THE AMPL INTERFACE TO CONSTRAINT PROGRAMMING SOLVERS

Victor Zverovich, Robert Fourer

AMPL Optimization

The 13th INFORMS Computing Society Conference (ICS)
January 6th - 8th, 2013, Santa Fe, New Mexico, USA

WHY AMPL?

- AMPL is a popular algebraic modeling language:
  - used in businesses, government agencies, and academic institutions (over 100 courses in 2012)
  - large community
    (> 1,300 members in AMPL Google Group alone)
  - the most popular input format on NEOS
    (> 200,000 or 57% submissions in 2012)
- AMPL supports a wide range of problem types: linear, mixed integer, quadratic, second-order cone, nonlinear, complementarity problems and more.

CONSTRAINT PROGRAMMING IN AMPL

- **Constraint programming** (CP) allows natural formulation of many combinatorial optimization problems.
- AMPL has supported CP since early 2000s when the paper Extending an Algebraic Modeling Language to Support Constraint Programming [1] was published.
- CP solvers connected to AMPL:
  - **IBM/ILOG CP Optimizer** (ilogcp)
  - **Gecode**

SUPPORTED CP CONSTRUCTS

- Logical operators: `and`, `or`, `not`; iterated `exists`, `forall`
- Conditional operators: `if-then`, `if-then-else`, `==>`, `==> else`, `<==`, `<==>`
- Counting operators: iterated `count`, `atmost`, `atleast`, `exactly`, `numberof`
- Pairwise operators: `alldiff`

See **http://www.ampl.com/NEW/LOGIC/index.html**

AMPL SOLVER LIBRARY

AMPL Solver Library (ASL) is an open source library for connecting solvers to AMPL.

- Available from:
  - Netlib: **http://www.netlib.org/ampl/**
  - GitHub: **https://github.com/vitaut/ampl**
- Includes drivers for several solvers:
  - CPLEX
  - Gurobi
  - MINOS

  ...

AMPL SOLVER INTERFACES

- C interface:
  - described in **Hooking Your Solver to AMPL**
  - used by most solvers
- C++ interface (new):
  - a very thin wrapper around the C interface
  - type-safe, no casts needed when working with expression trees
  - easy to use, less boilerplate code
  - efficient
  - used by **ilogcp** and **gecode**

PERFORMANCE

Problem	Input+Conv Time	C API Time	C++ API Time	C/C++ Ratio
assign1	2.001	0.0156233	0.0154757	1.009537533
balassign1	2.106	0.086136	0.085996	1.0016279827
flowshp1	0.529	0.000594689	0.000598812	0.9931147004
flowshp2	0.784	0.000593329	0.00059615	0.9952679695
magic	0.558	0.00276411	0.0028042	0.9857035875
mapcoloring	0.557	8.6488E-007	8.6538E-007	0.9994222191
money	0.472	0.000241843	0.000238001	1.0161427893
nqueens	1.073	8.6449E-007	8.6593E-007	0.998337048
sched1	0.595	0.0172613	0.0174135	0.9912596549
sched2	0.684	0.0142709	0.0146368	0.9750013664
party1	33.688	3.88768	3.94516	0.9854302487
party2	126.194	1.20008	1.21847	0.9849073018
sudokuHard	0.661	8.6618E-007	8.6361E-007	1.0029758803
sudokuVeryEasy	0.726	8.6547E-007	8.6383E-007	1.0018985217
				0.9957590574

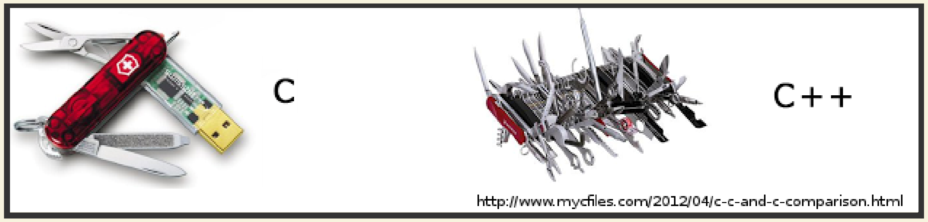
Time is in milliseconds.

SOLVER WORKFLOW

1. Get a problem instance in AMPL form
2. Process solver options
3. Convert the problem from AMPL to solver form
4. Solve the problem
5. Return solution(s)

Step 3 is the most interesting especially for a CP solver, because it has to deal with expression trees. Other steps are easy to implement.

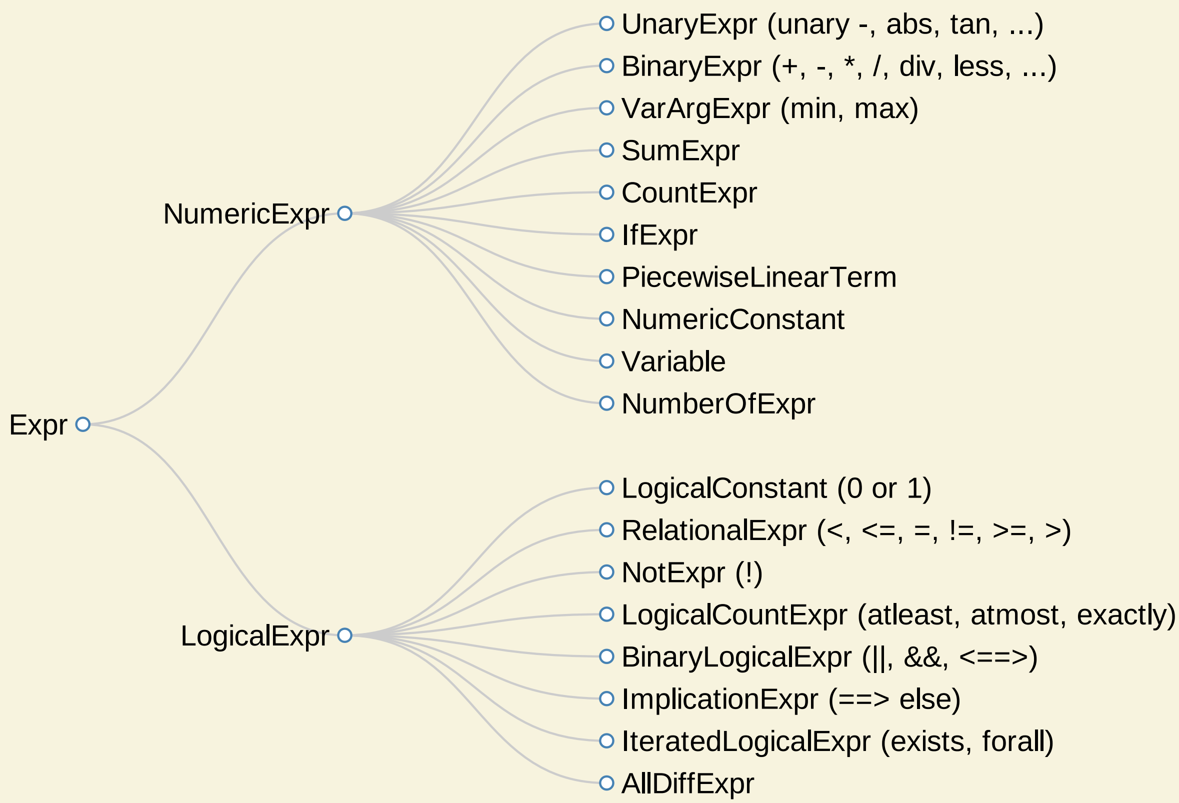
CHOICE OF LANGUAGE



Both IBM/ILOG CP Optimizer and Gecode use C++ for their main APIs. Therefore I'll give all examples in C++ with the new interface library.

However, everything discussed here is possible to do with the C API with a bit more work.

EXPRESSION TREES



WORKING WITH EXPRESSIONS

- `c.value()` returns the value of a numeric or logical constant `c`.
- `e.arg()` returns an argument of a unary expression `e`.
- `e.lhs()` and `e.rhs()` return arguments of a binary expression `e`.
- `Cast<ExprType>(e)` casts expression `e` to `ExprType` if possible, otherwise returns a null expression. Fast alternative to `dynamic_cast`.

ITERATING OVER ARGUMENTS

- Iterating over arguments of an expression `e` with variable number of arguments:

```
for (auto arg: e) // C++11
  // use arg
```

or

```
for (SumExpr::iterator i = e.begin(); i != e.end(); ++i)
  // use *i
```

- Works with `VarArgExpr`, `SumExpr`, `CountExpr`, `NumberOfExpr`, `IteratedLogicalExpr` and `AllDiffExpr`.

TREE TRAVERSAL WITH VISITORS

ilogcp

```
class IlogCPSolver :
  public ExprVisitor<IlogCPSolver, IloExpr, IloConstraint> {
 public:
  // Convert logical expressions.
  // Convert numeric expressions.
};
```

gecode

```
class NLToGecodeConverter :
  private ExprVisitor<NLToGecodeConverter,
                      Gecode::LinExpr, Gecode::BoolExpr> {
  // Convert logical expressions.
  // Convert numeric expressions.
 public:
  LinExpr ConvertFullExpr(NumericExpr e) { return Visit(e); }
  BoolExpr ConvertFullExpr(LogicalExpr e) {
    // Process global constraints calling Visit(e)
    // for nested expressions.
  }
};
```

CONVERTING NUMERIC EXPRESSIONS

```
IloExpr VisitNumericConstant(NumericConstant n) {
  return IloExpr(env_, n.value());
}

IloExpr VisitVariable(Variable v) {
  return vars_[v.index()];
}

IloExpr VisitPlus(BinaryExpr e) {
  return Visit(e.lhs()) + Visit(e.rhs());
}

IloExpr VisitPow(BinaryExpr e) {
  return IloPower(Visit(e.lhs()), Visit(e.rhs()));
}

IloExpr VisitSum(SumExpr e) {
  IloExpr sum(env_);
  for (auto arg: e) // C++11
    sum += Visit(arg);
  return sum;
}
```

CONVERTING LOGICAL EXPRESSIONS

```
IloConstraint VisitLogicalConstant(LogicalConstant c) {
  return IloNumVar(env_, 1, 1) == c.value();
}

IloConstraint VisitEqual(RelationalExpr e) {
  return Visit(e.lhs()) == Visit(e.rhs());
}

IloConstraint VisitGreater(RelationalExpr e) {
  return Visit(e.lhs()) > Visit(e.rhs());
}

IloConstraint VisitAnd(BinaryLogicalExpr e) {
  return Visit(e.lhs()) && Visit(e.rhs());
}

IloConstraint IlogCPSolver::VisitExists(IteratedLogicalExpr e) {
  IloOr disjunction(env_);
  for (auto arg: e) // C++11
    disjunction.add(Visit(arg));
  return disjunction;
}
```

HANDLING NUMBEROF

```
class IlogCPSolver {
  // CreateVar is a functor that creates an IlogCP variable.
  NumberOfMap<IloIntVar, CreateVar> numberofs_;
  ...
};

IloExpr IlogCPSolver::VisitNumberOf(NumberOfExpr e) {
  NumericExpr value = e.value();
  if (NumericConstant num = Cast<NumericConstant>(value))
    return numberofs_.Add(num.value(), e);
  IloExpr sum(env_);
  IloExpr concert_value(Visit(value));
  for (Expr arg: e)
    sum += (Visit(arg) == concert_value);
  return sum;
}
```

`NumberOfMap` is a map from `numberof` expressions with the same argument lists to values and corresponding variables. Such expression can be converted to a single `IloDistribute` constraint.

EXPRESSION VISITOR

Copy the great architectures.
-- Edward Tufte

- Inspired by the AST visitor from the **Clang compiler frontend**
- Visitor design pattern with static instead of dynamic polymorphism
- Uses **curiously recurring template pattern**
- Very efficient: no virtual function calls, `Visit*` functions can be inlined

TWO-LEVEL CONVERSION

1. Top level - global constraints such as `alldiff` and possible optimizations for the case when expression value is not used
   - ilogcp: no extra work is necessary, the Concert interface does necessary processing
   - gecode: manual handling of `alldiff` in `ConvertFullExpr`
2. General case for nested expressions

   Example:

   ```
   s.t. c: alldiff ({j in 1..n} Row[j]+j);
   ```

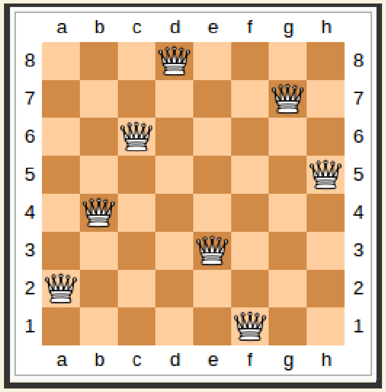
   `alldiff (...)` - top level expression

   `Row[j] + j` - subexpression

SUPPORTING MULTIPLE SOLVERS

- Separate hierarchies for logical and numeric expressions (ilogcp and gecode) are handled easily
- Possible to deal with more complex expression hierarchies, but with more efforts
- Not necessary to convert all expressions, solver will report an error when unhandled expression is encountered and exit gracefully. For example, gecode doesn't support many nonlinear expressions.

EXAMPLE




```
# Place n queens on an n by n board
# so that no two queens can attack
# each other (nqueens.mod).

param n integer > 0;
var Row {1..n} integer >= 1 <= n;

subj to c1: alldiff ({j in 1..n} Row[j]);
subj to c2: alldiff ({j in 1..n} Row[j]+j);
subj to c3: alldiff ({j in 1..n} Row[j]-j);
```

More examples available at
http://www.ampl.com/NEW/LOGIC/examples.html

EXAMPLE - ILOGCP

```
ampl: model nqueens.mod;
ampl: let n := 20;
ampl: option solver ilogcp;
ampl: solve;
ilogcp 12.4.0: feasible solution
2898 choice points, 1286 fails
ampl: display Row;
Row [*] :=
 1  14
 2  16
 3   8
 4   6
 5  15
 6   3
  ...
19  10
20   5
;
```

EXAMPLE - GECODE

```
ampl: model nqueens.mod;
ampl: let n := 20;
ampl: option solver gecode;
ampl: solve;
gecode 3.7.3: feasible solution
220 nodes, 105 fails
ampl: display Row;
Row [*] :=
 1   1
 2   3
 3   5
 4   7
 5  14
 6  10
  ...
19  15
20  13
;
```

LINKS

- GitHub repository: **https://github.com/vitaut/ampl**
- C++ Solver API: solvers/util
- Gecode AMPL solver: solvers/gecode
- IlogCP AMPL solver: solvers/ilogcp