We address a recent trend in **modeling systems** for **optimization** and **constraint satisfaction**:

- **High-level models** invoke **multiple solvers**.
- Models are **flattened** to low-level models for individual solvers.

**Thesis:** **Semantically typed logic models** are well suited to this task.

- Variable declarations become **relational database queries**.
Logic is deeply connected to optimization and constraint satisfaction. For example:

- **Optimization duals** are **logical inference** problems.
- The **resolution method** of logical inference is a special case of **cutting planes** in combinatorial optimization.
- Constraint satisfaction problems are often formulated directly as **SAT problems**.
- **Conflict-driven clause learning** for SAT is a special case of **Benders decomposition**.
- **BDDs** provide basis for **discrete optimization** (relaxation, primal heuristics, constraint propagation, postoptimality).
Prelude: Logic in Optimization

• Boole’s **probability logic** poses an optimization problem (linear programming) that can be solved with **column generation**.

• Inference in **belief logics**, **nonmonotonic logics**, etc., can be formulated as **linear and integer programming** problems.

• **Infinite-dimensional integer programming** is based on a compactness theorem equivalent to **Herbrand’s theorem** in 1st order logic.

• **Bayesian logic** can be solved with **nonlinear programming**.

• **Logic models** can provide high-level formulations of optimization problems.
• Boole’s probability logic poses an optimization problem (linear programming) that can be solved with column generation.

• Inference in belief logics, nonmonotonic logics, etc., can be formulated as linear and integer programming problems.

• Infinite-dimensional integer programming is based on a compactness theorem equivalent to Herbrand’s theorem in 1st order logic.

• Bayesian logic can be solved with nonlinear programming.

• Logic models can provide high-level formulations of optimization problems.

Today’s topic
Prelude: Logic in Optimization

• A constraint satisfaction problem \( P(x) \) is the logic problem of finding a model (in the logical sense) for

\[
\exists x P(x)
\]

• An optimization problem \( \min \{ f(x) \mid P(x) \} \) is the logic problem of finding a model (in the logical sense) for

\[
\exists x \forall y \left[ P(x) \land \left( P(y) \rightarrow (f(y) \geq f(x)) \right) \right]
\]
Basic Problem

• Write a high-level model that:
  
  • Invokes multiple solvers to exploit special structure in the problem.
  
  • Consists of high-level metaconstraints that convey special structure to the flattening process.
Basic Problem

• Write a high-level model that:
  • Invokes multiple solvers to exploit special structure in the problem.
  • Consists of high-level metaconstraints that convey special structure to the flattening process.
• But metaconstraint processing introduces new variables.
  • This poses a fundamental problem of variable management.
  • How to solve it?
Basic Problem

• Write a high-level model that:
  •_invokes multiple solvers to exploit special structure in the problem.
  • Consists of high-level metaconstraints that convey special structure to the flattening process.
• But metaconstraint processing introduces new variables.
  • This poses a fundamental problem of variable management.
  • How to solve it?
• Treat variable declarations are database queries.
  • In a logic with semantic typing.
Why Exploit Problem Structure?

• You can’t solve hard problems without exploiting special structure (No Free Lunch Theorem).
Why Exploit Problem Structure?

• You can’t solve hard problems without exploiting special structure (**No Free Lunch** Theorem).

• For SAT solvers:
  
  • Efficient encoding of problem in SAT form
Why Exploit Problem Structure?

• You can’t solve hard problems without exploiting special structure (No Free Lunch Theorem).

• For SAT solvers:
  • Efficient encoding of problem in SAT form

• For CP (constraint programming) solvers:
  • Careful choice of global constraints
  • Redundant constraints, search strategy, etc.
Why Exploit Problem Structure?

• You can’t solve hard problems without exploiting special structure (No Free Lunch Theorem).

• For SAT solvers:
  • Efficient encoding of problem in SAT form

• For CP (constraint programming) solvers:
  • Careful choice of global constraints
  • Redundant constraints, search strategy, etc.

• For MIP (mixed integer programming) solvers:
  • Careful choice of variables for tight formulation
  • Addition of valid inequalities
Conveying structure to the solver(s)

• Formulate problem with **global constraints** or **metaconstraints** to reveal structure.

• Automatically **flatten** the model in a way that best allows specific solvers to exploit structure:
  
  • Best choice of **variables**.
  
  • Reformulation of **constraints**.
    
    – For **effective propagation** or **tight relaxation**

  • Best choice of **domain filters**.

  • Generation of **valid inequalities**
Conveying structure to the solver(s)

- Formulate problem with **global constraints** or **metaconstraints** to reveal structure
- Automatically **flatten** the model in a way that best allows specific solvers to exploit structure:
  - Best choice of **variables**.
  - Reformulation of **constraints**.
    - For **effective propagation** or **tight relaxation**
  - Best choice of **domain filters**.
  - Generation of **valid inequalities**
- However, metaconstraints pose a fundamental problem of **variable management**...
Variable management problem

• Reformulation typically introduces **new variables**
  • Different metaconstraints may introduce variables that are functionally **the same variable**
  • …or **related** in some other way.
• **Recognizing these relationships** is essential to obtaining a good model (e.g., a tight continuous relaxation)
• How can the solver “understand” what is going on in the model?
Variable management problem

- **Example:** Let $x_j$ = worker assigned to job $j$
  
  $c_{ji} = \text{cost of assigning worker } i \text{ to job } j$

  Find **min-cost assignment**:

  $$\min \sum_j c_{x,j}$$

  alldiff ($x_1, \ldots, x_n$)

  Where metaconstraint \textbf{alldiff} = all variables take different values
Variable management problem

Find min-cost assignment:

$$\min \sum_j c_{x,j}$$

$$\text{alldiff}(x_1, \ldots, x_n)$$

This should be flattened to a classical assignment problem, which can be solved very rapidly by a specialized solver.

Let binary variable $y_{ij} = 1$ if worker $i$ is assigned to job $j$

$$\min \sum_{ij} c_{ij} y_{ij}$$

$$\sum_j y_{ij} = 1, \text{ all } i; \quad \sum_i y_{ij} = 1, \text{ all } j; \quad y_{ij} \in \{0,1\}$$
Variable management problem

Find min-cost assignment:

$$\min \sum_{j} c_{x,j}$$

$$\text{alldiff}(x_1, \ldots, x_n)$$

Objective function is automatically reformulated with 0-1 variables:

$$\min \sum_{ij} c_{ij} y_{ij} \quad \text{where} \quad x_j = \sum_{i} i y_{ij}$$
Variable management problem

Find **min-cost assignment**: 

$$
\min \sum_{j} c_{x,j}
$$

$$
\text{alldiff} \left( x_1, \ldots, x_n \right)
$$

Objective function is automatically reformulated with 0-1 variables: 

$$
\min \sum_{ij} c_{ij} y_{ij}
$$

where 

$$
x_j = \sum_i i y_{ij}
$$

Alldiff constraint is automatically reformulated with 0-1 variables: 

$$
\sum_i y'_{ij} = 1, \text{ all } j; \quad \sum_j y'_{ij} = 1, \text{ all } i
$$
Variable management problem

Find min-cost assignment:

$$\min \sum_j c_{jx_j}$$

$$\text{alldiff}(x_1, \ldots, x_n)$$

Objective function is automatically reformulated with 0-1 variables:

$$\min \sum_{ij} c_{ij} y_{ij} \quad \text{where} \quad x_j = \sum_i i y_{ij}$$

Alldiff constraint is automatically reformulated with 0-1 variables:

$$\sum_i y_{ij} = 1, \text{ all } j; \quad \sum_j y_{ij} = 1, \text{ all } i$$

How does the solver know that we want $y_{ij} = y_{ij}'$, allowing the problem to be solved rapidly as a classical assignment problem?

Declare variables with semantic typing.
Semantic typing

- We assume that all variables are declared.
Semantic typing

• We assume that all variables are **declared**.

• **Semantic typing** assigns a different meaning to each variable…

  • By associating the variable with a multi-place **predicate** and **keyword**.

  • The keyword “**queries**” the relation denoted by the predicate, as one queries a **relational database**.
Semantic typing

- We assume that all variables are **declared**.

- **Semantic typing** assigns a different meaning to each variable…
  - By associating the variable with a multi-place **predicate** and **keyword**.
  - The keyword “**queries**” the relation denoted by the predicate, as one queries a **relational database**.

- Advantage:
  - This allows the solver to **deduce relationships** between variables associated with the same predicate.
  - Can automatically add **channeling constraints**.
  - It is also **good modeling practice**.
How variables are introduced

• A model may include two formulations of the problem that use related variables.
  • Common in CP, because it strengthens propagation.
How variables are introduced

- A model may include **two formulations** of the problem that use related variables.
  - Common in CP, because it strengthens **propagation**.
  - For example,
    \[
    x_i = \text{job assigned to worker } i \\
    y_j = \text{worker assigned to job } j
    \]
  - Solver should generate **channeling constraints** to relate the variables to each other:
    \[
    j = x_{y_j}, \quad i = y_{x_i}
    \]
How variables are introduced

• The solver may reformulate a disjunction of linear systems

\[ \bigcup_k A_k x \geq b^k \]

using a convex hull (or big-M) formulation:

\[
A_k x^k \geq b^k y_k, \text{ all } k \\
x = \sum_k x^k, \quad \sum_k y_k = 1 \\
y_k \in \{0,1\}, \text{ all } k
\]

• Other constraints may be based on same set of alternatives, and corresponding auxiliary variables (\(y_k\) etc.) should be equated.
How variables are introduced

• A nonlinear or global solver may use **McCormick factorization** to replace nonlinear subexpressions with auxiliary variables
  
  • … to obtain a linear relaxation.
How variables are introduced

• A nonlinear or global solver may use **McCormick factorization** to replace nonlinear subexpressions with auxiliary variables

  • … to obtain a linear relaxation.

  • For example, bilinear term $xy$ can be linearized by replacing it with new variable $z$ and constraints

    \[
    L_y x + L_x y - L_x L_y \leq z \leq L_y x + U_x y - L_x U_y
    \]

    \[
    U_y x + U_x y - U_x U_y \leq z \leq U_y x + L_x y - U_x L_y
    \]

    where $x \in [L_x, U_x], \ y \in [L_y, U_y]$

  • Factorization of different constraints may create variables for identical subexpressions.

  • They should be identified to get a tight relaxation.
How variables are introduced

• The solver may reformulate different global constraints from CP by introducing variables that have the same meaning.
How variables are introduced

• The solver may reformulate different **global constraints** from CP by introducing variables that have the same meaning.

• For example, **sequence** constraint limits how many jobs of a given type can occur in given time interval:

\[
\text{sequence}(x), \quad x_i = \text{job in position } i
\]

and **cardinality** constraint limits how many times a given job appears

\[
\text{cardinality}(x), \quad x_j = \text{job in position } j
\]

Both may introduce variables

\[
y_{ij} = 1 \text{ when job } j \text{ occurs in position } i
\]

that should be identified.
How variables are introduced

• The solver may introduce equivalent variables while interpreting metaconstraints designed for **classical MIP modeling situations:**
  
  • Fixed-charge network flow
  • Facility location
  • Lot sizing
  • Job shop scheduling
  • Assignment (3-dim, quadratic, etc.)
  • Piecewise linear
Motivating example

• Allocate 10 advertising spots to 5 products

\[ x_i = \text{how many spots allocated to product } i \]

\[ y_{ij} = 1 \text{ if } j \text{ spots allocated to product } i \]
Motivating example

- Allocate 10 advertising spots to 5 products

\[ x_i = \text{how many spots allocated to product } i \]
\[ y_{ij} = 1 \text{ if } j \text{ spots allocated to product } i \]

\[ \leq 4 \text{ spots per product} \]
Motivating example

• Allocate 10 advertising spots to 5 products

\[ x_i = \text{how many spots allocated to product } i \]
\[ y_{ij} = 1 \text{ if } j \text{ spots allocated to product } i \]

\[ \leq 4 \text{ spots per product} \]
\[ \text{Advertise } \leq 3 \text{ products} \]
Motivating example

- Allocate 10 advertising spots to 5 products

\[ x_i = \text{how many spots allocated to product } i \]
\[ y_{ij} = 1 \text{ if } j \text{ spots allocated to product } i \]

\( \leq 4 \text{ spots per product} \)

Advertise \( \leq 3 \text{ products} \)

\( \geq 4 \text{ spots for at least one product} \)
Motivating example

• Allocate 10 advertising spots to 5 products

\[ x_i = \text{how many spots allocated to product } i \]

\[ y_{ij} = 1 \text{ if } j \text{ spots allocated to product } i \]

\[ \leq 4 \text{ spots per product} \]

Advertise \( \leq 3 \) products

\[ \geq 4 \text{ spots for at least one product} \]

\[ P_{ij} = \text{profit from allocating } j \text{ spots to product } i \]

Objective: maximize profit
Motivating example

- spots in \{0..4\}
- product in \{A,B,C,D,E\}

Index sets
Motivating example

spots in \{0..4\}
product in \{A,B,C,D,E\}

\texttt{data P\{product,spots\}}
Motivating example

spots in {0..4}
product in {A,B,C,D,E}
data P{product,spots}
x[i] is how many spots allocate(product i)
Motivating example

spots in \{0..4\}
product in \{A,B,C,D,E\}
data P\{product,spots\}

\[ \text{x[i] is how many spots allocate(product i)} \]

This makes it a variable declaration
Motivating example

spots in \{0..4\}
product in \{A,B,C,D,E\}
data \text{P}\{\text{product,spots}\}
x[i] \text{ is } \text{howmany spots allocate}(\text{product i})

Declaration of variable \(x_i\)

This is the semantic type

\wedge
Motivating example

spots in \{0..4\}
product in \{A,B,C,D,E\}
data \ P\{product,spots\}
x[i] is \textbf{howmany} spots allocate(product i)

Indicates an integer quantity

Declaration of variable \(x_i\)

Other keywords:
howmuch
whether
Motivating example

spots in \{0..4\}
product in \{A,B,C,D,E\}
data P\{product, spots\}
x[i] is how many spots allocate(product i)

How many of what?
Motivating example

spots in \(0..4\)
product in \{A, B, C, D, E\}
data \(P\{\text{product}, \text{spots}\}\)
\(x[i] \text{ is how many spots \(\text{allocate}(\text{product } i)\)}\)

Declaration of variable \(x_i\)

2-place predicate associated with variable \(x\)

Every variable is associated with a predicate that gives it meaning
Motivating example

spots in \{0..4\}
product in \{A,B,C,D,E\}
data P\{product, spots\}
\(x[i] \) is how many spots allocate \(\text{allocate}(product \ i)\)

Declaration of variable \(x_i\)

Other term of the predicate
Motivating example

spots in \{0..4\}
product in \{A,B,C,D,E\}
\textit{data} P\{product, spots\}
\( x[i] \) \textit{is how many spots allocate(product}\[i\]

Declaration of variable \( x_i \)

Associates
index of \( x[i] \) with
index set \textit{product}
Motivating example

\[ \max \sum_{i} P_{ix_i} \]

spots in \( \{0..4\} \)
product in \( \{A,B,C,D,E\} \)
data \( P\{product,spots\} \)
x[i] is how many spots allocate(product i)

maximize sum{product i} \( P[i,x[i]] \)  Objective function
Motivating example

spots in \{0..4\}
product in \{A,B,C,D,E\}
data \(P\{\text{product}, \text{spots}\}\)
x\[i\] is how many spots allocate(product \(i\))
maximize sum \{product \(i\}\} \(P[i,x[i]]\)
sum \{product \(i\}\} x[i] \leq 10 \quad 10 \text{ spots available}
Motivating example

\[
\begin{align*}
\text{maximize} & \quad \sum_i P_{ix_i} \\
\text{subject to} & \quad \sum_i x_i \leq 10
\end{align*}
\]

spots in \{0..4\}
product in \{A,B,C,D,E\}
data \textit{P\{product,spots\}}
x[i] is how many spots allocate(product \textit{i})
maximize sum\{product \textit{i}\} \textit{P[I,x[i]]}
sum\{product \textit{i}\} x[i] \leq 10
\textit{y[i,j]} is \underline{whether} allocate(product \textit{i}, spots \textit{j})

\underline{Indicates 0-1 variable}

\underline{Declare \textit{y}_{ij}}
Motivating example

\begin{align*}
\text{spots in } \{0..4\} \\
\text{product in } \{A,B,C,D,E\} \\
data \ P\{\text{product, spots}\} \\
x[i] \text{ is how many spots allocate(} \text{product } i \text{)} \\
\text{maximize } \sum \{\text{product } i\} \ P[i, x[i]] \\
\text{sum}\{\text{product } i\} \ x[i] \leq 10 \\
y[i,j] \text{ is whether allocate(} \text{product } i, \text{ spots } j\) \\
\text{Declare } y_{ij}
\end{align*}

\text{Associated with same predicate as } x[i]
Motivating example

\[ \text{max } \sum_i P_{ix_i} \]
\[ \sum_i x_i \leq 10, \quad \sum_i y_{i0} \geq 2 \]

spots in \{0..4\}
product in \{A,B,C,D,E\}
data \text{ P\{product,spots\}}
x[i] is how many spots allocate(product i)
maximize sum\{product i\} \text{ P}[i,x[i]]
\text{sum\{product i\} x[i] <= 10}
y[i,j] is whether allocate(product i, spots j)
\text{sum\{product i\} y[i,0] >= 2}  \quad \text{At least 2 products not advertised}
Motivating example

\[
\max \sum_i P_{ix_i} \\
\sum_i x_i \leq 10, \ \sum_i y_{i0} \geq 2, \ \sum_i y_{i4} \geq 1
\]

spots in \{0..4\}
product in \{A,B,C,D,E\}
data \(P\{\text{product,spots}\}\)
x[i] is how many spots allocate \(\text{allocate}(\text{product } i)\)
maximize \(\text{sum}\{\text{product } i\} \ P[i, x[i]]\)
\(\text{sum}\{\text{product } i\} \ x[i] \leq 10\)
y[i,j] is whether \(\text{allocate}(\text{product } i, \text{spots } j)\)
\(\text{sum}\{\text{product } i\} \ y[i,0] \geq 2\)
\(\text{sum}\{\text{product } i\} \ y[i,4] \geq 1\) At least 1 product gets \(\geq 4\) spots
Motivating example

spots in \{0..4\}
product in \{A,B,C,D,E\}
data \(P\{\text{product,spots}\}\)

\(x[i]\) is how many spots allocate \(\text{allocate}(\text{product } i)\)
maximize \(\sum \{\text{product } i\} \ P[i, x[i]]\)
\(\sum \{\text{product } i\} \ x[i] \leq 10\)

\(y[i,j]\) is whether allocate \(\text{allocate}(\text{product } i, \text{spots } j)\)
\(\sum \{\text{product } i\} \ y[i,0] \geq 2\)
\(\sum \{\text{product } i\} \ y[i,4] \geq 1\)

\(\sum \{\text{product } i\} \ \sum \{\text{spots } j\} \ y[i,j] = 1\)
\(\sum \{\text{product } i\} \ x[i] = \sum \{\text{spots } j\} \ j \cdot y[i,j]\)

Solver generates linking constraints because
\(x[i]\) and \(y[i,j]\) are associated with the same predicate.
Motivating example

spots in 

product in 

data \( P\{\text{product, spots}\} \)

\( x[i] \) is how many spots allocate (product i)

maximize \( \sum \{\text{product } i\} \ P[i, x[i]] \)

The objective function must be linearized. Solver generates

\[
\begin{align*}
\text{max } & \sum_{i} z_i \\
\sum_{i} x_i & \leq 10, \quad \sum_{i} y_{i0} \geq 2, \quad \sum_{i} y_{i4} \geq 1 \\
\sum_{j} y_{ij} & = 1, \quad x_i = \sum_{j} jy_{ij}, \quad \text{all } i
\end{align*}
\]

\( z_i = \sum_{j=0}^{4} P_{ij} y'_{ij}, \quad \sum_{j=0}^{4} y'_{ij} = 1, \quad x_i = \sum_{j=0}^{4} jy'_{ij}, \quad \text{all } i \)

\( y'[i,j] \) is whether allocate(product i, spots j)
Motivating example

spots in \{0..4\}
product in \{A,B,C,D,E\}
data \text{P}\{\text{product, spots}\}
\text{x}[i] \text{ is how many spots allocate (product i)}

maximize \sum \text{sum}\{\text{product i} \} \text{ P}[i, \text{x}[i]]

The objective function must be linearized. Solver generates

\[ z_i = \sum_{j=0}^{4} P_{ij} \text{y'}[ij], \sum_{j=0}^{4} \text{y'}[ij] = 1, x_i = \sum_{j=0}^{4} j\text{y'}[ij], \text{all i} \]

\text{y'}[i,j] \text{ is whether allocate (product i, spots j)}
\text{y} \text{ and } \text{y'} \text{ are identified because they have the same type:}
\text{y}[i,j] \text{ is whether allocate (product i, spots j)}
Predicates and relations

Predicate **allocate** denotes 2-place **relation** (set of tuples). Schematically indicated by:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>product</strong></td>
<td><strong>spots</strong></td>
</tr>
<tr>
<td>$i$</td>
<td>$X_i$</td>
</tr>
</tbody>
</table>
Predicates and relations

Predicate **allocate** denotes 2-place **relation** (set of tuples). Schematically indicated by:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
<td>spots</td>
</tr>
<tr>
<td>$i$</td>
<td>$x_i$</td>
</tr>
</tbody>
</table>

Column corresponding to a variable must be a **function** of other columns.
Predicates and relations

Predicate allocate denotes 2-place relation (set of tuples). Schematically indicated by:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
<td>spots</td>
</tr>
<tr>
<td>i</td>
<td>$X_i$</td>
</tr>
</tbody>
</table>

Declaration of $x[i]$ as

- howmany spots allocate (product i)

and $y[i,j]$ as

- whether allocate (product i, spots j)

query the relation for how many and whether.
Predicates and relations

Predicate **allocate** denotes 2-place **relation** (set of tuples). Schematically indicated by:

\[
\begin{array}{|c|c|}
\hline
1 & 2 \\
\hline
\text{product} & \text{spots} \\
\hline
i & x_i \\
\hline
\end{array}
\]

Declaration of \( x[i] \) as
- howmany spots allocate (product \( i \))
and \( y[i,j] \) as
- whether allocate (product \( i \), spots \( j \))

**query** the relation for how many and whether.

In general, **keywords** are **queries** (analogous to **relational database**).
Predicates and relations

Relation table reveals channeling constraints. For example,

\[ x[i] \text{ is which job assign(worker } i) \]
\[ y[j] \text{ is which worker assign(job } i) \]

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>job</td>
<td>worker</td>
</tr>
<tr>
<td>(j, x_i)</td>
<td>(i, y_j)</td>
</tr>
</tbody>
</table>

We can read off the channeling constraints

\[ j = x_i = x_{y_i} \]
\[ i = y_j = y_{x_i} \]
Predicates and relations

If several jobs can be assigned to a worker, we declare

\[ z[i] \text{ is whichset job assign(worker i)} \]

The channeling constraints are

\[ j \in Z_{y_j} \]
Previous work

- **Model management** uses semantic typing to help combine models and use inheritance.
  - Originally inspired by object-oriented programming
    Bradley & Clemence (1988)
  - *Quiddity*: a rigorous attempt to analyze conditions for variable identification
    Bhargava, Kimbrough & Krishnan (1991)
  - **SML** uses typing in a structured modeling framework
    Geoffrion (1992)
  - **Ascend** uses strongly-typed, object-oriented modeling
    Bhargava, Krishnan & Piela (1998)
Previous work

• Our semantic typing differs:

  • **Less ambitious** because it doesn’t attempt model management.
    • There is only one model.

  • **More ambitious** because we recognize relationships other than equivalence.

• We manage variables *introduced by solver*. 
Previous work

• Modeling systems that convey some structure to solver:
  • CP modeling systems use global constraints.
  • AIMMS uses typed index sets.
  • MiniZinc reformulates metaconstraints for specific solvers.
  • Savile Row uses common subexpression elimination.
  • OPL, Xpress-Kalis, Comet, etc., use interval variables.
  • SAT solver SymChaff uses high-level AI planning language PDDL.
  • SIMPL has full metaconstraint capability.
Previous work

• However, none of these systems deals systematically with the variable management problem.
  
  • We address it with semantic typing of variables.
Assignment problem

worker in \{1..m\}
job in \{1..n\}
data \(C\{\text{worker},\text{job}\}\)
\(x[j]\) is which worker assign(job j)
minimize sum\{job j\} \(C[x[j],j]\)
alldiff\{x[\ast]\}\)

\[
\min \sum_{j} c_{x,jj}
\]

alldiff \((x_1, \ldots, x_n)\)
Assignment problem

worker in \( \{1..m\} \)
job in \( \{1..n\} \)
data \( C\{\text{worker,job}\} \)
\( x[j] \) is which worker assign(jobs j)
minimize \( \sum \{ \text{worker j} \} \ C[x[j],j] \)
alldiff\{x[*]\}

Objective function is reformulated

\[
\max \sum_i c_{ij} y_{ij}, \quad x_i = \sum_j y_{ij}, \text{ all } i
\]

\( y[i,j] \) is whether assign(\text{worker i, job j})
Assignment problem

worker in \{1..m\}
job in \{1..n\}
data \text{C}\{\text{worker, job}\}
x[j] is which worker assign(job j)
minimize sum\{worker j\} \text{C}\{x[j], j\}
alldiff\{x[*]\}

Objective function is automatically reformulated

\[
\max \sum_{ij} c_{ij} y_{ij}, \quad x_i = \sum_j y_{ij}, \text{ all } i
\]

\[
y[i,j] \text{ is whether assign(worker i, job j)}
\]

Alldiff is automatically reformulated

\[
\sum_j y'_{ij} = 1, \text{ all } i, \quad \sum_i y'_{ij} = 1, \text{ all } j, \quad x_i = \sum_j jy'_{ij}, \text{ all } i
\]

\[
y'[i,j] \text{ is whether assign(worker i, job j)}
\]

Solver identifies \(y\) and \(y'\) to create classical AP.
Latin squares

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Numbers in every row and column are distinct. We will use **three** formulations to improve propagation.
Latin squares

Numbers in every row and column are distinct.
We will use three formulations to improve propagation.

row, col, num in \{1..n\}
x[i,j] is which num assign(row i, col j)
y[i,k] is which col assign(row i, num k)
z[j,k] is which row assign(col j, num k)

\[
\begin{array}{ccc}
2 & 3 & 1 \\
3 & 1 & 2 \\
1 & 2 & 3 \\
\end{array}
\]

\[
alldiff (x_{i1}, \ldots x_{in}), \text{ all } i \\
alldiff (x_{1j}, \ldots x_{nj}), \text{ all } j \\
alldiff (y_{i1}, \ldots y_{in}), \text{ all } i \\
alldiff (y_{1k} \ldots y_{nk}), \text{ all } k \\
alldiff (z_{j1}, \ldots x_{jn}), \text{ all } j \\
alldiff (z_{1k}, \ldots x_{nk}), \text{ all } k
\]
Latin squares

Numbers in every row and column are distinct.
We will use three formulations to improve propagation.

\[
\begin{array}{ccc}
2 & 3 & 1 \\
3 & 1 & 2 \\
1 & 2 & 3
\end{array}
\]

\[\text{row, col, num in } \{1..n\}\]
\[\text{x}[i,j] \text{ is which num assign(row i, col j)}\]
\[\text{y}[i,k] \text{ is which col assign(row i, num k)}\]
\[\text{z}[j,k] \text{ is which row assign(col j, num k)}\]
\{row i\} alldiff{x[i,*]}; \{col j\} alldiff{x[*,j]}
\{row i\} alldiff{y[i,*]}; \{num k\} alldiff{y[*,j]}
\{col j\} alldiff{z[j,*]}; \{num k\} alldiff{z[*,k]}

1 alldiff \(x_{i1}, \ldots x_{in}\), all i
1 alldiff \(x_{1j}, \ldots x_{nj}\), all j
1 alldiff \(y_{i1}, \ldots y_{in}\), all i
1 alldiff \(y_{1k} \ldots y_{nk}\), all k
1 alldiff \(z_{j1}, \ldots x_{jn}\), all j
1 alldiff \(z_{1k} \ldots x_{nk}\), all k
Latin squares

The predicate `assign` denotes the 3-place relation

```
row, col, num in {1..n}
x[i,j] is which num assign(row i, col j)
y[i,k] is which col assign(row i, num k)
z[j,k] is which row assign(col j, num k)
{row i} alldiff{x[i,*]}; {col j} alldiff{x[*],j})
{row i} alldiff{y[i,*]}; {num k} alldiff{y[*],j})
{col j} alldiff{z[j,*]}; {num k} alldiff{z[*],k})
```
Latin squares

The predicate \texttt{assign} denotes the 3-place relation

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\text{num} & \text{col} & \text{row} \\
k, x_{ij} & j, y_{ik} & i, z_{jk}
\end{array}
\]

We can read off the channeling constraints:

\[
k = x_{z_{jk} y_{ik}}, \quad j = y_{z_{jk} x_{ij}}, \quad i = z_{y_{ik} x_{ij}}, \quad \text{all } i, j, k
\]

which can be propagated.
Latin squares

The 3 formulations generate 3 identical MIP models:

\[
\begin{align*}
    x_{ij} &= \sum_k k \delta_{ijk}^x, \quad \sum_k \delta_{ijk}^x = 1, \text{ all } i, j; \\
    y_{ik} &= \sum_j j \delta_{ijk}^y, \quad \sum_j \delta_{ijk}^y = 1, \text{ all } i, k; \\
    z_{jk} &= \sum_i i \delta_{ijk}^z, \quad \sum_i \delta_{ijk}^z = 1, \text{ all } j, k
\end{align*}
\]
Latin squares

\{row i\} \textit{alldiff}\{x[i,*]\}; \{col j\} \textit{alldiff}\{x[*\,j]\}
\{row i\} \textit{alldiff}\{y[i,*]\}; \{num k\} \textit{alldiff}\{y[*\,j]\}
\{col j\} \textit{alldiff}\{z[j,*]\}; \{num k\} \textit{alldiff}\{z[*\,k]\}

The 3 formulations generate 3 identical MIP models:

\begin{align*}
x_{ij} &= \sum_k k \delta_{ijk}^x; \quad \sum_k \delta_{ijk}^x = 1, \text{ all } i, j; \quad \sum_j \delta_{ijk}^x = 1, \text{ all } i, k; \quad \sum_i \delta_{ijk}^x = 1, \text{ all } j, k \\
y_{ik} &= \sum_j j \delta_{ijk}^y; \quad \sum_j \delta_{ijk}^y = 1, \text{ all } i, k; \quad \sum_k \delta_{ijk}^y = 1, \text{ all } i, j; \quad \sum_i \delta_{ijk}^y = 1, \text{ all } j, k \\
z_{jk} &= \sum_i i \delta_{ijk}^z; \quad \sum_i \delta_{ijk}^z = 1, \text{ all } j, k; \quad \sum_k \delta_{ijk}^z = 1, \text{ all } i, j; \quad \sum_j \delta_{ijk}^z = 1, \text{ all } i, k
\end{align*}

The solver declares \( \delta_{ijk}^x, \delta_{ijk}^y, \delta_{ijk}^z \)

\textbf{whether assign}(\text{row } i, \text{ col } j, \text{ num } k)

So it treats them as the same variable and generates only 1 MIP model.
Multiple *which* variables

In general, an $n$-place predicate that denotes the relation

<table>
<thead>
<tr>
<th>1</th>
<th>...</th>
<th>$k$</th>
<th>$k+1$</th>
<th>...</th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$term_1$</td>
<td>...</td>
<td>$term_k$</td>
<td>$term_{k+1}$</td>
<td>...</td>
<td>$term_n$</td>
</tr>
<tr>
<td>$i_1, x^i_{i(1)}$</td>
<td>...</td>
<td>$i_k, x^i_{i(k)}$</td>
<td>$i_{k+1}$</td>
<td>...</td>
<td>$i_n$</td>
</tr>
</tbody>
</table>

for *which* variables, where $i(j) = i_1 \cdots i_{j-1} i_{j+1} \cdots i_n$

generates the channeling constraints

$$i_j = x^j_{i(1)} \cdots x^j_{i(j-1)} x^j_{i(j+1)} \cdots x^j_{i(k)} i_{k+1} \cdots i_n,$$
all $i_1, \ldots, i_n$, $j = 1, \ldots, k$
Multiple whether variables

whether keywords serve as projection operators on the relation.

\[ y[i,j,d] \text{ is } \text{whether assign}(\text{worker } i, \text{ job } j, \text{ day } d) \]

Project out \( d \):
\[ y_1[i,j] \text{ is } \text{whether assign}(\text{worker } i, \text{ job } j) \]

Project out \( j \) and \( d \):
\[ y_2[i] \text{ is } \text{whether assign}(\text{worker } i) \]
Short forms

Declare $x_i$ to be cost of activity $i$:

$x[i]$ is *howmuch* cost(activity $i$)

which is short for the formal declaration

$x[i]$ is *howmuch* cost cost(activity $i$)

in which a new term *cost* is generated

Declare $x$ to be cost:

$x$ is *howmuch* cost

which is short for

$x$ is *howmuch* cost cost()
Piecewise linear function $z = f(x)$

Breakpoints in $A$, ordinates in $C$

- $x$ is how much output
- index in $\{1..n\}$
- data $A,C\{\text{index}\}$
- $z$ is how much cost
- \text{piecewise}(x,z,A,C)

This metaconstraint defines $z = f(x)$
Piecewise linear

Piecewise linear function \( z = f(x) \)
Breakpoints in \( A \), ordinates in \( C \)

\( x \) is how much output
index in \( \{1..n\} \)
data \( A,C\{\text{index}\} \)
z is how much cost
\( \text{piecewise}(x,z,A,C) \)

Solver generates the **locally ideal** model

\[
x = a_1 + \sum_{i=1}^{n-1} \bar{x}_i, \quad z = c_1 + \sum_{i=1}^{n-1} \frac{c_{i+1} - c_i}{a_{i+1} - a_i} \bar{x}_i
\]

\[(a_{i+1} - a_i)\delta_{i+1} \leq \bar{x}_i \leq (a_{i+1} - a_i)\delta_i, \quad \delta_i \in \{0,1\}, \quad i = 1,\ldots,n-1\]

We need to declare auxiliary variables \( \delta_i, x_i \)
Piecewise linear

Piecewise linear function $z = f(x)$
Breakpoints in $A$, ordinates in $C$

$x$ is howmuch output
index in $\{1..n\}$
data $A, C$\{index\}
$z$ is howmuch cost
\texttt{piecewise}(x, z, A, C)

\texttt{piecewise} constraint induces solver to declare a new index set that associates \texttt{index} with $A$, and use it to declare $\delta_i, x_i$

$xbar[i]$ is howmuch output.$A(\text{index } i)$
delta[i] is whether lastpositive output.$A(\text{index } i)$

Both declarations create predicates inherited from output and $A$
Piecewise linear

Suppose there is another piecewise function on the same break points

\( x \) is how much output
index in \( \{1..n\} \)
data \( A,C\{\text{index}\} \)
z is how much cost
\( \text{piecewise}(x,z,A,C) \)
data \( C'\{\text{index}\} \)
z' is how much profit
\( \text{piecewise}(x,z',A,C') \)
x'[i] is how much cost output \( A(\text{index } i) \)
\( \text{delta}'[i] \) is whether last positive output \( A(\text{index}) \)
Piecewise linear

Suppose there is another piecewise function on the same break points

- \( x \) is how much output index in \( \{1..n\} \)
- \( \text{data } A,C\{\text{index}\} \)
- \( z \) is how much cost
- \( \text{piecewise}(x,z,A,C) \)
- \( \text{data } C'\{\text{index}\} \)
- \( z' \) is how much profit
- \( \text{piecewise}(x,z',A,C') \)
- \( x'[i] \) is how much cost output.A(index i)
- \( \text{delta'}[i] \) is whether last positive output.A(index)

Because new piecewise constraint is associated with the same \( x \) and \( A \), solver again creates \text{output.A}. The solver creates variables \( \delta_i' \) and \( x_i' \) with same types as \( \delta_i \) and \( x_i \) and so identifies them.
Interval variables

cumulative\((x, D, R, L)\)

\(x_j \subseteq W_j, \text{ all } j\)

Each job \(j\) runs for a time interval \(x_j\).
We wish to schedule jobs so that total resource consumption
never exceeds \(L\).

\(\text{job in } \{1..n\}\)
\(\text{time in } \{t..T\}\)
\(\text{data } W, D, R\{\text{job}\} \text{ window, duration, resource}\)
\(\text{running in } [\text{time, time}] \text{ makes running an interval variable}\)
\(x[j] \text{ is when running } \text{sched(job j)} \text{ subset } W[j]\)
\(\text{cumulative}(x, D, R, L)\)
Interval variables

each job $j$ runs for a time interval $x_j$.
we wish to schedule jobs so that total resource consumption never exceeds $L$.

```
job in {1..n}
time in {t..T}
data W,D,R{job} window, duration, resource
running in [time,time] makes running an interval variable
x[j] is when running sched(job j) subset W[j]
cumulative(x,D,R,L)
```

solver generates the model

$$\sum_t \delta_{jt} = 1, \text{ all } j; \sum_j R_j \phi_{jt} \leq L, \text{ all } t$$

$$\phi_{jt} \geq \delta_{jt'}, \text{ all } t,t' \text{ with } 0 \leq t - t' < D_j, \text{ all } j$$

delta[j,t] is whether running.start sched(job j, time t)
phi[j,t] is whether running sched(job j, time t)
Interval variables

Suppose we want finish times to be separated by at least $T_0$

job in {1..n}
time in {t..T}
data W,D,R{job}
running in [time,time]
x[j] is when running sched(job j) subset W[j]
cumulative($x, D, R, L$)

{$job j, job k | j<>k$} $|x[j].end - x[k].end| >= T_0$
delta[j,t] is whether running.start sched(job j, time t)
phi[j,t] is whether running sched(job j, time t)
Interval variables

Suppose we want finish times to be separated by at least $T_0$

job in $\{1..n\}$
time in $\{t..T\}$
data $W,D,R\{job\}$
running in $[\text{time, time}]$

$x[j]$ is when running $\text{sched(job j)}$ subset $W[j]$
cumulative$(x, D, R, L)$

$\{\text{job } j, \text{ job } k \mid j \neq k\}$ $|x[j].\text{end} - x[k].\text{end}| \geq T_0$
delta$[j,t]$ is whether running $\text{start}$ $\text{sched(job j, time t)}$
phi$[j,t]$ is whether running $\text{sched(job j, time t)}$

Solver generates

$\varepsilon_{jt} + \varepsilon_{kt'} \leq 1$, all $t, t'$ with $0 < t' - t < T_0$, all $j, t$ with $j \neq k$
epsilon$[j,t]$ is whether running $\text{end}$ $\text{sched(job j, time t)}$
Interval variables

Variables $\delta_{jt}$ and $\varepsilon_{jt}$ are related by an offset. Solver associates `running.end` in declaration of $\varepsilon_{jt}$ with `running.start` in declaration of $\delta_{jt}$ and deduces

$$e_{j,t+D_j} = \delta_{jt}, \text{ all } j,t$$

`delta[j,t]` is whether `running.start` sched(job j, time t)

`phi[j,t]` is whether `running` sched(job j, time t)

`epsilon[j,t]` is whether `running.end` sched(job j, time t)
Interval variables

Variables $\delta_{jt}$ and $\varepsilon_{jt}$ are related by an offset. Solver associates `running.end` in declaration of $\varepsilon_{jt}$ with `running.start` in declaration of $\delta_{jt}$ and deduces

$$e_{j,t+D_j} = \delta_{jt}, \text{ all } j,t$$

Solver also associates `running.end` in declaration of $\varepsilon_{jt}$ with `running` in declaration of $\phi_{jt}$ and deduces the redundant constraints

$$\phi_{jt} \geq \varepsilon_{jt'}, \text{ all } t,t' \text{ with } 0 \leq t' - t < D_j, \text{ all } j$$

delta[j,t] is whether running.start sched(job j, time t)
phi[j,t] is whether running sched(job j, time t)
epsilon[j,t] is whether running.end sched(job j, time t)
TSP with Side Constraints

Traveling salesman problem with missing arcs and precedence constraints.

city, position in \{1..n\}
data \text{D}\{\text{city, city}\} \quad \text{Distances}
data \text{Prec}\{\text{city, city}\} \quad \text{Prec}_{i,j}=1 \text{ if } i \text{ must precede } j$
data \text{Succ}\{\text{city}\} \quad \text{Succ}[j] = \text{set of successors of city } j$

\[
\min \sum_{i} D_{is_i}
\]

\[
\text{alldiff}(x), \quad \text{circuit}(s)
\]

\[
x_i < x_j, \quad \text{all } i, j \text{ with } \text{prec}_{i,j} = 1
\]

\[
s_i \in \text{Succ}_i
\]
TSP with Side Constraints

Traveling salesman problem with missing arcs and precedence constraints.

city, position in \{1..n\}
data \(D\{\text{city, city}\}\) Distances
data \(\text{Prec}\{\text{city, city}\}\) \(\text{Prec}[i,j]=1\) if \(i\) must precede \(j\)
data \(\text{Succ}\{\text{city}\}\) \(\text{Succ}[j] =\) set of successors of city \(j\)

Two variable systems:
\[x[i] \text{ is which position ordering(city i)}\]
\[s[i] \text{ is successor city ordering(city i) subset Succ[i]}\]

\[
\min \sum_i D_{is_i} \\
\text{alldiff}(x), \text{circuit}(s) \\
x_i < x_j, \text{ all } i, j \text{ with } \text{prec}_{ij} = 1 \\
s_i \in \text{Succ}_i
\]
TSP with Side Constraints

Traveling salesman problem with missing arcs and precedence constraints.

city, position in \( \{1..n\} \)
data \( D\{\text{city, city}\} \) \quad \text{Distances}
data \( \text{Prec}\{\text{city, city}\} \) \quad \text{Prec}[i,j] = 1 \text{ if } i \text{ must precede } j
data \( \text{Succ}\{\text{city}\} \) \quad \text{Succ}[j] = \text{set of successors of city } j

Two variable systems:
\( x[i] \) \text{ is which position ordering(} \text{city } i \text{)}
\( s[i] \) \text{ is successor city ordering(} \text{city } i \text{) subset Succ}[i]

Precedence constraints require \( x \) variables
\( \text{prec}\{\text{city } i, \text{ city } j \mid \text{Prec}[i,j] = 1\} \): \( x[i] < x[j] \)
Missing arc constraints (implicit in data Succ) require \( s \) variables

\[
\min \sum_i D_{is_i}
\]
\[
\text{alldiff}(x), \text{ circuit}(s)
\]
\[
x_i < x_j, \text{ all } i, j \text{ with } \text{prec}_{ij} = 1
\]
\[
s_i \in \text{ Succ}_i
\]
TSP with Side Constraints

Traveling salesman problem with missing arcs and precedence constraints.

city, position in \{1..n\}
data \(D]\{city, city\} \quad \text{Distances}
data \(Prec]\{city, city\} \quad \text{Prec}[i,j]=1 \text{ if } i \text{ must precede } j$
data \(Succ]\{city\} \quad \text{Succ}[j] = \text{set of successors of city } j$

Two variable systems:
\(x[i] \text{ is which position ordering(city i)}\)
\(s[i] \text{ is successor city ordering(city i) subset Succ[i]}\)

Precedence constraints require \(x\) variables
\(prec\{city i, city j \mid Prec[i,j] = 1\}: x[i] < x[j]\)

Missing arc constraints (implicit in data Succ) require \(s\) variables

\[
\min \sum_i D_{is_i} \quad \text{alldiff}(x), \text{circuit}(s)
\]
x\(_i < x_j\), all \(i, j\) with \(prec\_{ij} = 1\)
s\(_i \in Succ_i\)

\[
\min \sum \{\text{city } i\} D[i,s[i]] \quad \text{Objective function}
\]
TSP with Side Constraints

The solver can give $\text{alldiff}(x)$ a conventional assignment model using $z_{ik} = \text{whether city } i \text{ is in position } k$.

$z[i,k]$ is whether ordering(city $i$, position $k$)
The solver can give \texttt{alldiff}(x) a conventional assignment model using $z_{ik} =$ whether city $i$ is in position $k$.

$z[i,k]$ is whether ordering(city $i$, position $k$)

For \texttt{circuit(s)}, the solver can introduce $w_{ij} =$ whether city $i$ immediately precedes city $j$.

$w[i,j]$ is whether successor ordering(city $i$, city $j$)
The solver can give $\text{alldiff}(x)$ a conventional assignment model using $z_{ik} =$ whether city $i$ is in position $k$.

$z[i,k]$ is whether ordering(city $i$, position $k$)

For circuit(s), the solver can introduce $w_{ij} =$ whether city $i$ immediately precedes city $j$.

$w[i,j]$ is whether successor ordering(city $i$, city $j$)

Declaration of $z$ tells solver that predicate is ordering(city,position), not ordering(city,city).
TSP with Side Constraints

The solver can give \texttt{alldiff}(x) a conventional assignment model using \( z_{ik} = \) whether city \( i \) is in position \( k \).

\( z[i,k] \) is whether ordering(city \( i \), position \( k \))

For \texttt{circuit(s)}, the solver can introduce
\( w_{ij} = \) whether city \( i \) immediately precedes city \( j \).

\( w[i,j] \) is whether successor ordering(city \( i \), city \( j \))

Declaration of \texttt{z} tells solver that predicate is \texttt{ordering(city,position)}, not \texttt{ordering(city,city)}.
Solver generates cutting planes in \texttt{w}-space and \texttt{s}-space.
The solver can give \texttt{alldiff}(x) a conventional assignment model using \( z_{ik} \) = whether city \( i \) is in position \( k \).

\( z[i,k] \) is whether \texttt{ordering(city \( i \), position \( k \))}

For \texttt{circuit(s)}, the solver can introduce \( w_{ij} \) = whether city \( i \) immediately precedes city \( j \).

\( w[i,j] \) is whether \texttt{successor ordering(city \( i \), city \( j \))}

Declaration of \( z \) tells solver that predicate is \texttt{ordering(city,position)}, not \texttt{ordering(city,city)}. Solver generates cutting planes in \( w \)-space and \( s \)-space.

The \texttt{successor} keyword tells solver how \( z \) and \( w \) relate.

\[ \phi_{jt} \geq \varepsilon_{jt'}, \text{ all } t, t' \text{ with } 0 \leq t' - t < D_j, \text{ all } j \]
TSP with Side Constraints

Suppose we also have constraints on which city is in position $k$. Simply declare

$$y[k] = \text{which city ordering(position k)}$$

The solver generates the channeling constraints between $y[k]$ and $x[i] = \text{which position is city } i$
Suppose we also have constraints on which city is in position \( k \). Simply declare

\[ y[k] = \text{which city ordering(position \( k \))} \]

The solver generates the channeling constraints between \( y[k] \) and \( x[i] = \text{which position is city} \ i \)

The solver can also introduce a second (equivalent) objective function

\[ \min \ \text{sum(position \( k \))} \ D[y[k], y[k+1]] \]

which may improve bounding.
Pros and Cons of Semantic Typing

• Pros

  • Conveys problem structure to the solver(s)
    – …by allowing use of metaconstaints
  • Incorporates state of the art in formulation, valid inequalities
  • Allows solver to expand repertory of techniques
    – Domain filtering, propagation, cutting plane algorithms
  • Good modeling practice
    – Self-documenting
    – Bug detection
Pros and Cons of Semantic Typing

• Cons
  • Modeler must be familiar with a library of metaconstraints
    – Rather than few primitive constraints
Pros and Cons of Semantic Typing

• Cons
  • Modeler must be familiar with a library of metaconstraints
    – Rather than few primitive constraints
  • Response
    – Modeler must be familiar with the underlying concepts anyway
    – Modeling system can offer sophisticated help, improve modeling
Pros and Cons of Semantic Typing

• Cons
  • Modeler must be familiar with a library of metaconstraints
    – Rather than few primitive constraints
  • Response
    – Modeler must be familiar with the underlying concepts anyway
    – Modeling system can offer sophisticated help, improve modeling
  • OR, SAT community is not accustomed to high-level modeling
    – Typed languages like Ascend never really caught on, resistance to CP.
Pros and Cons of Semantic Typing

• Cons
  • Modeler must be familiar with a library of metaconstraints
    – Rather than few primitive constraints
  • Response
    – Modeler must be familiar with the underlying concepts anyway
    – Modeling system can offer sophisticated help, improve modeling
  • OR, SAT community is not accustomed to high-level modeling
    – Typed languages like Ascend never really caught on, resistance to CP.
  • Response
    – Train the next generation!