OMT check to CP/MILP problems

A comparison of OptiMathSat performances against CP and MILP solvers

Author:
Trentin Patrick

Summary

In the recent years some SMT solvers have been extended with the capability of processing optimization problems, thus these tools are now being referred as OMT solvers. The aim of this document is to compare the performances of OptiMathSat, an OMT solver, in respect to some CP/MILP solvers against benchmarks residing in the rational domain space. To do this a couple of software tools able to encode the problems written in the language of one solver into the other will be implemented.
Contents

1 Motivation and Goals 1

2 Developed Tools 1
   2.1 FZtoSL2: FlatZinc to SmtLibv2 2
       2.1.1 Predicate declarations 2
       2.1.2 Parameter and Variable declarations 2
       2.1.3 Constraints 3
       2.1.4 Solve Goal 11
       2.1.5 Annotations 11
       2.1.6 Usage of FZtoSL2 11
   2.2 SL2toMZ: SmtLibv2 to MiniZinc 12
       2.2.1 SmtLibv2 Scripts 12
       2.2.2 Lexicon 13
       2.2.3 Term and Formulas 13
       2.2.4 Logics and Theories 13
       2.2.5 ITE and LET statements 14
       2.2.6 Solve Goal 14
       2.2.7 Usage of SL2toMZ 15

3 Benchmarks 15
   3.1 Benchmarks: QF_LRA 15
   3.2 Experiments 17
       3.2.1 Gecode 17
       3.2.2 Eclipse 22

4 Conclusion and Future Work 24
1 Motivation and Goals

In the field of Satisfiability Modulo Theory (SMT), the goal is to check whether a given (closed) logical formula $\varphi$ is satisfiable in the context of a background theory $T$, i.e. there exists an assignment for $T$ that makes $\varphi$ true.

In the last years, there has been done some work in order to extend the SMT solvers to deal with optimization problems with $\mathcal{L}\mathcal{A}(\mathbb{Q})$ cost functions. We speak in such cases of Optimization Modulo Theory (OMT) problems, in which the goal is formulated in terms of a pair $<\varphi, \text{cost}>$ such that $\varphi$ is a SMT($\mathcal{L}\mathcal{A}(\mathbb{Q}) \cup T$) formula and cost is a $\mathcal{L}\mathcal{A}(\mathbb{Q})$ variable occurring in $\varphi$, representing the cost to be minimized.

As a consequence SMT and OMT solvers are now widening their horizon to a broader set of problems and benchmarks than those available before. One of such examples is the domain of Constraint Problems (CP). A CP problem is usually formulated as a conjunction of constraints restricting the combination of values that a set of decision variables may take simultaneously. Various approaches can be taken to solve a CP: Mixed Integer Linear Programming (MILP), Lazy Clause Generation (Lazy fd), Genetic Algorithms, SAT, etc.

In this document, we will try to assess the performances of a SMT/OMT solvers in the domain of Constraint Problems in respect to various competitors, both when the requirement is to find a satisfactory solution or to optimize a decision variable in respect to a cost function.

To do this comparison we will first examine how a problem encoded in the SmtLibv2 standard, used by MathSat and OptiMathSat, can be translated into the MiniZinc language and vice-versa. This is done through a couple of newly developed tools introduced in section 2.

After this, we will try to benchmark the performances of different solvers in section 3, taking problems both from the libraries of MiniZinc and SmtLibv2. Note that the focus of this research is on optimization problems with a rational cost function, although other fields may be evaluated too.

2 Developed Tools

In order to allow the comparison of the MathSat and OptiMathSat tools performances to the FlatZinc solvers, it has been necessary to develop a couple of compilers able to translate a model from the SmtLibv2 language to MiniZinc and vice-versa. Let us briefly review the characteristics of these two languages:

- **MiniZinc** is a Constraint Satisfaction Programming (CSP) and Constraint Optimization Programming (COP) modelling language that allows for modelling problems for a range of different solvers using Finite Domain (FD) and Linear Programming (LP) techniques. Some examples of such solvers include: Gecode, ECLiPSe, ILOG Solver, Minion, Choco.

  MiniZinc is a first order language that provides three scalar types (booleans, integers and floats) and two compound types (sets and arrays of fixed size), which can be instantiated as parameters or decision variables in the case of integers, floats, booleans and set of integers.

  Annotations can be used to add non-declarative information, such as search strategies, and solver specific information, such as variable representations, to be layered on top of declarative models.

  From a MiniZinc it can be generated, through a flattening phase followed by some post-flattening operations, an equivalent FlatZinc model. The FlatZinc language is a low-level solver-input language that allows solvers to support MiniZinc with a minimum of effort.

- **SmtLibv2** standard is the result of an international effort to develop a common languages and interfaces for SMT solvers. The version 2.0 of the SmtLibv2 standard is indeed a major upgrade that simplified and extended the language, including a new command language to interface with a SMT solver. This language contains formalizations of arithmetic, arrays, bit vectors, algebraic data types, equality with uninterpreted functions and various combinations of these theories.

  As it can be seen in figure 1, the chosen translation scheme takes advantage of the mzn2fzn tool provided with the MiniZinc distribution. This choice has been made for a couple of reasons:

  - it allows the mzn2fzn tool to apply optimizations during the flattening of the MiniZinc model toward the representation that best fits its usage;
  - it simplifies the SL2toMZ tool logic, thanks to the similarities inbetween the syntax of the two languages;
At the time being, the FZtoSL2 and SL2toMZ tools are able to translate only a restriction of the domain language, because some features have been left out since they were unnecessary to the goal to be reached or did not have any corresponding syntax construct in the target language. These limitations will be better understood in the next two following sections, devoted to a deep overview of the implemented tools.

2.1 FZtoSL2: FlatZinc to SmtLibv2

This section reviews the subset of the FlatZinc language that the FZtoSL2 tool covers and translates into SmtLibv2. The basic syntax of a model is the following:

```
flatzinc model ::= [pred decl*] [param decl*] [var decl*] [constraint*] solve goal
```

2.1.1 Predicate declarations

This kind of statement allows non standard predicates to be used within a model, its syntax is:

```
predicate prename (type: argname, ...);
```

The FZtoSL2 application assumes that the MiniZinc models are transformed into FlatZinc models using its standard mzn2fzn tool. Therefore, personalized predicates specified in the former language are already substituted in the latter one without the need of further declarations and this kind of statements do not need to be used.

2.1.2 Parameter and Variable declarations

Parameters are fixed quantities explicitly specified in the model, while variables are quantities to be decided by the solver. Currently the FZtoSL2 tool does not make any distinction among parameters and variables, therefore a possible improvement could be the substitution of each parameter name occurrence with its fixed value. While this change is cheap and easy to implement, it should be noted that OptiMathSat solver is able to do constant propagation and to learn clauses by itself.
A comprehensive list of FlatZinc basic types and their mapping\(^1\) into SmtLibv2 is shown in table 1. As pointed out in the table, sets are not yet supported although the tool can be extended in this direction.

A handful set of parameters and variables translation examples are provided in the following list.

\[
\text{bool: } \text{a.bool} = \text{false};
\]
\[
\text{= (declare - fun a.bool() Bool)}
\]
\[
\text{= (assert (= a.bool false))}
\]

\[
\text{var float: } \text{b.float} = 0.9;
\]
\[
\text{= (declare - fun b.float() Real)}
\]
\[
\text{= (assert (= b.float 0.9))}
\]

\[
\text{var } \{1, 2, 3\}: \text{r.dom;}
\]
\[
\text{= (declare - fun r.dom() Int)}
\]
\[
\text{= (assert (or (= r.dom 3) (or (= r.dom 2) (= r.dom 1))))}
\]

\[
\text{var } -2.0..1.5: \text{f.dom;}
\]
\[
\text{= (declare - fun f.dom() Real)}
\]
\[
\text{= (assert (and (<= (\text{-} 2) f.dom) (<= f.dom 1.5)))}
\]

\[
\text{array } [1..3] \text{ of } \text{var bool: } \text{ar.bool} = [\text{true}, \text{false}, \text{true}];
\]
\[
\text{= ; ; Declaring array } [1..3] \text{ of type } \text{Bool}, \text{named } \text{ar.bool.}
\]
\[
\text{= (declare - fun ar.bool.ARRAY\_1() Bool)}
\]
\[
\text{= (declare - fun ar.bool.ARRAY\_2() Bool)}
\]
\[
\text{= (declare - fun ar.bool.ARRAY\_3() Bool)}
\]
\[
\text{= (assert (= ar.bool.ARRAY\_1 true))}
\]
\[
\text{= (assert (= ar.bool.ARRAY\_2 false))}
\]
\[
\text{= (assert (= ar.bool.ARRAY\_3 true))}
\]

Note that each declaration in FlatZinc results in a sequence of one or more statements in SmtLibv2: variable declaration, domain restriction conditions and value assignment. Due to the way in which FlatZinc treats arrays, these constructs are better mapped into independent variables rather than in the native array construct of SmtLibv2. While on one hand this requires a larger number of identifiers, the model is still readable and the research space itself does not change at all. Every new variable introduced by FZtoSL2 as a place holder for an array location is labeled with the string “__ARRAY__“, followed by its index.

2.1.3 Constraints

FlatZinc has a rich set of constraints with which is possible to shape a model and, excluding those involving sets, allmost all of them have a direct mapping into SmtLibv2. Each of the following subsections will provide an example of FlatZinc constraint and its own version in SmtLibv2. All decision variables in the examples are assumed to be declared, while all arrays will assumed to be of length 3.

Array Constraints

\[(\forall i \in 1..n : as[i]) \iff r, \text{ where } n \text{ is the length of } as\]

\(^1\)Note that FZtoSL2 parses int values with \text{atoll()} into int64_t containers, while it uses \text{atof()} to parse float values into double containers.
\[
\begin{align*}
\text{array\_bool\_and} & (\text{array \{int\} of \text{var bool}: \text{as}, \text{var bool}: r) \\
& = \quad \text{(assert (and (=> (and (and as\_bool\_ARRAY\_1 b as\_bool\_ARRAY\_2) as\_bool\_ARRAY\_3) r, bool)) (=> r, bool (and (and as\_bool\_ARRAY\_1 as\_bool\_ARRAY\_2) as\_bool\_ARRAY\_3))))} \\

b \in 1..n \land as[b] = c, \text{ where } n \text{ is the length of } as \\
\text{array\_bool\_element} & (\text{var int: b, array \{int\} of bool: as, var bool: c) \\
& = \quad \text{(assert (and (and (=> (b\_int \_1) (= as\_bool\_ARRAY\_1 b\_bool)) (=> (b\_int 2) (= as\_bool\_ARRAY\_2 c\_bool))) (=> (b\_int 3) (= as\_bool\_ARRAY\_3 b\_bool))))} \\

(\exists i \in 1..n : as[i]) \iff r, \text{ where } n \text{ is the length of } as \\
\text{array\_bool\_or} & (\text{array \{int\} of bool: as, var bool: r) \\
& = \quad \text{(assert (and (=> (or (or as\_bool\_ARRAY\_1 b as\_bool\_ARRAY\_2) as\_bool\_ARRAY\_3) r, bool)) (=> r, bool (or (or as\_bool\_ARRAY\_1 as\_bool\_ARRAY\_2) as\_bool\_ARRAY\_3))))} \\

((\sum_{i=1}^n as[i] \mod 2) = 1, \text{ where } n \text{ is the length of } as \\
\text{array\_bool\_xor} & (\text{array \{int\} of bool: as) \\
& = \quad \text{(assert (xor (xor as\_bool\_ARRAY\_1 as\_bool\_ARRAY\_2 as\_bool\_ARRAY\_3)))} \\

b \in 1..n \land as[b] = c, \text{ where } n \text{ is the length of } as \\
\text{array\_float\_element} & (\text{var int: b, array \{int\} of float: as, var float: c) \\
& = \quad \text{(assert (and (and (=> (b\_int \_1) (= as\_float\_ARRAY\_1 b\_float)) (=> (b\_int 2) (= as\_float\_ARRAY\_2 c\_float))) (=> (b\_int 3) (= as\_float\_ARRAY\_3 b\_float))))} \\

b \in 1..n \land as[b] = c, \text{ where } n \text{ is the length of } as \\
\text{array\_int\_element} & (\text{var int: b, array \{int\} of int: as, var int: c) \\
& = \quad \text{(assert (and (and (=> (b\_int \_1) (= as\_int\_ARRAY\_1 b\_int)) (=> (b\_int 2) (= as\_int\_ARRAY\_2 c\_int))) (=> (b\_int 3) (= as\_int\_ARRAY\_3 b\_int))))} \\

b \in 1..n \land as[b] = c, \text{ where } n \text{ is the length of } as \\
\text{array\_set\_element} & (\text{var int: b, array \{int\} of set of int: as, set of int: c) \\
& = \quad \text{unsupported} \\

b \in 1..n \land as[b] = c, \text{ where } n \text{ is the length of } as \\
\text{array\_var\_bool\_element} (\text{var int: b, array \{int\} of var bool: as, var bool: c) \\
& = \quad \text{(assert (and (and (=> (b\_int \_1) (= as\_bool\_ARRAY\_1 b\_bool)) (=> (b\_int 2) (= as\_bool\_ARRAY\_2 c\_bool))) (=> (b\_int 3) (= as\_bool\_ARRAY\_3 b\_bool))))} \\

b \in 1..n \land as[b] = c, \text{ where } n \text{ is the length of } as \\
\text{array\_var\_float\_element} (\text{var int: b, array \{int\} of var float: as, var float: c) \\
& = \quad \text{(assert (and (and (=> (b\_int \_1) (= as\_float\_ARRAY\_1 b\_float)) (=> (b\_int 2) (= as\_float\_ARRAY\_2 c\_float))) (=> (b\_int 3) (= as\_float\_ARRAY\_3 b\_float))))} \\

b \in 1..n \land as[b] = c, \text{ where } n \text{ is the length of } as \\
\text{array\_var\_int\_element} (\text{var int: b, array \{int\} of var int: as, var int: c) \\
& = \quad \text{(assert (and (and (=> (b\_int \_1) (= as\_int\_ARRAY\_1 b\_int)) (=> (b\_int 2) (= as\_int\_ARRAY\_2 c\_int))) (=> (b\_int 3) (= as\_int\_ARRAY\_3 b\_int))))} \\

b \in 1..n \land as[b] = c, \text{ where } n \text{ is the length of } as \\
\text{array\_var\_set\_element} (\text{var int: b, array \{int\} of var set of int: as, var set of int: c) \\
& = \quad \text{unsupported} 
\end{align*}
\]
Boolean Constraints

\[(a \iff b = 1) \land (\neg a \iff b = 0)\]

.constraint bool2int(var bool a, var int b):
  = (assert (and (and (=> a (= b 1)) (=> (= b 1) a)) (and (=> (not a) (= b 0)) (=> (= b 0) (not a)))))

\[(a \land b) \iff r\]

.bool_and(var bool a, var bool b, var bool r) = (assert (and (=> (and a b) r) (=> r (and a b))))

\[\exists i \in 1..n_{as} : as[i] \land \exists i \in 1..n_{bs} : \neg bs[i], \text{ where n is the length of as}\]

.bool_clause(array [int] of var bool as, array [int] of var bool bs) = (assert (or (or (as_ARRAY_1 as_ARRAY_2) as_ARRAY_3) (or (not bs_ARRAY_1) (not bs_ARRAY_2) (not bs_ARRAY_3))))

\[a = b\]

.bool_eq(var bool a, var bool b) = (assert (= a b))

\[a = b \iff r\]

.bool_eq_reif(var bool a, var bool b, var bool r) = (assert (and (=> (= a b) r) (=> r (= a b))))

\[\neg a \lor b\]

.bool_le(var bool a, var bool b) = (assert (or (not a) b))

\[\neg a \lor b \iff r\]

.bool_le_reif(var bool a, var bool b, var bool r) = (assert (and (=> (or (not a) b) r) (=> r (or (not a) b))))

\[\sum_{i=1}^{n} as[i].bs[i] = c, \text{ where n is the common length of as and bs}\]

.bool_lin_eq(array [int] of int as, array [int] of var bool bs, var int c) = (assert (= (+ (+ (ite bs_ARRAY_1 as_ARRAY_1 0) (ite bs_ARRAY_2 as_ARRAY_2 0) (ite bs_ARRAY_3 as_ARRAY_3 0)) c))

\[\sum_{i=1}^{n} as[i].bs[i] \leq c, \text{ where n is the common length of as and bs}\]

.bool_lin_le(array [int] of int as, array [int] of var bool bs, int c) = (assert (<= (+ (+ (ite bs_ARRAY_1 as_ARRAY_1 0) (ite bs_ARRAY_2 as_ARRAY_2 0) (ite bs_ARRAY_3 as_ARRAY_3 0)) s))

\[\neg a \land b\]

.bool_lt(var bool a, var bool b) = (assert (and (not a) b))

\[\neg a \land b \iff r\]

.bool_lt_reif(var bool a, var bool b, var bool r) = (assert (and (=> (and (not a) b) r) (=> r (and (not a) b))))

\[\neg a = b\]

.bool_not(var bool a, var bool b) = (assert (= (not a) b))

\[\neg a = b \iff r\]

.bool_or(var bool a, var bool b, var bool r) = (assert (and (=> (or a b) r) (=> r (or a b))))

\[a \neq b\]

.bool_xor(var bool a, var bool b, var bool r) = (assert (and (=> (xor a b) r) (=> r (xor a b))))
Int Constraints

\[ a = b \]
\[
\text{int}_{\text{abs}}(\text{var int: a, var int: b}) = \ (\text{assert (}= (\text{abs a}) b))
\]
\[ a/b = c \text{ rounding towards zero.} \]
\[
\text{int}_{\text{div}}(\text{var int: a, var int: b, var int: c}) = \ (\text{assert (= (/ a b) c)})
\]
\[ a = b \]
\[
\text{int}_{\text{eq}}(\text{var int: a, var int: b}) = \ (\text{assert (= a b)})
\]
\[ (a = b) \iff r \]
\[
\text{int}_{\text{eq}_{\text{reif}}}(\text{var int: a, var int: b, var bool: r}) = \ (\text{assert (and (= a b) r) (= r (= a b))))}
\]
\[ a \leq b \]
\[
\text{int}_{\text{le}}(\text{var int: a, var int: b}) = \ (\text{assert (<= a b)})
\]
\[ (a \leq b) \iff r \]
\[
\text{int}_{\text{le}_{\text{reif}}}(\text{var int: a, var int: b, var bool: r}) = \ (\text{assert (and (<= a b) r) (<= r (<= a b)))}
\]
\[ \sum_{i=1}^{n} a[i].bs[i] = c, \text{ where } n \text{ is the common length of } as \text{ and } bs \]
\[
\text{int}_{\text{lin}_{\text{eq}}}(\text{array [int] of int: as, array [int] of var int: bs, int: c}) = \ (\text{assert (= (+ (+ (* bs_{\text{ARRAY}_\_1} as_{\text{ARRAY}_\_1}) (* bs_{\text{ARRAY}_\_2} as_{\text{ARRAY}_\_2})) (* bs_{\text{ARRAY}_\_3} as_{\text{ARRAY}_\_3}) c))}
\]
\[ (\sum_{i=1}^{n} a[i].bs[i] = c) \iff r, \text{ where } n \text{ is the common length of } as \text{ and } bs \]
\[
\text{int}_{\text{lin}_{\text{eq}_{\text{reif}}}}(\text{array [int] of int: as, array [int] of var int: bs, int: c, var bool: r}) = \ (\text{assert (and (=) (= a b) r) (= r (= a b)))}
\]
\[ \sum_{i=1}^{n} a[i].bs[i] \leq c, \text{ where } n \text{ is the common length of } as \text{ and } bs \]
\[
\text{int}_{\text{lin}_{\text{le}}}(\text{array [int] of int: as, array [int] of var int: bs, int: c}) = \ (\text{assert (<= (+ (+ (* bs_{\text{ARRAY}_\_1} as_{\text{ARRAY}_\_1}) (* bs_{\text{ARRAY}_\_2} as_{\text{ARRAY}_\_2})) (* bs_{\text{ARRAY}_\_3} as_{\text{ARRAY}_\_3}) c))}
\]
\[ (\sum_{i=1}^{n} a[i].bs[i] \leq c) \iff r, \text{ where } n \text{ is the common length of } as \text{ and } bs \]
\[
\text{int}_{\text{lin}_{\text{le}_{\text{reif}}}}(\text{array [int] of int: as, array [int] of var int: bs, int: c, var bool: r}) = \ (\text{assert (and (<=) (= (+ (+ (* bs_{\text{ARRAY}_\_1} as_{\text{ARRAY}_\_1}) (* bs_{\text{ARRAY}_\_2} as_{\text{ARRAY}_\_2})) (* bs_{\text{ARRAY}_\_3} as_{\text{ARRAY}_\_3}) c))}
\]
\[ \sum_{i=1}^{n} a[i].bs[i] \neq c, \text{ where } n \text{ is the common length of } as \text{ and } bs \]
\[
\text{int}_{\text{lin}_{\text{ne}}}(\text{array [int] of int: as, array [int] of var int: bs, int: c}) = \ (\text{assert (distinct (+ (+ (* bs_{\text{ARRAY}_\_1} as_{\text{ARRAY}_\_1}) (* bs_{\text{ARRAY}_\_2} as_{\text{ARRAY}_\_2})) (* bs_{\text{ARRAY}_\_3} as_{\text{ARRAY}_\_3}) c))}
\]
\[ (\sum_{i=1}^{n} a[i].bs[i] \neq c) \iff r, \text{ where } n \text{ is the common length of } as \text{ and } bs \]
\[
\text{int}_{\text{lin}_{\text{ne}_{\text{reif}}}}(\text{array [int] of int: as, array [int] of var int: bs, int: c, var bool: r}) = \ (\text{assert (and (distinct (+ (+ (* bs_{\text{ARRAY}_\_1} as_{\text{ARRAY}_\_1}) (* bs_{\text{ARRAY}_\_2} as_{\text{ARRAY}_\_2})) (* bs_{\text{ARRAY}_\_3} as_{\text{ARRAY}_\_3}) c))}
\]
\[ a < b \]
\[
\text{int}_{\text{lt}}(\text{var int: a, var int: b}) = \ (\text{assert (< a b)})
\]
\[(a < b) \iff r\]

\[
\text{int lt reif}(\text{var int: } a, \text{var int: } b, \text{var bool: } r) = \quad (\text{assert (and (=> (< a b) r) (=> r (< a b))))}
\]

\[\max(a, b) = c\]

\[
\text{int max}(\text{var int: } a, \text{var int: } b, \text{var int: } c) = \quad (\text{assert (ite (<= a b) (= b c) (= a c)})
\]

\[\min(a, b) = c\]

\[
\text{int min}(\text{var int: } a, \text{var int: } b, \text{var int: } c) = \quad (\text{assert (ite (<= a b) (= a c) (= b c)})
\]

\[a - x.b = c, \text{ where } x = a/b \text{ rounding towards zero.}\]

\[
\text{int mod}(\text{var int: } a, \text{var int: } b, \text{var int: } c) = \quad (\text{assert (= (mod a b) c)})
\]

\[a \neq b\]

\[
\text{int ne}(\text{var int: } a, \text{var int: } b) = \quad (\text{assert (distinct a b)})
\]

\[(a \neq b) \iff r\]

\[
\text{int ne reif}(\text{var int: } a, \text{var int: } b, \text{var bool: } r) = \quad (\text{assert (and (=> (distinct a b) r) (=> r (distinct a b))))}
\]

\[a + b = c\]

\[
\text{int plus}(\text{var int: } a, \text{var int: } b, \text{var int: } c) = \quad (\text{assert (= (+ a b) c)})
\]

\[a \times b = c\]

\[
\text{int times}(\text{var int: } a, \text{var int: } b, \text{var int: } c) = \quad (\text{assert (= (* a b) c)})
\]

\[a = b\]

\[
\text{int2float}(\text{var int: } a, \text{var float: } b) = \quad (\text{assert (= a b)})
\]

**Float constraints**

\[|a| = b\]

\[
\text{float abs}(\text{var float: } a, \text{var float: } b) = \quad (\text{assert (= (ite (> a 0) a (- a)) b)})
\]

\[a = b\]

\[
\text{float eq}(\text{var float: } a, \text{var float: } b) = \quad (\text{assert (= a b)})
\]

\[(a = b) \iff r\]

\[
\text{float eq reif}(\text{var float: } a, \text{var float: } b, \text{var bool: } r) = \quad (\text{assert (and (=> (= a b) r) (=> r (= a b))))}
\]

\[a \leq b\]

\[
\text{float le}(\text{var float: } a, \text{var float: } b) = \quad (\text{assert (<= a b)})
\]

\[(a \leq b) \iff r\]

\[
\text{float le reif}(\text{var float: } a, \text{var float: } b, \text{var bool: } r) = \quad (\text{assert (and (=> (<= a b) r) (=> r (<= a b))))}
\]

\[\sum_{i=1}^{n} as[i].bs[i] = c, \text{ where } n \text{ is the common length of } as \text{ and } bs\]
float_lin_eq(array [int] of float: as, array [int] of var float: bs, float: c) = 
    (assert (= (+ (* bs_ARRAY_.1 as.ARRAY_.1) (* bs_ARRAY_.2 as.ARRAY_.2)) (* bs_ARRAY_.3 as.ARRAY_.3)) c))

(\sum_{i=1}^{n} as[i].bs[i] = c) \iff r, where n is the common length of as and bs

float_lin_eq_ref(array [int] of float: as, array [int] of var float: bs, float: c, var bool: r) = 
    (assert (= (+ (* bs_ARRAY_.1 as.ARRAY_.1) (* bs_ARRAY_.2 as.ARRAY_.2) (* bs_ARRAY_.3 as.ARRAY_.3)) c) r) (= r (+ (+ (* bs_ARRAY_.1 as.ARRAY_.1) (* bs_ARRAY_.2 as.ARRAY_.2)) (* bs_ARRAY_.3 as.ARRAY_.3) c)))

(\sum_{i=1}^{n} as[i].bs[i] \leq c, where n is the common length of as and bs

float_lin_le_ref(array [int] of float: as, array [int] of var float: bs, float: c) = 
    (assert (= (+ (* bs_ARRAY_.1 as.ARRAY_.1) (* bs_ARRAY_.2 as.ARRAY_.2) (* bs_ARRAY_.3 as.ARRAY_.3)) c))

(\sum_{i=1}^{n} as[i].bs[i] \leq c) \iff r, where n is the common length of as and bs

float_lin_lt(array [int] of float: as, array [int] of var float: bs, float: c) = 
    (assert (= (+ (* bs_ARRAY_.1 as.ARRAY_.1) (* bs_ARRAY_.2 as.ARRAY_.2)) (* bs_ARRAY_.3 as.ARRAY_.3) c))

(\sum_{i=1}^{n} as[i].bs[i] < c, where n is the common length of as and bs

float_lin_le_ref(array [int] of float: as, array [int] of var float: bs, float: c, var bool: r) = 
    (assert (= (+ (* bs_ARRAY_.1 as.ARRAY_.1) (* bs_ARRAY_.2 as.ARRAY_.2) (* bs_ARRAY_.3 as.ARRAY_.3)) c) r) (= r (+ (+ (* bs_ARRAY_.1 as.ARRAY_.1) (* bs_ARRAY_.2 as.ARRAY_.2)) (* bs_ARRAY_.3 as.ARRAY_.3) c)))

(\sum_{i=1}^{n} as[i].bs[i] < c, where n is the common length of as and bs

float_lin_ne(array [int] of float: as, array [int] of var float: bs, float: c) = 
    (assert (= (+ (* bs_ARRAY_.1 as.ARRAY_.1) (* bs_ARRAY_.2 as.ARRAY_.2)) (* bs_ARRAY_.3 as.ARRAY_.3) c))

(\sum_{i=1}^{n} as[i].bs[i] \neq c, where n is the common length of as and bs

float_lin_ne_ref(array [int] of float: as, array [int] of var float: bs, float: c, var bool: r) = 
    (assert (= (+ (* bs_ARRAY_.1 as.ARRAY_.1) (* bs_ARRAY_.2 as.ARRAY_.2) (* bs_ARRAY_.3 as.ARRAY_.3)) c) r) (distinct (+ (+ (* bs_ARRAY_.1 as.ARRAY_.1) (* bs_ARRAY_.2 as.ARRAY_.2)) (* bs_ARRAY_.3 as.ARRAY_.3) c)))

a < b

float_eq(var float: a, var float: b) = 
    (assert (< a b))

(a < b) \iff r

float_le_ref(var float: a, var float: b, var bool: r) = 
    (assert (= r (< a b) r) (= r (< a b))))

max(a, b) = c

float_max(var float: a, var float: b, var float: c) = 
    (assert (= a b) (= b c) (= a c))

min(a, b) = c

float_min(var float: a, var float: b, var float: c) = 
    (assert (= a b) (= a c) (= b c))
\( a \neq b \)

\[
\text{float}_\text{ne}(\text{var float}: a, \text{var float}: b) = (\text{assert (distinct } a \text{ b)})
\]

\( (a \neq b) \iff r \)

\[
\text{float}_\text{ne}_\text{reif}(\text{var float}: a, \text{var float}: b, \text{var bool}: r) = (\text{assert (and (=> (distinct } a \text{ b) } r) (=> r (distinct } a \text{ b))))
\]

\( a + b = c \)

\[
\text{float}_\text{plus}(\text{var float}: a, \text{var float}: b, \text{var float}: c) = (\text{assert (= (+ } a \text{ b) } c))
\]

**Trigonometric and Exponential Functions**

The FlatZinc language allows to specify advanced constraints involving *trigonometric* and *exponential functions*. These functions have the following prototype:

\[
\text{constraint fun}_\text{name}(\text{var float}: a, \text{var float}: b);
\]

Since SmtLibv2 does not provide trigonometric and exponential functions, in general these constraints can’t be supported. However, whenever the first parameter happen to be a float literal, FZtoSL2 is able to compute on the fly the result of the operation and properly translate it into SmtLibv2. Note that this does not happen when the first value is an identifier of a parameter with a fixed value, although an improvement in this direction is possible.

\( \text{acos} \ 0.8 = b \)

\[
\text{float}_\text{acos}(0.8, \text{var float}: b) = (\text{assert (= 0.643501 } b))
\]

\( \text{asin} \ 0.8 = b \)

\[
\text{float}_\text{asin}(0.8, \text{var float}: b) = (\text{assert (= 0.927295 } b))
\]

\( \text{atan} \ 0.8 = b \)

\[
\text{float}_\text{atan}(0.8, \text{var float}: b) = (\text{assert (= 0.674741 } b))
\]

\( \text{cos} \ 0.8 = b \)

\[
\text{float}_\text{cos}(0.8, \text{var float}: b) = (\text{assert (= 0.674741 } b))
\]

\( \text{cosh} \ 0.8 = b \)

\[
\text{float}_\text{cosh}(0.8, \text{var float}: b) = (\text{assert (= 1.33743 } b))
\]

\( \text{exp} \ 0.8 = b \)

\[
\text{float}_\text{exp}(0.8, \text{var float}: b) = (\text{assert (= } 2.22554 \text{ } b))
\]

\( \text{ln} \ 0.8 = b \)

\[
\text{float}_\text{ln}(0.8, \text{var float}: b) = (\text{assert (= } -0.223144 \text{ } b))
\]

\( \text{log}_{10} \ 0.8 = b \)

\[
\text{float}_\text{log10}(0.8, \text{var float}: b) = (\text{assert (= } -0.90691 \text{ } b))
\]

\( \text{log}_2 \ 0.8 = b \)

\[
\text{float}_\text{log2}(0.8, \text{var float}: b) = (\text{assert (= } -0.321928 \text{ } b))
\]

\( \sqrt{0.8} = b \)
float_sqrt(0.8, var float: b):
    = (assert (= 0.894427 b))

\sin 0.8 = b

float_sin(0.8, var float: b):
    = (assert (= 0.717356 b))

\sinh 0.8 = b

float_sinh(0.8, var float: b):
    = (assert (= 0.888106 b))

tan 0.8 = b

float_tan(0.8, var float: b):
    = (assert (= 1.02964 b))

tanh 0.8 = b

float_tanh(0.8, var float: b):
    = (assert (= 0.664037 b))

Set Constraints

The set language construct and its associated constraints are currently not supported, since not required by the benchmarking comparison to be put in place. Follows a list of the statements that appear in the FlatZinc language.

\[ |a| = b \]

set_card(var set of int: a, var int: b)

\[ a - b = c \]

set_diff(var set of int: a, var set of int: b, var set of int: c)

\[ a = b \]

set_eq(var set of int: a, var set of int: b)

\[ (a = b) \iff r \]

set_eq\_reif(var set of int: a, var set of int: v, var bool: r)

\[ a \in b \]

set_in(var int: a, var set of int: b)

\[ (a \in b) \iff r \]

set_in\_reif(var int: a, var set of int: b, var bool: r)

\[ a \cap b = c \]

set_intersect(var set of int: a, var set of int: b, var set of int: c)

\[ a \subseteq b \lor \min(a \Delta b) \in a \]

set_le(var set of int: a, var set of int: b)

\[ a \subset b \lor \min(a \Delta b) \in a \]

set_lt(var set of int: a, var set of int: b)

\[ a \neq b \]

set_ne(var set of int: a, var set of int: b)

\[ (a \neq b) \iff r \]

set_ne\_reif(var set of int: a, var set of int: b, var bool: r)

\[ a \subseteq b \]

set_subset(var set of int: a, var set of int: b)
\[(a \subseteq b) \iff r\]

\[
\text{set_subset_reif(var set of int: a, var set of int: b, var bool: r)}
\]

\[
a \Delta b = c
\]

\[
\text{set_symdiff(var set of int: a, var set of int: b, var set of int: c)}
\]

\[
a \cup b = c
\]

\[
\text{set_union(var set of int: a, var set of int: b, var set of int: c)}
\]

### 2.1.4 Solve Goal

A goal in FlatZinc can be specified using one of the following three statements:

- `solve satisfy`
- `solve minimize variable`
- `solve maximize variable`

When the first line is encountered, a `(check-sat)` instruction is produced in output. The other two language constructs impose a little of difficulty, since OptiMathSat can actually solve only minimization problems. At the present moment, the approach taken by the FZtoSL2 tool is to declare on the fly a variable that is used as an alias of the goal variable in the case of a minimization problem or its dual in the case of a maximization problem.

The binary search put in place by OptiMathSat exploits the information regarding the bounds of the optimization variable, initialized within \([-\text{INF}]+\text{INF}\] otherwise. Hence the FZtoSL2 tool ensures that any domain restriction applied to the original target variable during its declaration is reproduced on its alias.

### 2.1.5 Annotations

Annotations are optional suggestions to the FlatZinc solver concerning how individual variables and constraints should be handled and how search should proceed. A short list of annotation types follows:

- **search annotations** allow to specify how the next variable to be assigned is chosen, how its value will be constrained and the search strategy;
- **output annotations** select the model output;
- **variable definition annotations** signaling where a variable has been defined the first time;
- **intermediate variables** signaling variables introduced during conversion from higher-level model to FlatZinc;
- **constraint annotations** advising how constraint should be implemented (e.g. propagation bounds setters);

An implementation of FlatZinc is free to ignore any annotations it does not recognize, although it should print a warning on the standard error stream if it does so.

In its current version, the program FZtoSL2 ignores all annotations, except in one case. Whenever FZtoSL2 is run with the option `–produce-models`, the tool adds at the end of the SmtLibv2 code the necessary instructions needed to display the content of the variables of interest of the model.

### 2.1.6 Usage of FZtoSL2

The FZtoSL2 tool requires always at least one input, which is the file to be encoded, while it can have zero or more discretionary options. Most of these options allow to set special configurations for SmtLibv2, as can be seen in the following list:

- `[-c|--print-success]`: sets up `print-success` flag to `true`;
- `[-r|--regular-output-channel] arg`: sets up the `regular-output-channel` to `arg` value;
• [-d|--diagnostic-output-channel] arg: sets up the diagnostic-output-channel to arg value;
• [-e|--expand-definitions]: sets up expand-definitions flag to true;
• [-i|--interactive-mode]: sets up interactive-mode flag to true;
• [-p|--produce-proofs]: sets up produce-proofs flag to true;
• [-u|--produce-unsat-cores]: sets up produce-unsat-cores flag to true;
• [-m|--produce-models]: sets up produce-models flag to true;
• [-a|--produce-assignments]: sets up produce-assignments flag to true;
• [-s|--random-seed] arg: sets up random-seed to arg value;
• [-v|--verbosity] arg: sets up verbosity to arg value;
• [-l|--set-logic] arg: sets up logic to arg value;
• [-f|--output-file] arg: sets up the output file of the conversion to arg value;

2.2 SL2toMZ: SmtLibv2 to MiniZinc

This section covers the aspects of the SmtLibv2 language that can be translated into MiniZinc thanks to the SL2toMZ tool. It should be mentioned here that the latter uses as a back end the SmtLibv2 parser library developed by Alberto Griggio, version 1.4+. Note that the back-end library has been hacked in a couple of points to allow a proper management of the target language, thus avoiding more complex and spaghetti-like programming on the front-end software.

2.2.1 SmtLibv2 Scripts

A SmtLibv2 model is commonly known as script, a file containing a sequence of commands that allow the communication with the SMT solver in a read-eval-print loop. It’s syntax is the following:

\[
\text{<script> ::= <command>∗}
\]

Just like what happened with in the other way around, only a restriction of the SmtLibv2 language is supported by the SL2toMZ tool:

• **Supported Commands**: statements that will be translated into the MiniZinc target language.

\[
\text{<command> ::= (assert <term>)}
\]
\[
\text{ (get−value <term>+) }
\]
\[
\text{ (declare−fun <symbol> (<sort>∗) <sort>)}
\]
\[
\text{ (define−fun <symbol> (<sorted−var>*) <sort> <term>)}
\]
\[
\text{ ...}
\]

Note that the MiniZinc language is first order only, hence only decision variables of the supported types can be declared.

• **Unsupported Commands**: statements that can not be mapped intoa corresponding construct of the MiniZinc language that have an impact on the specification of the model. If not differently specified in the <minizinc_common.h> file (see EXIT_MASK value), a command of this type causes the immediate termination of the execution.

\[
\text{<command> ::= ...}
\]
\[
\text{ (declare−sort <symbol> <numeral>)}
\]
\[
\text{ (define−sort <symbol> (<symbol>∗) <sort>)}
\]
\[
\text{ (push <numeral>)}
\]
\[
\text{ (pop <numeral>)}
\]
\[
\text{ (get−unsat−core)}
\]
\[
\text{ (get−assignment)}
\]
\[
\text{ ...}
\]
Note that the MiniZinc language does not allow the declaration of user defined sorts.

- **Ignored Commands**: statements that don’t have a corresponding mapping in the MiniZinc language which execution is not fundamental. In this set there are:

```plaintext
<command> ::= ...
  | (set-logic <symbol>)
  | (set-option <option>)
  | (set-info <attribute>)
  | (check-sat)
  | (get-assertions)
  | (get-proof)
  | (get-option <keyword>)
  | (get-info <info_flag>)
  | (exit)
```

### 2.2.2 Lexicon

The lexical tokens of the categories `<numeral>`, `<hexadecimal>` and `<binary>` are parsed and converted - when necessary - into an *unsigned numeral* stored in a string. A similar processing is applied to the lexicons belonging to the `<decimal>` category.

Tokens of the categories `<string>`, `<symbol>` or reserved words are left as is and consumed or stored at need, while `<keyword>` symbols are ignored.

### 2.2.3 Term and Formulas

In the SmtLibv2 syntax terms are constructed out of constant symbols, variables, function symbols, binders (forall, exists, let) and the reserved symbol `!`, used as annotation operator. As it is shown in the grammar specification below, observe that there is no syntactical distinction among functions and predicate symbols, which are simply function symbols whose result is `bool`.

```plaintext
<qual_identifier> ::= <identifier> | (as <identifier> <sort>)
<var_binding> ::= (<symbol> <term>)
<sorted_var> ::= (<symbol> <sort>)
<term> ::= <spec_constant>
  | <qual_identifier>
  | (let <var_binding> <term>+++)
  | (forall <sorted_var> <term>)
  | (exists <sorted_var> <term>)
  | (! <term> <attribute>+++)
```

The universal and existential quantifiers are not yet supported in the SL2toMZ tool, while annotations are ignored.

### 2.2.4 Logics and Theories

The SmtLibv2 comes with a number of theories, not all of which have been mapped in the MiniZinc, partly due to the intrinsic limitations of the target language and to the focus over a specific set of problems. Of those supported there are:

- **The Core Theory**, which contains the basic elements of the Boolean logic.
- **The Ints Theory**, which contains the basic elements of integer arithmetic;
- **The Reals Theory**, which contains a theory for rational arithmetic;
- **The Reals_Ints Theory**, which contains elements crossing the Integer and Rational domain. Note that the statements `to_real` of SmtLibv2 is emulated with `int2float` of MiniZinc, `to_int` with the `floor` function and `is_int` with a personalized predicate. Note that the `floor` function requires its argument to be a parameter and not a decision variable.
Hence, the SL2toMZ tool should correctly translate scripts of the logics: \( QF\_UF, QF\_LIA, QF\_NIA,\) \( QF\_LRA.\)

### 2.2.5 ITE and LET statements

A few more words should be spent in respect to the ITE and LET statements.

The ITE construction of SmtLibv2 can not mapped into the ITE construction of MiniZinc, since the latter requires the guard to not involve decision variables. Instead, this syntax construction is translated into a more flexible implication construction through a tree reorganization. For example, consider the following SmtLibv2 code:

```plaintext
(declare-fun x () Int)
(declare-fun y () Int) ;; with no known value

(assert (= x (ite (> y 5) (2) (3))))
```

Since the guard refers to a variable which value is not known at parsing time, the assertion can not be mapped into the MiniZinc if-then-else-end statement. Also, since the type of the then/else branches is not boolean such a constraint can not be translated into MiniZinc without the aid of an interface variable, like this:

```plaintext
var int: x;
var int: y;
var bool: new_bool_01;
var int: new_int_02;

constraint (new_bool_01 = (y > 5));
constraint (((not new_bool_01) \(\lor\) (new_int_02 = 2)) \(\lor\)
            ((new_bool_01) \(\lor\) (new_int_02 = 3)));
constraint (x = new_int_02);
```

This behaviour can be disabled in the future, when the capabilities of MiniZinc will be expanded, by removing the ENABLE\_ITE\_HACK flag from the `<debug_flags.h>` file.

The limitations of the ITE construction had an impact over the LET construction too, since the constraints introduced to support ITE must have the possibility of referring to the variables introduced with a local scope.

The SL2toMZ tool allows two approaches in respect to the LET statements:

- The expansion of each and every local variable introduced in all its innermost references inside the formula. In this case there is no more any local scope to deal with, hence the management of the ITE construction is not affected;

- The direct mapping of the SmtLibv2 LET construction into the MiniZinc LET lexicon. In this case, each and every time an ITE expression must be expanded, there is no other general solution than reproducing for any newly introduced assertion the same local context. This solution spares space in the translation of the formulas, although it will force the MiniZinc solvers to evaluate over and over the same expressions.

This behaviour can be set by acting on the ENABLE\_LET\_HACK flag in the `<debug_flags.h>` file: its removal will force the expansion of the local variable declarations during the translating process.

### 2.2.6 Solve Goal

The solve goal is determined by the option with which the SL2toMZ tool is launched, whereas the check-sat statements are ignored. Hence a model will be searched for a satisfactory solution when no option is specified, or else it will require the specified decision variable to be optimized. Note that it’s not possible to automatically assign any search strategy to the MiniZinc model.
2.2.7 Usage of SL2toMZ

Unlike FZtoSL2, the SL2toMZ program is designed to take its input as a stream, just like the MathSat and OptiMathSat solvers. It also accepts a brief list of discretionary options:

- \([-h]\): prints the help on screen;
- \([-f] \, arg\): sets the output file to \(arg\), otherwise the stdout is used;
- \([-v] \, arg\): sets the name of the decision variable to be optimized. If not specified, the tool defaults to the satisfy check only;
- \([-a] \, arg\): sets the optimization type to \(\{min, max\}\), defaults to \(min\);
- \([-b] \, arg\): sets the absolute value that any float variable of the research space can achieve. This option has been introduced since most target solvers use propagation based solving techniques that will suffer when there are no specified bounds;

3 Benchmarks

Once the tools have been implemented and verified for correctness, a number of benchmarks have been done in order to compare the performances of the solvers belonging to the two different problem domains. The machine on which this phase has been accomplished had the following characteristics:

Fedora 18 x86_64, kernel 3.8.0−17
Intel Core i7 Processor 2630QM: 2.2/2.9 Ghz
DDR3 1333 Mhz 16 GB SDRAM

with the CPU set to maximum performance with hyper threading technology enabled.

Before proceeding with the benchmarks, I decided to verify the correctness of the developed tools by translating SmtLibv22 problems into MiniZinc and backward, thus checking that the optimum or satisfiable models remained consistent. This has been done on a randomly chosen set of 30 files, over which no problem has been detected.

3.1 Benchmarks: QF_LRA

Since OptiMathSat is capable of optimizing problems in the QF_LRA domain, the initial goal of the entire research project was indeed to determine its performances in this problem area in respect to the CP/MILP solvers capable of parsing problems written in the MiniZinc language. Unfortunately, although the MiniZinc language already extensively supports floats, most of the solvers do not. Only three solvers were availabl:

- Sicstus Prolog, could not be tested since is not a free ware;
- Gecode, its latest version (4.0.0) introduced support to float decision variables. Although the latest version of the svn repository has been used, the experience with the solver has been characterized by the following problems:
  - A bug that prevented the \(\text{fzn-gecode}\) tool from accepting float optimization variables has been identified, reported and immediately solved by the maintainers;
  - A number of memory management bugs affecting the tool have been detected, although it was impossible to identify the problem. Hence a number of models have been sent to the gecode project maintainers, and finally the issues appears to be solved;
  - The solver still requires that each float decision variable domain is bounded with a strict and explicit interval. This issue has been solved by adding to the translating tool a special option that allows to set an upper bound to the absolute value of any float variable within a model.
  - The solver does not terminate within 20 minutes on any of the tested benchmarks, excluding the case of toy examples or benchmarks augmented with a satisfiable model solution;
The fzn-gecode solver did not perform well on any of the tested benchmarks, as shown in the scatter plot 2. For what regards OptiMathSat, on average the linear approach performed better with an average search time of 74s, while the mixed scored 120s and the adaptive 115s.

![Figure 2: Result of Benchmarking evaluation of optimathsat and fzn-gecode on the same set of models, with time limit set to 20m.](image)

- **Eclipse**, whose fzn_ic search engine was indeed able to terminate within 20 minutes on some satisfiable problems. The average time required by eclipse on the overall set of benchmarks (1314 models) is 1002.31s, whereas OptiMathSat required 98.69s. As it is possible to see in figure 3, eclipse outperformed OptiMathSat on 76 problems of the Strip Packing set (413 models).

![Figure 3: Result of Benchmarking evaluation of optimathsat and eclipsee on the same set of models, with time limit set to 20m.](image)

Restricting the comparison in between the two solvers on the Strip Packing problems, it can be observed that on average eclipse search terminated within 573.93s while OptiMathSat found the solution in 43.71s. The Strip Packing problems with a higher number of variables and a higher boolean combinatorial com-
putation tend to be hard for eclipse, which did perform generally better than OptiMathSat on the simpler models.

3.2 Experiments

Given the particularly poor performances of the tested tools, I decided to perform some further experimentation. In order to verify that there wasn’t any particular issue with the benchmarks fed to the solvers, I generated for each minimization problem an optimal solution model from which I extracted the final values of all the float variables. Using this data, a new and simplified problem has been generated in which each float variable domain was restricted to an interval of ±1 around the solution value. Despite of this aid the gecode tool did not converge within 4 hours, whereas eclipse did show an improvement and was able to terminate in 1h13m. Note that this work has been done on a limited number of test samples, and may not reflect the general situation of the overall pool of benchmarks.

Referring to various references online, it emerged that these CLP tools implement constraint propagation and interval narrowing techniques during their research. As a consequence both the interval sizes of float variables and the definition of epsilon may affect the search time. In the literature the problems of "slow convergence" and "early quiescence" that affect interval narrowing are well described and understood, although it is not known by the author of this report whether the advanced narrowing techniques that solve these problems are implemented within the used tools.

I wanted to verify to what extent these issues could be at the root of the poor performances detected by the benchmarking evaluations, therefore I performed a sequence of experiments based on very simple problems taken out from LP problem domain.

3.2.1 Gecode

Let’s consider an optimization problem in which the cost function is \( \text{cost} = x + y \) with the following system of inequalities:

\[
\begin{aligned}
-2.196116 &\leq \omega y + (0.464961)x; \quad (eq1) \\
-2.546293 &\leq (0.009998)x + (-0.476432)y; \quad (eq2) \\
1.198102 &\leq (-0.407378)x + (0.106645)y; \quad (eq3) \\
-10 &\leq x \leq 10; \\
-10 &\leq y \leq 10;
\end{aligned}
\]

where \( \omega = 10^{-17} \). The associated polytope is shown in figure 4:

![Figure 4: Polytope associated to the first system of equations](image)
The problem can be encoded into MiniZinc as follows:

```plaintext
var -10.0..10.00: x;
var -10.0..10.00: y;
float: omega = 0.0000000000000001;
var float: cost;
constraint (-2.196116 <= (omega)*y + (0.464961)*x);
constraint (-2.546293 <= (0.009998)*x + (-0.476432)*y);
constraint (1.198102 <= (-0.407378)*x + (0.106645)*y);
constraint (cost = x + y);
solve minimize cost;
output [
  "x = ", show(x), "\n",
  "y = ", show(y), "\n",
  "cost = ", show(cost), "\n"
]
```

The problem can be converted into FlatZinc and easily solved with fzn-gecode:

```
$ mzn2fzn dummy.mzn ; fzn-gecode -n 1 -restart geometric -mode stat dummy.fzn
```

```
cost = -11.5312009586246;
x = -4.72322624908325;
y = -6.80797470954136;
```

---

As we can see, apparently gecode is indeed able to easily tackle problems with float variables when their domain is reasonably small.

**Introducing polytopes variations**

Let’s slightly modify the previous problem by giving increasing values to \( \omega \) (\( \times 10 \)), an action that can be approximatively imagined as increasing the angular coefficient of the line associated to the cut eq1. The results are shown in table 2.

<table>
<thead>
<tr>
<th>Stats</th>
<th>10^{-17}</th>
<th>10^{-16}</th>
<th>10^{-15}</th>
<th>10^{-14}</th>
<th>10^{-13}</th>
<th>10^{-12}</th>
<th>10^{-11}</th>
<th>10^{-10}</th>
<th>10^{-9}</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime: (s)</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.013</td>
<td>1.444</td>
<td>14.193</td>
<td>138.152</td>
<td>1336.185</td>
<td>1336.185</td>
</tr>
<tr>
<td>solvetime: (s)</td>
<td>0.002</td>
<td>0.002</td>
<td>0.003</td>
<td>0.019</td>
<td>0.163</td>
<td>1.443</td>
<td>14.193</td>
<td>138.152</td>
<td>1336.184</td>
</tr>
<tr>
<td>solutions:</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>15</td>
<td>140</td>
<td>1474</td>
<td>14484</td>
<td>145078</td>
<td>14532536</td>
</tr>
<tr>
<td>variables:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>propagators:</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>propagations:</td>
<td>4291</td>
<td>4908</td>
<td>8695</td>
<td>50785</td>
<td>491167</td>
<td>4666765</td>
<td>45648544</td>
<td>446198811</td>
<td>4374573070</td>
</tr>
<tr>
<td>nodes:</td>
<td>1184</td>
<td>1295</td>
<td>1929</td>
<td>8325</td>
<td>71561</td>
<td>645615</td>
<td>6017795</td>
<td>55076329</td>
<td>501104980</td>
</tr>
<tr>
<td>failures:</td>
<td>150</td>
<td>166</td>
<td>289</td>
<td>1225</td>
<td>10480</td>
<td>96300</td>
<td>905384</td>
<td>8498650</td>
<td>77787280</td>
</tr>
<tr>
<td>restarts:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>peak depth:</td>
<td>1129</td>
<td>1128</td>
<td>1122</td>
<td>1122</td>
<td>1116</td>
<td>1109</td>
<td>1106</td>
<td>1101</td>
<td>101</td>
</tr>
<tr>
<td>memory: (KB)</td>
<td>486</td>
<td>522</td>
<td>958</td>
<td>4545</td>
<td>40960</td>
<td>393955</td>
<td>3903921</td>
<td>38983332</td>
<td>376127605</td>
</tr>
</tbody>
</table>

Table 2: Search statistics by increasing values of \( \omega \)

The solution produced by the various searches is stable around its value with an error \( \leq 10^{-9} \). The same model has been tested with OptiMathSat which returned the same solution in 0.000s with \( \omega = 10^{-17} \)
and 0.004s with $\omega = 10^{-9}$. The latter solver did (correctly) not change its solution, thanks to the built-in infinite precision arithmetic.

Observations:

- The problem in $\mathbb{R}$ domain would have a unique solution;
- The fzn-gecode solver finds an increasing number of “solutions” by increasing values of $\omega$. This relation seems to be linear also in terms of propagations, nodes and failures;
- By stopping the search at increasing time lapses, it can be shown that the intermediate solutions live along the line that corresponds to the constraint $eq1$. Since the number of solutions visited increases linearly over time, I suggest that the search is also proceeding with a linear convergence toward the optimal solution;

Open Questions:

- Why do performance decrease so much on a slight perturbation of the problem?
- Are there options that may improve the search time?

Playing with solver options

To verify the possibility of improvement, the problem has been fixed with $\omega = 10^{-11}$ and run with different command line options. None of the restart options did improve the performances (this seems due to the fact that no restarts are performed for this particular problem), as it happened with the decay factor option. The recomputation commit distance ($-c-d$) option could bring the solve time down to 0.668s from the initial 14.193s. This gain is a result of a reduced number of solution visited, but it should be said that the relation among the values of this parameter and the solve time could not be clearly understood, as it does not show general a clear trend depending on the value of input. Therefore this result should be considered just an ad-hoc improvement that may work nicely only on the current problem but not on a wider set of models. The recomputation adaption distance did slightly improve the performances when reduced to 0.

Using gradient-parallel constraint

Another test is to substitute the constraint $eq1$ with a new inequality that is parallel to the gradient of the cost function, and verify how the solver behaves. That is for example:

$$-5 \leq \theta \cdot y + x, \theta = 1.0$$

Such a system would have infinite solutions in $\mathbb{R}$. Besides that, as it is shown in the following piece of text, the fzn-gecode solver immediately finds one optimal solutions (in $\mathbb{Q}$):

```
%  runtime:  0.001 (1.946 ms)
%  solvetime:  0.001 (1.571 ms)
%  solutions:  1
%  variables:  0
%  propagators:  3
%  propagations:  723
%  nodes:  100
%  failures:  13
%  restarts:  0
%  peak depth:  99
%  peak memory:  47 KB
```

Then we try again to perturb the line by changing its angular coefficient in both directions. With $\theta = 1.1$ it converges in 21.685s:

```
%  runtime:  21.685 (21685.548 ms)
%  solvetime:  21.685 (21685.203 ms)
%  solutions:  17697
%  variables:  0
%  propagators:  4
```
Whereas with several values $\theta$ in the interval $[0, 1]$ we fall back in a slow convergence behaviour.

**Reducing the variable domain (1)**

To clear out all the possibilities one may wonder whether by reducing the size of a polytope the search performances improve or not. This may happen thanks to a reduction in the domain of admissible values for float variables, as it has been suggested during a technical exchange with a developer of gecode. Let’s start from the following system of equalities, using the same cost function as before:

$$\begin{align*}
(-5.0 \leq 0.1 * y + x); & \quad (eq4) \\
(3.0 \leq 1.1 * y - 0.7 * x); & \quad (eq5) \\
(-2.196116 \leq \omega * y + (0.464961) * x); & \quad (eq1) \\
(-2.546293 \leq (0.009998) * x + (-0.476432) * y); & \quad (eq2) \\
(1.198102 \leq (-0.407378) * x + (0.106645) * y); & \quad (eq3) \\
-10 \leq x \leq 10; & \\
-10 \leq y \leq 10;
\end{align*}$$

Such a system is shown in figure 5, and as before the solver is not able to find it within 20m. With the help of a plot tool, I will try to scale down the polytope while maintaining the proportions of the satisfiable region. Since this job has been done by hand it only had a partial success, the result is depicted in figure 6 and in the new system of equations:
\[
\begin{align*}
-3.378222 &\leq -0.1 \cdot y + x; \quad (eq6) \\
4.455555 &\leq 1.1 \cdot y - 0.7 \cdot x; \quad (eq7) \\
-0.99888 &\leq (0.009998) \cdot x + (-0.476432) \cdot y; \quad (eq2) \\
1.51 &\leq (-0.407378) \cdot x + (0.106645) \cdot y; \quad (eq3) \\
-10 &\leq x \leq 10; \\
-10 &\leq y \leq 10;
\end{align*}
\]

Figure 6: Polytope associated to third system of equations

The new problem is solved in a few seconds:

- **runtime**: 4.205 (4205.576 ms)
- **solvetime**: 4.205 (4205.195 ms)
- **solutions**: 41674
- **variables**: 0
- **propagators**: 5
- **propagations**: 5073414
- **nodes**: 424745
- **failures**: 93644
- **restarts**: 0
- **peak depth**: 60
- **peak memory**: 191787 KB

Although interesting, I admit that this experiment may not be particularly relevant, since I can not assess how much the human intrusion during the miniaturization process did affect the result obtained.

### Reducing the variable domain (2)

A better idea is to automatically generate polytopes with a small area, just to see whether it is generally true that strict bounds over float variables help search performances. After a python script has been set up for this purpose, we immediately find a counterexample (figure 7):
\[
\begin{align*}
(y \leq (-1.872908) \times x + 0.000001); & \quad (eq8) \\
(y \leq (-2.513482) \times x + 0.000008); & \quad (eq9) \\
(y \geq (-3.956500) \times x - 0.000006); & \quad (eq10) \\
(y \geq (2.625192) \times x + 0.000008); & \quad (eq11) \\
-0.00001 \leq x \leq 0.00001; & \\
-0.00001 \leq y \leq 0.00001; & 
\end{align*}
\]

The problem is not solved by \texttt{fzn-gecode} within 20m. This fact does disprove the initial hypothesis that the size of the domain of \texttt{float} variables is all there is that really prevents \texttt{gecode} from properly solving the \texttt{OMT} benchmarks.

### 3.2.2 Eclipse

The last tests are aimed to identifying the limitations of the \texttt{eclipse} search tool. Generally speaking, \texttt{eclipse} outperforms \texttt{gecode} in the sense that it is capable to solve more LP problems and in a shorter amount of time. These results confirm the behaviour identified during the benchmarks evaluation phase.

### 2D LP problems with boolean constraints

The model used in this test is a linear programming problem where the polytope region in the 2D plane is described by 50 lower constraints plus a couple of loose upper and lower bounds. Each constraint is attached to a boolean flag and the solver is instructed to find a solution that satisfies a certain fixed number of constraints. The results are shown in table 3. The first time column refers to a problem in which at least N lower bounds where required to be satisfied, while in the second time column are shown the results for exactly N constraints satisfied.

In table 3 we clearly see that the more the model is constrained the better are the performances of the solver. Thus the presence of boolean formulas seems to affect the search performances when it is dominant. Interestingly, the solver did not terminate within 20m for the value 10, for undiscovered reasons.
<table>
<thead>
<tr>
<th>Constraints</th>
<th>Time (s) (≤)</th>
<th>Time (s) (≥)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 (AND)</td>
<td>5.560</td>
<td>5.360</td>
</tr>
<tr>
<td>49</td>
<td>1.510</td>
<td>1.500</td>
</tr>
<tr>
<td>48</td>
<td>6.310</td>
<td>6.360</td>
</tr>
<tr>
<td>46</td>
<td>7.670</td>
<td>7.670</td>
</tr>
<tr>
<td>45</td>
<td>2.350</td>
<td>2.370</td>
</tr>
<tr>
<td>44</td>
<td>0.590</td>
<td>0.570</td>
</tr>
<tr>
<td>41</td>
<td>2.310</td>
<td>2.250</td>
</tr>
<tr>
<td>40</td>
<td>2.910</td>
<td>2.950</td>
</tr>
<tr>
<td>39</td>
<td>13.140</td>
<td>12.830</td>
</tr>
<tr>
<td>38</td>
<td>16.420</td>
<td>16.140</td>
</tr>
<tr>
<td>35</td>
<td>15.190</td>
<td>15.470</td>
</tr>
<tr>
<td>34</td>
<td>20.680</td>
<td>20.560</td>
</tr>
<tr>
<td>31</td>
<td>30.020</td>
<td>30.210</td>
</tr>
<tr>
<td>30</td>
<td>35.400</td>
<td>36.360</td>
</tr>
<tr>
<td>29</td>
<td>37.070</td>
<td>38.370</td>
</tr>
<tr>
<td>26</td>
<td>28.900</td>
<td>30.880</td>
</tr>
<tr>
<td>25</td>
<td>28.840</td>
<td>29.740</td>
</tr>
<tr>
<td>24</td>
<td>16.670</td>
<td>17.910</td>
</tr>
<tr>
<td>21</td>
<td>33.560</td>
<td>34.570</td>
</tr>
<tr>
<td>20</td>
<td>68.990</td>
<td>68.470</td>
</tr>
<tr>
<td>19</td>
<td>57.360</td>
<td>54.960</td>
</tr>
<tr>
<td>16</td>
<td>43.600</td>
<td>41.060</td>
</tr>
<tr>
<td>15</td>
<td>42.530</td>
<td>36.810</td>
</tr>
<tr>
<td>14</td>
<td>23.430</td>
<td>17.580</td>
</tr>
<tr>
<td>11</td>
<td>44.620</td>
<td>29.530</td>
</tr>
<tr>
<td>10</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>9</td>
<td>42.000</td>
<td>23.160</td>
</tr>
<tr>
<td>6</td>
<td>30.020</td>
<td>6.770</td>
</tr>
<tr>
<td>5</td>
<td>43.450</td>
<td>19.420</td>
</tr>
<tr>
<td>4</td>
<td>45.190</td>
<td>19.830</td>
</tr>
<tr>
<td>2</td>
<td>38.890</td>
<td>9.710</td>
</tr>
<tr>
<td>1 (OR)</td>
<td>45.420</td>
<td>12.340</td>
</tr>
<tr>
<td>0</td>
<td>43.940</td>
<td>9.730</td>
</tr>
</tbody>
</table>

Table 3: Search statistics for a varying number of constraints to be satisfied
Higher dimensional LP problems

A possible expansion of previous test is to increase the number of variables that are taken into consideration inside the problem.

Minimizing the same cost function over a problem with 4 float variables while satisfying 6 over 8 constraints required 248.51s of computational time. With 5 variables the search time increases up to 913.130s. Although this test has been done on a reduced number of models, it can be shown that the time increases further by increasing the number of float variables even when the number of boolean constraints does not change. This is to be expected since the combinatorial effects increase with the number of the variable in any case.

4 Conclusion and Future Work

The work done during this research project experience has shown that the current implementation of OptiMathSat outperforms by several degrees of magnitude the tested CP/MILP solvers on its own set of OMT benchmarks. Although interesting, it was not possible to do this comparison using optimization problems with float decision variables taken from the CP/MILP domain, since absent. The tested CP/MILP solvers appear to show different difficulties against the used benchmarks. One one hand gecode does show some serious limitation even on certain simple LP problems, without regard for the presence of boolean variables nor of the size of the domain of float variables. On the other hand the eclipse solver seems to be able to outperform OptiMathSat only on simpler problems, those that are not dominated by boolean combinatorial search and do not need to consider too many variables. Nonetheless most FlatZinc solvers do not support float decision variables but are planning to do so, hence the current situation may change in the near future.

This project experience had the invaluable merit of allowing me to deepen the knowledge on a number of topics, bringing enhancements both to my programming skills and to my theoretical background. Still the research done in this document does look like it is unfinished, since more significant comparison could be done once OptiMathSat will be able to solve problems in the $\texttt{QF\_LIA}$ domain where CP/MILP FlatZinc solvers are currently strong.