;
6 let revenue = db_get_revenue() ;;
7 let capacity = db_get_capacity() ;;
8 let demand = db_get_demand() ;;
9 let n_locations = Array.length locations ;;
```

Lines 1 and 2 are a typical preamble for Inez scripts. The module `Script` contains useful functions for interacting with Inez, while `Core.Std` refers to Jane Street Core, which is a featureful alternative to INRIA's OCaml base library. Inez uses Jane Street Core internally, and we recommend that Inez scripts also use this library. Lines 4 to 8 perform queries to a database instance to obtain data relevant to the problem: (4) an integer `n_cities` with the number of cities we plan to serve; (5) an array `locations` where each position corresponds to the ZIP code of a potential location; (6) a matrix `revenue`, such that for $0 \leq i <$ `n_cities` and $0 \leq j <$ `n_locations`, `revenue`$.(i).(j)$ represents the revenue of selling to city i a unit of product fabricated at location j; (7) an array `capacity`, where for $0 \leq j <$ `n_locations`, `capacity`$.(j)$ is the production capacity for a factory in location j; and (8) an array `demand` that represents the demand, in units of product, from each city. Finally, we define `n_locations` as the size of the array `locations`, *i.e.*, the number of potential locations.

```
10 let build =
11   let f _ = fresh_bool_var () in Array.init n_locations ~f ;;
12
13 let production =
```

[2] `http://www.ccs.neu.edu/home/pete/2015/cav-example.zip`

```
14  let f _ =
15    let f _ =
16      let v = fresh_int_var () in
17      constrain (~logic (v >= 0)); v in
18    Array.init n_locations ~f in
19  Array.init n_cities ~f ;;
```

Each city is identified by an integer $c \in [0 \ldots n\_cities - 1]$. Each location is identified by the corresponding ZIP code in the `locations` array. Line 10 defines an array of size `n_locations`, where each element is an Inez Boolean variable, created by the function `fresh_bool_var`. The library function `Array.init` initializes each element of the array, by calling its `f` argument (a function) with the corresponding index as the argument. (In our case, the argument to `f` is ignored, hence the underscore.) Each Boolean variable corresponds to a location and represents whether a facility is built there or not. Similarly, line 13 defines a two-dimensional matrix of Inez integer variables. The two dimensions correspond to cities and locations: for each possible pair of city c and location l, `production.(c).(l)` represents the planned production (in units of product) of a factory to-be-built in location l destined to city c. The Inez-provided function `constrain` adds a formula to the solver context. In line 17 we constrain each integer variable so that it only takes positive values.

For expressing these constraints, we utilize the `~logic` keyword. `~logic` allows expressing terms and formulas with minimal syntactic overhead. We utilize Camlp4 infrastructure to preprocess applications of `~logic` to ensure that the intended meaning over terms and formulas applies. Specifically, (a) integer literals become Inez integer terms; (b) the literals `true` and `false` become formulas; and (c) operators like `+` and `&&` obtain meaning over terms and formulas (as opposed to their standard meaning over OCaml integers and Booleans, respectively). For instance, given integer variables `x` and `y`, `~logic (x + 1)` is an Inez term, while `~logic (1 <= y && x <= 0)` is an Inez formula. Inez integer terms and formulas are regular OCaml values that can be passed around.

```
20 let cost (_ : Int) = (~free : Int) ;;
21
22 for i = 0 to n_locations - 1 do
23   let i = toi i in
24   constrain (~logic (cost i >= hist_lb i &&
25                      cost i <= hist_ub i))
26 done ;;
```

Now consider the following situation. During an early planning phase, the exact cost of building a facility on a given location may be unknown. However, experience from similar previous developments could provide bounds for these costs. We use an uninterpreted function (UF) to express this. Given an integer representing the ID of a location, the UF `cost` (Line 20) returns an integer that corresponds to the cost of building a facility on that location. The syntax for UFs follows closely the standard syntax for defining OCaml functions. Inez recognizes the declaration of `cost` as its own responsibility because of the keyword `~free` in

the function body. The declaration produces an actual OCaml function `cost` from integer terms to integer terms. (*Integer terms* belong to an OCaml datatype that describes symbolic integer expressions; integer terms differ from OCaml integers.) Functions can also operate over Booleans. (The `Int` annotations could have been omitted, because integer is the default.)

The function `toi` (Line 23) converts an OCaml integer to an integer term. We use `constrain` to bound the return values of `cost` (Line 25). The upper (respectively lower) bound for each location is computed by the OCaml function `hist_ub` (respectively `hist_lb`), which retrieves historical construction data from a database and analyzes the current situation in order to provide estimate bounds for the cost of building. We impose this constraint across all locations by means of a standard OCaml `for` loop (Lines 22-26).

Also, suppose that we have some knowledge about the global building costs for each location and how they compare to one another. That is, given two ZIP codes, we can determine where it is cheaper to build a factory. This knowledge allows us to define an ordering among ZIP codes, and thus assign to each a unique identifier in the range $[0 \ldots n\_locations - 1]$ such that the ZIP code with *id* 0 is the cheapest location and the one with *id* $n\_locations - 1$ is the most expensive. Given such ids, `cost` is monotonically increasing. Monotonicity can be expressed in Inez by means of an axiom, as follows:

```
27 assert_axiom
28   (~forall x (~forall y ([x <= y], cost x <= cost y))) ;;
```

The function `assert_axiom` is used to introduce an axiom. The keyword `~forall` defines two universally quantified variables `x` and `y`. We subsequently provide a list of assumptions (in this case just `x <= y`), followed by a conclusion (`cost x <= cost y`). We provide details on our implementation of axioms in Section 4.2.

We subsequently add constraints to ensure that the units produced by each factory that is built will not exceed its capacity, *i.e.*, that

$$\forall l \in locations. \left[\left(\sum_{c \in cities} production[c][l] \right) \leq build[l] * capacity[l] \right].$$

The Inez encoding is

```
29 for l = 0 to n_locations - 1 do
30  constrain
31    (let cities = List.init n_cities ~f:Fn.id
32     and f c = production.(c).(l) in
33     ~logic
34       (sum cities ~f <= iite build.(l) (toi capacity.(l)) 0))
35 done ;;
```

We notably use the Inez-provided function `sum` (Line 34), to express the sum of Inez terms resulting from the application of the function `f` on each element of the list `cities`. Additionally, we use the function `iite` (Line 34) that encodes an if-then-else condition. The first argument to `iite` is an Inez formula, while

the second and third are Inez integer expressions for each possible case. Our application of `iite` ensures that, if a factory is built on location `l`, then we obtain the capacity from the corresponding array, otherwise the capacity is zero.

Our concrete example additionally enforces that the demands of each city are satisfied, which can be expressed mathematically as:

$$\forall c \in \mathit{cities}. \left[\left(\sum_{l \in \mathit{locations}} \mathit{production}[c][l] \right) = \mathit{demand}[c] \right]$$

In the interest of brevity, we omit the corresponding Inez code.

Finally, we define an objective function, which is to maximize the earnings, *i.e.*, the total revenue minus the cost of building the factories. The corresponding Inez code specifies the optimization criterion by means of the `maximize` function (and re-uses constructs that we have described already):

```
36 maximize
37   (let cities    = List.init n_cities ~f:Fn.id
38    and locations = List.init n_locations ~f:Fn.id in
39    let s1 =
40      ~logic (sum cities ~f:(fun c -> sum locations ~f:(fun l ->
41        revenue.(c).(l) * production.(c).(l))))
42    and s2 =
43      ~logic (sum locations ~f:(fun l ->
44        iite build.(l) (cost (toi l)) 0)) in
45    ~logic (s1 - s2)) ;;
46
47 solve_print_result() ;;
```

Line 47 starts the solving process and prints the result (which can be one of `opt`, `sat`, `unsat`, `unbounded`, or `unknown`) to the standard output. Note that our example builds a single set of constraints, and calls the underlying solver once. In general, Inez provides an incremental *push/pop* interface that allows the user to add and remove constraints, and perform multiple queries. As an example, consider that for the presented problem we had two different optimization criteria: first maximize the earnings and second minimize the number of factories. One could achieve this by *push*-ing the first maximization criterion, solving the problem, registering the maximum value obtained, and finally *pop*-ing the criterion. One could then add a constrain that restricts the first criterion to be equal to the registered value and minimize the second criterion. The full power of OCaml is available to determine future steps by examining intermediate results. Inez thus provides a framework for constraint-based algorithms.

We conclude this section with an overview of the OCaml features that we have utilized to provide the functionality described in this section. Interestingly, (a) *Generalized Algebraic Data Types (GADTs)* [18] allow us to represent terms and formulas in a type-safe way; (b) the extensibility of Inez is reflected on the *module system*, *i.e.*, extending the backend amounts to instantiating a *functor* that given a theory solver (wrapped up as a module) produces a solver for the

resulting logic (*i.e.*, another module); (c) the *toplevel system* allows us to build custom read-evaluate-print loops that interactively interpret OCaml plus our logic fragments; finally, (d) `camlidl` enables relatively seamless interaction with C/C++ code (like SCIP and our implementation of CC).

3 An Algorithmic View of BC(T)

This section provides a set of interconnected algorithms that together implement BC(T). We thus document the architecture empowering the backend of Inez. The algorithms primarily operate upon *nodes* and sets thereof. Each node is described by a set of *integer linear constraints*, *i.e.*, constraints of the form $c_1 \cdot v_1 + \cdots + c_n \cdot v_n \ \{< \mid \leq \mid = \mid \geq \mid >\} \ c$, where c_i are integer constants, v_i are variable symbols, and the right-hand side c is an integer constant. While we provide support for *mixed integer linear constraints* (*i.e.*, integer and real variables side-by-side) through an experimental version of Inez, our discussion focuses on the integer case for simplicity. A node characterizes an open subproblem that needs to be explored. Nodes also carry metadata, like known variable bounds.

In addition to the integer linear constraints, the input to the solver contains uninterpreted function (UF) constraints. We assume that variable abstraction [20] has happened as a preprocessing step, resulting in linear constraints that do not involve UF terms. The definitions symbol that is used in the pseudocode stands for a collection of atomic formulas of the form $v = f(l)$, where v is a variable symbol, f is a UF symbol, and l is a list of arguments of the form $w+k$, where w is a variable symbol and k is an integer constant. Entirely concrete terms are a special case that can be encoded with a single integer variable fixed to zero. UF terms thus involve limited arithmetic, as is common practice [27]. definitions is an immutable global constant.

Our pseudocode uses *sum types* (also known as *tagged unions*) for some of the variables. Sum types have multiple *constructors* that correspond to different cases for the values carried. The constructor of a particular element serves as a tag denoting which case the element belongs in. Furthermore, the magic constant $*$ stands for non-deterministic Boolean choice. $*$ is used in conditionals where heuristics apply. $\langle e \rangle$ denotes that standard operators within e are to be interpreted over syntactic objects, *e.g.*, $\langle v - w \rangle$ is not a concrete integer or real, but a term representing the subtraction of w from v. We follow a generally applicative style, *e.g.*, operations that modify a node (by producing new linear constraints and bounds) produce a new node. Our presentation is top-down.

Our CC solver is implemented by the functions with suffix `_cc` (`check_cc`, `enforce_cc`, `propagate_cc`, and `branch_cc`). These are the functions that need to be replaced (or enhanced) for supporting a theory other than $\emptyset$.

3.1 High-Level Functions

The top-level B&C procedure, `bc`, accepts as its argument a set of integer linear constraints, p. p corresponds to the root node of the B&C search tree. `bc` keeps

```
function bc(p : node) : (Unsat| Optimal(assignment))
    P ← {p}
    α ← None
    while P ≠ ∅ do
        q ← pick(P)
        P ← P \ {q}
        match solve_node(q, obj(α)) with
            case Unsat
                noop()  // do nothing
            case Solved(β)
                α ← Some(β)
            case Branched(Q)
                P ← P ∪ Q
    match α with
        case None
            return Unsat
        case Some(β)
            return Optimal(β)
```

track of a set P of nodes to be examined (initialized with $\{p\}$). α carries a candidate satisfying (integer) assignment. α, belonging in a sum type, is of the form $\mathsf{Some}(\beta)$ if an assignment β is known; α is None otherwise. The loop body in `bc` picks one of the remaining nodes in P and processes it by calling `solve_node`. The implementation of `pick` (not provided) may involve sophisticated heuristics for the choice of next node to be examined. A bias towards the children of the node that was more recently branched upon (*i.e.*, depth-first search) is common.

`solve_node` receives as arguments the node p to be processed, in addition to an upper bound l for the objective values of the solutions of interest. l corresponds to an already-known solution. (We assume that the function `obj` that computes l and our comparisons with l take care of the possibility of no known solution or unbounded solution, by supporting the special constants $+\infty, -\infty$.) `solve_node` performs three processing stages: (a) propagation (Section 3.2); (b) solving a continuous relaxation of the linear constraints; and (c) enforcing constraints against a relaxation-obtained solution (Section 3.3). Enforcing may result in branching (Section 3.4). The aforementioned stages operate on one node at a time, always called p in the respective functions, while their output may be multiple nodes, as a result of branching.

3.2 Propagation

The function `propagate` attempts to reduce the domain of variables. In the process of doing so, it may detect infeasibility (response Unsat); if it succeeds, `propagate` returns a version p' of the original node p modified with new bounds

```
function
solve_node(p : node, l : int) : (Unsat| Solved(assignment)| Branched({node}))
    match propagate(p) with
        case Unsat
            return Unsat
        case Unmodified
            noop()
        case Modified(p′)
            p ← p′
    match solve_relaxation(p) with
        case Unsat
            return Unsat
        case Modified(p′)
            return solve_node(p′, l)
        case Solved(α)
            if obj(α) ≥ l then
                return Unsat
            else
                match enforce(p, α) with
                    case Sat
                        return Solved(α)
                    case Unsat
                        return Unsat
                    case Modified(p′)
                        return solve_node(p′, l)
                    case Branched(P)
                        return Branched(P)
```

(Modified(p')); Unmodified means that no propagation was possible, neither was the function able to detect infeasibility. The implementation we provide combines ILP (`propagate_ilp`) and CC (`propagate_cc`) propagation techniques. We do not explain `propagate_ilp`, which is internal to the ILP solver, and as such orthogonal to BC(T). Either kind of propagation can be skipped. We repeatedly perform propagation, until a fixpoint is reached, or until a heuristic for termination returns true, *e.g.*, after a fixed number of rounds. In practice, SCIP employs various *constraint handlers* that provide propagation procedures of different *priority*. The top-level propagation procedure takes into account priorities to combine the sub-procedures.

`propagate_cc` is described in a declarative way. Our concrete implementation is similar to the CC procedures in SMT, and therefore takes offsets into account [27]. equalities(p) stands for known equalities of the form $v = w + k$, where v and w are integer variables and k is an integer constant. We implement this by defining an auxiliary variable $d_{v,w} = v - w$ for every interesting pair of

```
function propagate(p : node) : (Unsat| Modified(node)|Unmodified)
    m ← false
    while ∗ do
        if ∗ then
            match propagate_ilp(p) with
                case Unsat
                    return Unsat
                case Modified(p′)
                    p ← p′
                    m ← true
                case Unmodified
                    noop()
        if ∗ then
            match propagate_cc(p) with
                case Unsat
                    return Unsat
                case Modified(p′)
                    p ← p′
                    m ← true
                case Unmodified
                    noop()
    return m ? Modified(p) : Unmodified
```

variables v and w. We can subsequently query whether $d_{v,w}$ is fixed. For any equality $v = w + k$ implied by the already known equalities (conjoined with definitions), we try to fix the upper and lower bound of $v - w$ to k (via the functions `set_lb` and `set_ub` that provide an interface to the ILP solver), and report unsatisfiability if this is impossible. The outer forall statement should be read as a declarative specification (*i.e.*, we range over all relevant v, w), not as a suggestion for efficient implementation.

3.3 Enforcing Continuous Relaxations

In `solve_node`, propagation is followed by solving a continuous relaxation. A response Unsat for the relaxation implies that the integer constraints of the node (which are strictly harder than the real constraints of the relaxation) are also unsatisfiable. If `solve_relaxation` returns an assignment α (case Solved(α)), `solve_node` first checks whether α is better than the already known solution (obj(α) $< l$), and does not further process the node if not; integer solutions can be at most as good as the solution to the relaxation. Otherwise, `enforce` is executed. If α is not integer, or if it is theory-inconsistent, `enforce` is responsible for explaining why, *e.g.*, by introducing implied linear constraints violated by α. `enforce` may determine that α satisfies all constraints (response Sat), or that the node (and not just α) is infeasible (response Unsat). In either of these cases,

```
function propagate_cc(p : node) : (Unsat| Modified(node)|Unmodified)
    m ← false
    forall v, w ∈ variables(p) do
        if equalities(p) ∧ definitions ⊨_Z v = w + k for some k ∈ ℤ then
            match set_lb(p, ⟨v − w⟩, k) with
                case Unsat
                    return Unsat
                case Modified(p′)
                    m ← true
                    p ← p′
                case Unmodified
                    noop()
            match set_ub(p, ⟨v − w⟩, k) with
                case Unsat
                    return Unsat
                case Modified(p′)
                    m ← true
                    p ← p′
                case Unmodified
                    noop()
    return m ? Modified(p) : Unmodified
```

`solve_node` has solved p. Enforcing may result in learning new linear constraints or bounds (case Modified(p')), in which case `solve_node` re-processes the node.

`enforce` combines different kinds of enforcement, in much the same way that `propagate` combines different kinds of propagation. The part of enforcement that is related to integrality (`enforce_ilp`) may branch around a real solution, or apply cut generation techniques [15, 12]. ILP cut generation techniques are beyond the scope of this paper. Conversely, the implementation of `enforce_ilp` is not shown. We proceed to describe CC enforcement (`enforce_cc`). First, `enforce_cc` calls `check_cc` to check whether α is theory-consistent. `check_cc` reports that α does not satisfy the UF constraints if there exist calls $v = f(l)$ and $v' = f(l')$ of some function f, such that all arguments in the respective positions of the lists of arguments l and l' have the same value under α, but $\alpha(v) \neq \alpha(v')$. `check_cc` then returns the *conflict* (f, v, v', l, l') to explain what is wrong with α. If no conflict is found, `bc` receives α, and α becomes the new candidate solution. In case `check_cc` returns a conflict, `enforce_cc` ensures that propagation has happened by calling `propagate_cc` again. The latter function may have been skipped during the propagation stage. `enforce_cc` only needs to act further if propagation can neither detect unsatisfiability, nor produce new information. In this case, `enforce_cc` proceeds by branching.

Note that CC enforcement happens after the corresponding method for the ILP constraints. CC enforcement thus only ever deals with integer assignments,

```
function
enforce(p : node, α : assignment) : (Sat|Unsat| Modified(node)| Branched({node}))
    match enforce_ilp(p, α) with
        case Sat
            return enforce_cc(p, α)
        case Unsat
            return Unsat
        case Modified(p′)
            return Modified(p′)
        case Branched(P)
            return Branched(P)
```

```
function enforce_cc(p : node, α : assignment) :
(Sat|Unsat| Modified(node)| Branched({node}))
    match check_cc(p, α) with
        case Conflict(c)
            match propagate_cc(p) with
                case Unsat
                    return Unsat
                case Modified(p′)
                    return Modified(p′)
                case Unmodified
                    return Branched(branch_cc(p, c))
        case Sat
            return Sat
```

which yields cleaner implementation. Additionally, this design prioritizes ILP-related over theory-related operations, thus emphasizing ILP-heavy problems.

3.4 Branching

Branching is what our CC implementation performs when all else fails. Concretely, the following invariant holds when we get to `branch_cc`. There exists an integer solution for the non-theory constraints of p (given that integrality enforcement has succeeded), but the integer bounds that hold for p do not allow any information to be propagated, neither can we deduce unsatisfiability of p.

When we call `branch_cc` from `enforce_cc`, we have access to a conflict (f, v, v', l, l'). Note that there must be some position $i \in [0, \texttt{arity}(f) - 1]$ such that the equality $l[i] = l'[i]$ is not implied by the bounds visible to `propagate_cc`. Otherwise, all arguments would have been equal, and `propagate_cc` would have produced the equality $v = v'$, which is violated. It is always possible to branch on whether $l[i] < l'[i]$, $l[i] = l'[i]$, or $l[i] > l'[i]$. If, according to the bounds on

```
function check_cc(p : node, α : assignment) : (Sat| Conflict(conflict))
    m ← {}  // m is a map
    foreach ⟨v = f(l)⟩ ∈ definitions do
        c ← [α(w) + k|⟨w + k⟩ in l]  // list comprehension over l
        if (f, c) ∈ keys(m) then
            (v′, l′) ← m[(f, c)]
            if α(v) ≠ α(v′) then
                return Conflict(f, v, v′, l, l′)
        else
            m[(f, c)] ← (v, l)
    return Sat
```

$v - v'$, $v = v'$ is a possibility, then we may instead choose to branch on whether $v < v'$, $v = v'$, or $v > v'$. Our branching involves very little guesswork. A conflict (f, v, v', l, l') provides a witness for the gap between the assignment under examination α (which is not feasible with respect to UF) and the more limited information that is available as bounds in p (which do not entail infeasibility). In order to steer the ILP solver away from the problematic assignment α (and other assignments similar to it), we have to examine the aforementioned gap. We do so by branching driven by the conflict.

The branching strategy we outlined is in alignment with the Nelson-Oppen (NO) scheme for combining decision procedures [25, 20]. We branch on pairs of variables that are shared between ILP and UF, *i.e.*, make progress towards an *arrangement* of the shared variables. Such branching will eventually produce subproblems for which CC has all the information on the shared variables that it needs to determine (in)feasibility of the UF constraints (in definitions); similarly, for the UF-feasible subproblems, the ILP solver (with no more input possible from CC) has all the information it needs to apply complete techniques and determine feasibility. We thus guarantee termination of the combination.

4 Extensions

4.1 Propositional Structure

We have so far not discussed integer linear constraints that appear under arbitrary propositional structure. Inez provides such support by utilizing *indicator constraints*. Such constraints have the form $l \Rightarrow \Sigma_{0 \leq i < n}[c_i \cdot x_i] \leq c$, where l is a possibly negated Boolean variable, c_i and c are constants, and v_i are variables. Indicator constraints can establish equivalence between a Boolean variable b and an inequality $\Sigma_{0 \leq i < n}[c_i \cdot x_i] \leq c$ via the constraints $b \Rightarrow \Sigma_{0 \leq i < n}[c_i \cdot x_i] \leq c$ and $\neg b \Rightarrow -\Sigma_{0 \leq i < n}[c_i \cdot x_i] \leq -c - 1$. Once we have Boolean variables like b, encoding propositional structure can be done via clauses (which are a special case of integer linear inequalities) in a Tseitin-like fashion.

```
function branch_cc(p : node, (f, v, v′, l, l′) : conflict) : {node}
    if ∗ ∧ lb(p, ⟨v − v′⟩) ≤ 0 ∧ ub(p, ⟨v − v′⟩) ≥ 0 then
        P ← {⟨p ∧ v = v′⟩}
        if lb(p, ⟨v − v′⟩) < 0 then
            P ← P ∪ {⟨p ∧ v < v′⟩}
        if ub(p, ⟨v − v′⟩) > 0 then
            P ← P ∪ {⟨p ∧ v > v′⟩}
        return P
    for i ∈ [0, arity(f) − 1] do
        if α(l[i]) = α(l′[i]) then
            if lb(p, ⟨l[i] − l′[i]⟩) < 0 ∨ ub(p, ⟨l[i] − l′[i]⟩) > 0 then
                P ← {⟨p ∧ l[i] = l′[i]⟩}
                if lb(p, ⟨l[i] − l′[i]⟩) < 0 then
                    P ← P ∪ {⟨p ∧ l[i] < l′[i]⟩}
                if ub(p, ⟨l[i] − l′[i]⟩) > 0 then
                    P ← P ∪ {⟨p ∧ l[i] > l′[i]⟩}
                return P
    assert(false) // unreachable
```

Indicator constraints can be encoded in terms of integer linear constraints [21], based on a technique that is known as Big-M. SCIP deals with indicator constraints via a specialized *constraint handler* (rather than via Big-M). This handler implements indicator constraints through propagation, enforcing, and branching functions that fit in BC(T) just like their CC counterparts (Section 3).

4.2 Local Theory Extensions

We demonstrate how to support user-provided axioms within BC(T) and lnez. Such axioms constrain newly defined function symbols (beyond the ones in the signature $\Sigma_{\mathcal{Z}}$ of Linear Integer Arithmetic, *e.g.*, +). We thus *extend* QFLIA by axiomatizing new functions. Throughout this section, we assume a first-order signature Σ, comprised of the axiomatized function symbols.

An example of the kinds of axioms we support was given via the axiom in Line 28 of our introductory example. Our axiom is only meaningful as an *extension* of (Integer Linear) Arithmetic. The intended meaning (monotonicity of `cost`) is only achieved because $\leq$ is already constrained by Arithmetic. More generally, we support axioms that are universally quantified disjunctions of inequalities that may contain function symbols. Our focus on clauses is not a restriction; collections of clausal axioms can be used to encode axioms with more complex structure.

Our implementation of axioms in lnez builds upon results on *local theory extensions* [30] that allow us to replace axioms with a finite set of instances thereof (computed based on the set of terms that appear in the formula). In

our case, the instantiation procedure produces clauses, where the literals involve arithmetic and the Σ-function symbols.

While in principle we can simply encode the axiom instances of interest as part of the input formula (Section 4.1), our implementation applies a specialized procedure that retains the clausal structure. The literals are inequalities, *e.g.*, for our monotonicity example we have inequalities of the form $x \leq y$ and $\texttt{cost}(x) \leq \texttt{cost}(y)$ over x and y that appear in the input as arguments to `cost`. By introducing fresh variables, we simplify these literals by rewriting them to the form $v \leq c$, where v is a variable and c is a constant. We then employ a SCIP handler for constraints of the form $\bigvee_i v_i \leq c_i$, that notably employs SAT-like techniques for clauses [6].

4.3 Databases

Inez provides an extension aimed at database analysis [22]. The workhorse of this extension is what we call *table membership* constraints, which have the form

$$(x_1{+}c_1, \ldots, x_k{+}c_k) \in \{(y_{1,1}{+}d_{1,1}, \ldots, y_{1,k}{+}d_{1,k}), \ldots, (y_{l,1}{+}d_{l,1}, \ldots, y_{l,k}{+}d_{l,k})\},$$

where x_i, $y_{i,j}$ are variables and c_i, $d_{i,j}$ are (integer) constants. On top of table membership, Inez provides higher-level database-inspired modeling constructs.

Table membership fits in BC(T) just like CC. Functions `propagate_db` and `enforce_db` replace (or enhance) the corresponding CC functions, while everything else remains unchanged. Design decisions in `enforce_db` resemble the ones in `enforce_cc`, *e.g.*, branching (driven by the data) happens only as a last resort.

5 Related Work

Frontend: Existing projects that enhance programming languages with constraints [19, 31, 7] differ from Inez both with respect to the language constructs that they provide and the underlying constraint technology.

Backend: Inez seeks to combine the strengths of MP solvers [2, 3, 8] and solvers for first-order theories [24, 29, 25], *e.g.*, as implemented within Lazy SMT [28]. Previous work on similar combinations has focused on improving the arithmetic capabilities [13, 16] of SMT solvers by integrating MP engines [14, 23, 9], and on extending SMT for optimization [26, 11]. MPMT differs by having as its core an MP solver, as opposed to a SAT solver.

6 Conclusions

We provided an overview of the techniques that empower the Inez constraint solver. Inez is an open-source, freely available system that instantiates the BC(T) architecture for Mathematical Programming Modulo Theories. We described the concrete algorithms used to in Inez to efficiently implement BC(T). Inez is an extension of OCaml that allows users to write programs that orchestrate arbitrary interleaving between general computation and MPMT constraint solving.

References

1. Camlp4. See `https://github.com/ocaml/camlp4/wiki`.
2. CPLEX. See `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.
3. Gurobi. See `http://www.gurobi.com`.
4. MySQL. See `https://www.mysql.com/`.
5. mysql_protocol. See `https://github.com/slegrand45/mysql_protocol`.
6. SCIP Constraint Handler for Bound Disjunction Constraints. See `http://scip.zib.de/doc-3.1.1/html/cons__bounddisjunction_8h.php`.
7. Z3py. See `http://rise4fun.com/z3py/tutorial`.
8. T. Achterberg. SCIP: Solving Constraint Integer Programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.
9. F. Besson. On using an inexact floating-point LP solver for deciding linear arithmetic in an SMT solver. In *SMT*, 2010.
10. E. Castillo, A. Conejo, P. Pedregal, R. Garca, and N. Alguacil. *Building and Solving Mathematical Programming Models in Engineering and Science*. Wiley, 2002.
11. A. Cimatti, A. Franzen, A. Griggio, R. Sebastiani, and C. Stenico. Satisfiability Modulo the Theory of Costs: Foundations and Applications. In *TACAS*, 2010.
12. G. Cornuejols. Valid inequalities for mixed integer linear programs. *Mathematical Programming*, 112(1):3–44, 2008.
13. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, 2006.
14. G. Faure, R. Nieuwenhuis, A. Oliveras, and E. Rodriguez-Carbonell. SAT Modulo the Theory of Linear Arithmetic: Exact, Inexact and Commercial Solvers. In *SAT*, 2008.
15. R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the AMS*, 64:275–278, 1958.
16. A. Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. *JSAT*, 8:1–27, 2012.
17. C. Hang, P. Manolios, and V. Papavasileiou. Synthesizing Cyber-Physical Architectural Models with Real-Time Constraints. In *CAV*, 2011.
18. S. P. Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple Unification-Based Type Inference for GADTs. In *ICFP*, 2006.
19. A. S. Koksal, V. Kuncak, and P. Suter. Constraints as Control. In *POPL*, 2012.
20. Z. Manna and C. Zarba. Combining Decision Procedures. In *10th Anniversary Colloquium of UNU/IIST*, 2002.
21. P. Manolios and V. Papavasileiou. ILP Modulo Theories. In *CAV*, 2013.
22. P. Manolios, V. Papavasileiou, and M. Riedewald. ILP Modulo Data. In *FMCAD*, 2014.
23. D. Monniaux. On using floating-point computations to help an exact linear arithmetic decision procedure. In *CAV*, 2009.
24. G. Nelson and D. Oppen. Fast Decision Algorithms Based on Union and Find. 1977.
25. G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *TOPLAS*, 1:245–257, 1979.
26. R. Nieuwenhuis and A. Oliveras. On SAT Modulo Theories and Optimization Problems. In *SAT*, 2006.
27. R. Nieuwenhuis and A. Oliveras. Fast Congruence Closure and Extensions. *Information and Computation*, 205:557–580, 2007.

28. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *JACM*, 53(6):937–977, 2006.
29. R. Shostak. A practical decision procedure for arithmetic with function symbols. *JACM*, 26(2):351–360, 1979.
30. V. Sofronie-Stokkermans. Hierarchic Reasoning in Local Theory Extensions. In *CADE*, 2005.
31. E. Torlak and R. Bodik. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *PLDI*, 2014.