A Modeling Language
Based on Semantic Typing

John Hooker
Carnegie Mellon University

Joint work with

André Ciré
University of Toronto

Tallys Yunes
University of Miami

Logic and Practice of Programming – LPOP 2018

Oxford, UK, July 2018

Logic Modeling for Optimization

- 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**.

Prelude: Logic in Optimization

- **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.

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.

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

$$\exists x \forall y \left[P(x) \wedge \left(P(y) \rightarrow (f(y) \geq f(x)) \right) \right]$$

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} = cost of assigning worker i to job j

Find **min-cost assignment**:

$$\min \sum_j c_{x_j j}$$

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

Where metaconstraint **alldiff** = all variables take different values

Variable management problem

Find **min-cost assignment**:

$$\min \sum_j c_{x_j j}$$

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

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**:

$$\min \sum_j c_{x_j j}$$

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

Objective function is automatically reformulated with 0-1 variables: $\min \sum_{ij} c_{ij} y_{ij}$ where $x_j = \sum_i i y_{ij}$

Variable management problem

Find **min-cost assignment**:

$$\min \sum_j c_{x_j j}$$

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

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: $\sum_i y'_{ij} = 1, \text{ all } j; \ \sum_j y'_{ij} = 1, \text{ all } i$

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

Alldiff constraint is automatically reformulated with 0-1 variables: $\sum_i y'_{ij} = 1, \text{ all } j; \; \sum_j y'_{ij} = 1, \text{ all } i$

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:

$$j = x_{y_j}, \qquad i = y_{x_i}$$

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:

$$\begin{aligned} & A_k x^k \geq b^k y_k, \quad \text{all } k \\ & x = \sum_k x^k, \quad \sum_k y_k = 1 \\ & y_k \in \{0,1\}, \quad \text{all } k \end{aligned}$$

- 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

$$L_y x + L_x y - L_x L_y \le z \le L_y x + U_x y - L_x U_y$$
$$U_y x + U_x y - U_x U_y \le z \le U_y x + L_x y - U_x L_y$$

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:

$$\text{sequence}(x),\ \ x_i = \text{ job in position } i$$

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

$$\text{cardinality}(x),\ \ x_j = \text{ job in position } j$$

Both may introduce variables

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

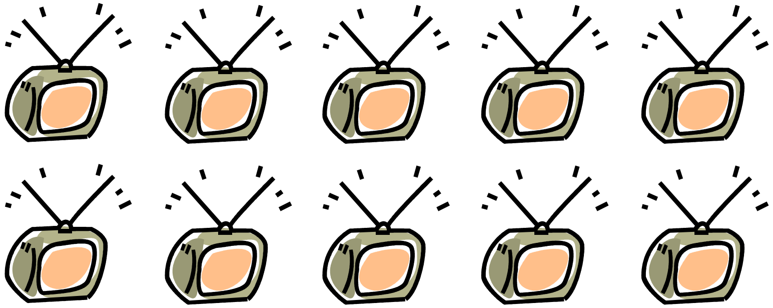
that should be identified.

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



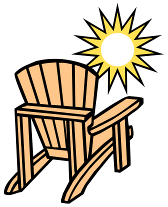
$x_i =$ how many spots allocated to product *i*

$y_{ij} =$ 1 if *j* spots allocated to product *i*

A

B

C

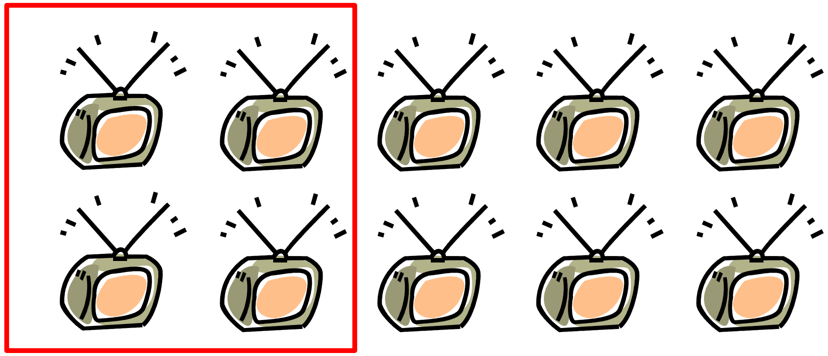


D

E

Motivating example

- Allocate 10 advertising spots to 5 products



$\le$ 4 spots per product

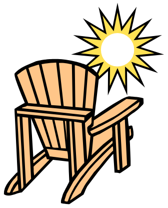
x_i = how many spots allocated to product *i*

y_{ij} = 1 if *j* spots allocated to product *i*

A

B

C

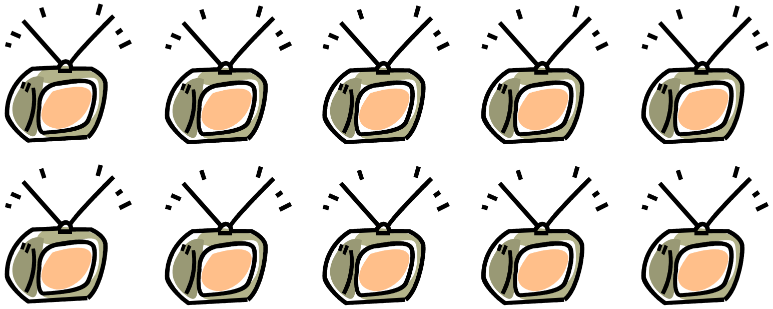


D

E

Motivating example

- Allocate 10 advertising spots to 5 products



$\leq$ 4 spots per product

Advertise $\leq$ 3 products

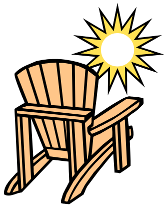
$x_i =$ how many spots allocated to product i

$y_{ij} =$ 1 if j spots allocated to product i

A

B

C

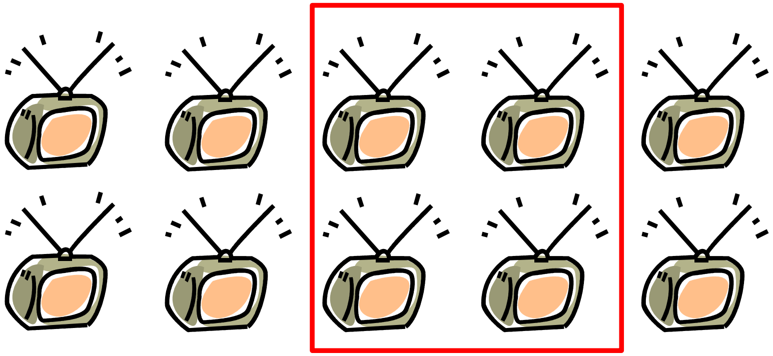


D

E

Motivating example

- Allocate 10 advertising spots to 5 products



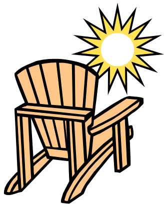
x_i = how many spots allocated to product i

y_{ij} = 1 if j spots allocated to product i

A

B

C



D

E

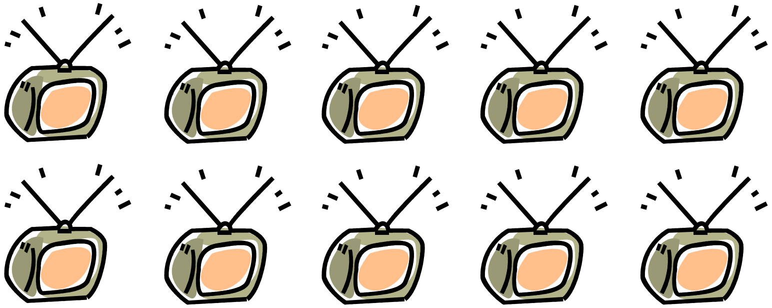
$\leq$ 4 spots per product

Advertise $\leq$ 3 products

$\geq$ 4 spots for at least one product

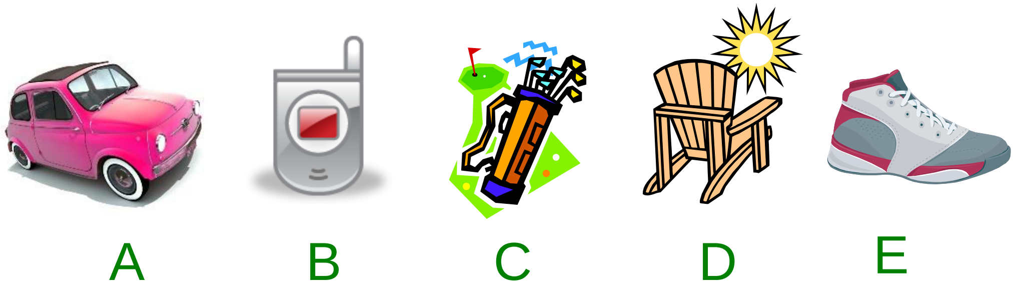
Motivating example

- Allocate 10 advertising spots to 5 products



$x_i =$ how many spots allocated to product i

$y_{ij} =$ 1 if j spots allocated to product i



$\leq$ 4 spots per product

Advertise $\leq$ 3 products

$\geq$ 4 spots for at least one product

P_{ij} = profit from allocating j 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}
data P{product,spots}
```

Data input

Motivating example

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

Declaration of variable x_i

Motivating example

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

Declaration of variable x_i

This makes it
a variable
declaration

∧

Motivating example

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

Declaration of variable x_i

This is the semantic type

^

Motivating example

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

Declaration of variable x_i

Indicates an integer quantity

Other keywords: ***howmuch*** ***whether***

∧

Motivating example

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

Declaration of variable x_i

How many of what?

∧

Motivating example

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

Declaration of variable x_i

Associates
index of `x[i]` with
index set `product`

Motivating example

$$\boxed{\max \sum_i P_{ix_i}}$$

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

Objective function

Motivating example

$$\max \sum_i P_{ix_i}$$

$$\boxed{\sum_i x_i \leq 10}$$

```
spots in {0..4}
product in {A,B,C,D,E}
data P{product,spots}
x[i] is howmany spots allocate(product i)
maximize sum{product i} P[i,x[i]]
sum{product i} x[i] <= 10
```

10 spots available

Motivating example

$$\max \sum_i P_{ix_i}$$

$$\sum_i x_i \leq 10$$

```
spots in {0..4}
product in {A,B,C,D,E}
data P{product,spots}
x[i] is howmany spots allocate(product i)
maximize sum{product i} P[I,x[i]]
sum{product i} x[i] <= 10
y[i,j] is whether allocate(product i, spots j)
```

Declare y_{ij}

Indicates 0-1 variable

Motivating example

$$\max \sum_i P_{ix_i}$$

$$\sum_i x_i \leq 10$$

```
spots in {0..4}
product in {A,B,C,D,E}
data P{product,spots}
x[i] is howmany spots allocate(product i)
maximize sum{product i} P[i,x[i]]
sum{product i} x[i] <= 10
y[i,j] is whether allocate(product i, spots j)
```

Declare y_{ij}

Associated with
same predicate
as **`x[i]`**

Motivating example

$$\max \sum_i P_{ix_i}$$

$$\sum_i x_i \leq 10, \boxed{\sum_i y_{i0} \geq 2}$$

```
spots in {0..4}
product in {A,B,C,D,E}
data P{product,spots}
x[i] is howmany spots allocate(product i)
maximize sum{product i} P[i,x[i]]
sum{product i} x[i] <= 10
y[i,j] is whether allocate(product i, spots j)
sum{product i} y[i,0] >= 2
```

At least 2 products not advertised

Motivating example

$$\max \sum_i P_{ix_i}$$

$$\sum_i x_i \le 10,\ \sum_i y_{i0} \ge 2,\ \boxed{\sum_i y_{i4} \ge 1}$$

```
spots in {0..4}
product in {A,B,C,D,E}
data P{product,spots}
x[i] is howmany spots allocate(product i)
maximize sum{product i} P[i,x[i]]
sum{product i} x[i] <= 10
y[i,j] is whether allocate(product i, spots j)
sum{product i} y[i,0] >= 2
sum{product i} y[i,4] >= 1
```

At least 1 product gets ≥4 spots

Motivating example

$$\max \sum_i P_{ix_i}$$

$$\sum_i x_i \le 10,\ \sum_i y_{i0} \ge 2,\ \sum_i y_{i4} \ge 1$$

$$\boxed{\sum_j y_{ij} = 1,\ x_i = \sum_j j y_{ij},\ \text{all } i}$$

```
spots in {0..4}
product in {A,B,C,D,E}
data P{product,spots}
x[i] is howmany spots allocate(product i)
maximize sum{product i} P[i,x[i]]
sum{product i} x[i] <= 10
y[i,j] is whether allocate(product i, spots j)
sum{product i} y[i,0] >= 2
sum{product i} y[i,4] >= 1
{product i} sum{spots j} y[i,j] = 1
{product i} x[i] = sum{spots j} j*y[i,j]
```

Solver generates linking constraints because
`x[i]` and **`y[i,j]`** are associated with the same predicate.

Motivating example

$$\boxed{\max \sum_i z_i}$$

$$\sum_i x_i \le 10,\ \sum_i y_{i0} \ge 2,\ \sum_i y_{i4} \ge 1$$

$$\sum_j y_{ij} = 1,\ x_i = \sum_j j y_{ij},\ \text{all } i$$

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

The objective function must be linearized. Solver generates

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

```
y'[i,j] is whether allocate(product i, spots j)
```

Motivating example

$$\boxed{\max \sum_i z_i}$$

$$\sum_i x_i \le 10,\ \sum_i y_{i0} \ge 2,\ \sum_i y_{i4} \ge 1$$

$$\sum_j y_{ij} = 1,\ x_i = \sum_j j y_{ij},\ \text{all } i$$

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

The objective function must be linearized. Solver generates

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

```
y′[i,j] is whether allocate(product i, spots j)
```

y and **y′** are identified because they have the same type:

```
y[i,j] is whether allocate(product i, spots j)
```

Predicates and relations

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

1	2
`product`	`spots`
i	x_i

Predicates and relations

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

1	2
`product`	**`spots`**
i	x_i

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:

1	2
product	spots
i	x_i

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:

1	2
`product`	`spots`
i	x_i

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] is which job assign(worker i)
y[j] is which worker assign(job i)
```

1	2
job	worker
j, x_i	i, y_j

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] 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

$$\min \sum_j c_{x_j j}$$

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

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

Assignment problem

$$\min \sum_j c_{x_j j}$$

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

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

Objective function
is reformulated

$$\max \sum_i c_{ij} y_{ij}, \; x_i = \sum_j y_{ij}, \text{ all } i$$

```
y[i,j] is whether assign(worker i, job j)
```

Assignment problem

$$\min \sum_j c_{x_j j}$$

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

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

Objective function is automatically reformulated

$$\max \sum_i c_{ij} y_{ij}, \; x_i = \sum_j y_{ij}, \text{ all } i$$

```
y[i,j] is whether assign(worker i, job j)
```

Alldiff is automatically reformulated

$$\sum_j y'_{ij} = 1, \text{ all } i, \; \sum_i y'_{ij} = 1, \text{ all } j, \; x_i = \sum_j j y'_{ij}, \text{ all } i$$

```
y′[i,j] is whether assign(worker i, job j)
```

Solver identifies *y* and *y′* to create classical AP.

Latin squares

		j	
	2	3	1
i	3	1	2
	1	2	3

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

Latin squares

j

i

2	3	1
3	1	2
1	2	3

$$\text{alldiff}\left(x_{i1},\ldots x_{in}\right),\ \text{all } i$$

$$\text{alldiff}\left(x_{1j},\ldots x_{nj}\right),\ \text{all } j$$

$$\text{alldiff}\left(y_{i1},\ldots y_{in}\right),\ \text{all } i$$

$$\text{alldiff}\left(y_{1k}\ldots y_{nk}\right),\ \text{all } k$$

$$\text{alldiff}\left(z_{j1},\ldots x_{jn}\right),\ \text{all } j$$

$$\text{alldiff}\left(z_{1k},\ldots x_{nk}\right),\ \text{all } k$$

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)
```

Latin squares

		j	
	2	3	1
i	3	1	2
	1	2	3

$$\text{alldiff}\left(x_{i1},\ldots x_{in}\right),\ \text{all}\ i$$

$$\text{alldiff}\left(x_{1j},\ldots x_{nj}\right),\ \text{all}\ j$$

$$\text{alldiff}\left(y_{i1},\ldots y_{in}\right),\ \text{all}\ i$$

$$\text{alldiff}\left(y_{1k}\ldots y_{nk}\right),\ \text{all}\ k$$

$$\text{alldiff}\left(z_{j1},\ldots x_{jn}\right),\ \text{all}\ j$$

$$\text{alldiff}\left(z_{1k},\ldots x_{nk}\right),\ \text{all}\ k$$

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)
{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 **`assign`** denotes the 3-place relation

1	2	3
`num`	`col`	`row`
k, x_{ij}	j, y_{ik}	i, z_{jk}

```
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 **`assign`** denotes the 3-place relation

1	2	3
`num`	**`col`**	**`row`**
k, x_{ij}	j, y_{ik}	i, z_{jk}

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

```
{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])
```

The 3 formulations generate 3 identical MIP models:

$$x_{ij} = \sum_k k\delta^x_{ijk}; \ \sum_k \delta^x_{ijk} = 1, \text{ all } i,j; \ \sum_j \delta^x_{ijk} = 1, \text{ all } i,k; \ \sum_i \delta^x_{ijk} = 1, \text{ all } j,k$$

$$y_{ik} = \sum_j j\delta^y_{ijk}, \ \sum_j \delta^y_{ijk} = 1, \text{ all } i,k; \ \sum_k \delta^y_{ijk} = 1, \text{ all } i,j; \ \sum_i \delta^y_{ijk} = 1, \text{ all } j,k$$

$$z_{jk} = \sum_i i\delta^z_{ijk}, \ \sum_i \delta^z_{ijk} = 1, \text{ all } j,k; \ \sum_k \delta^z_{ijk} = 1, \text{ all } i,j; \ \sum_j \delta^z_{ijk} = 1, \text{ all } i,k$$

Latin squares

```
{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])
```

The 3 formulations generate 3 identical MIP models:

$$x_{ij} = \sum_k k\delta^x_{ijk};\ \sum_k \delta^x_{ijk} = 1,\ \text{all } i,j;\ \sum_j \delta^x_{ijk} = 1,\ \text{all } i,k;\ \sum_i \delta^x_{ijk} = 1,\ \text{all } j,k$$

$$y_{ik} = \sum_j j\delta^y_{ijk},\ \sum_j \delta^y_{ijk} = 1,\ \text{all } i,k;\ \sum_k \delta^y_{ijk} = 1,\ \text{all } i,j;\ \sum_i \delta^y_{ijk} = 1,\ \text{all } j,k$$

$$z_{jk} = \sum_i i\delta^z_{ijk},\ \sum_i \delta^z_{ijk} = 1,\ \text{all } j,k;\ \sum_k \delta^z_{ijk} = 1,\ \text{all } i,j;\ \sum_j \delta^z_{ijk} = 1,\ \text{all } i,k$$

The solver declares $\delta^x_{ijk},\ \delta^y_{ijk},\ \delta^z_{ijk}$

whether **assign(row i, col j, 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

1	…	*k*	*k* + 1	…	*n*
$\texttt{term}_1$	…	$\texttt{term}_k$	$\texttt{term}_{k+1}$	…	$\texttt{term}_n$
$i_1,\ x^1_{i(1)}$	…	$i_k,\ x^k_{i(k)}$	i_{k+1}	…	i_n

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

generates the channeling constraints

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

Multiple `whether` variables

`whether` keywords serve as projection operators on the relation.

```
y[i,j,d] is whether assign(worker i, job j, day d)
```

Project out *d* :

```
y1[i,j] is whether assign(worker i, job j)
```

Project out *j* and *d* :

```
y2[i] is whether assign(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:

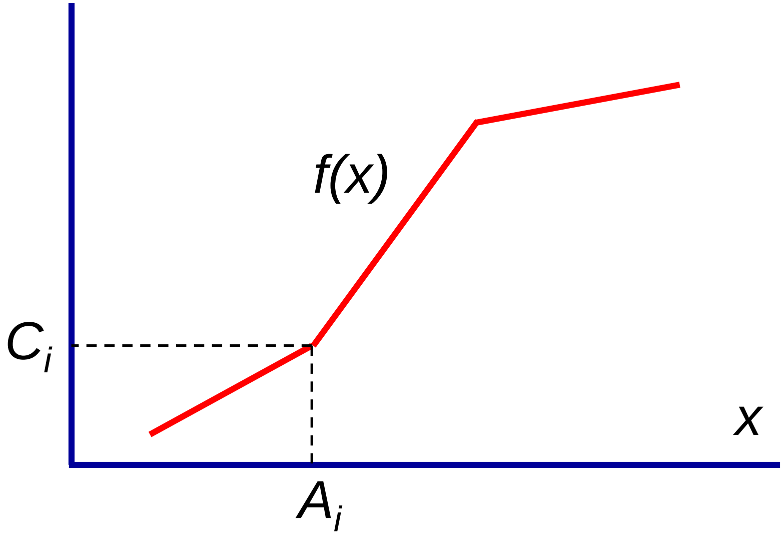
```
x is howmuch cost
```

which is short for

```
x is howmuch cost cost()
```

Piecewise linear

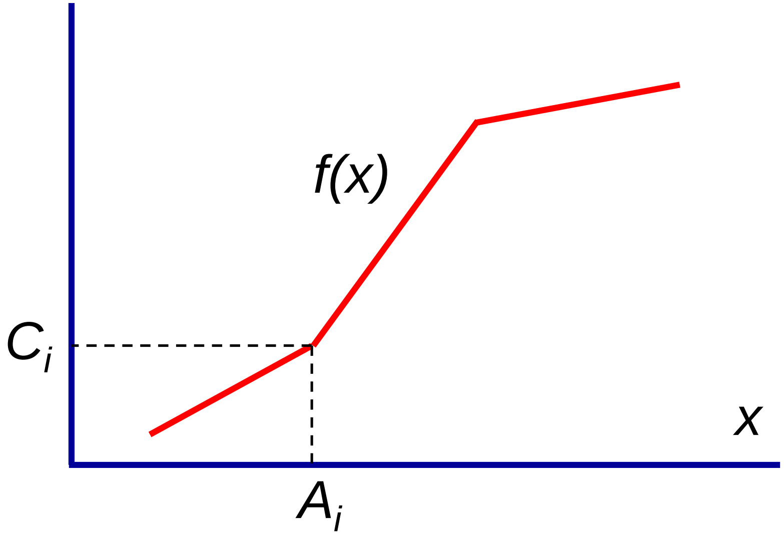
Piecewise linear function $z = f(x)$
Breakpoints in A, ordinates in C



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

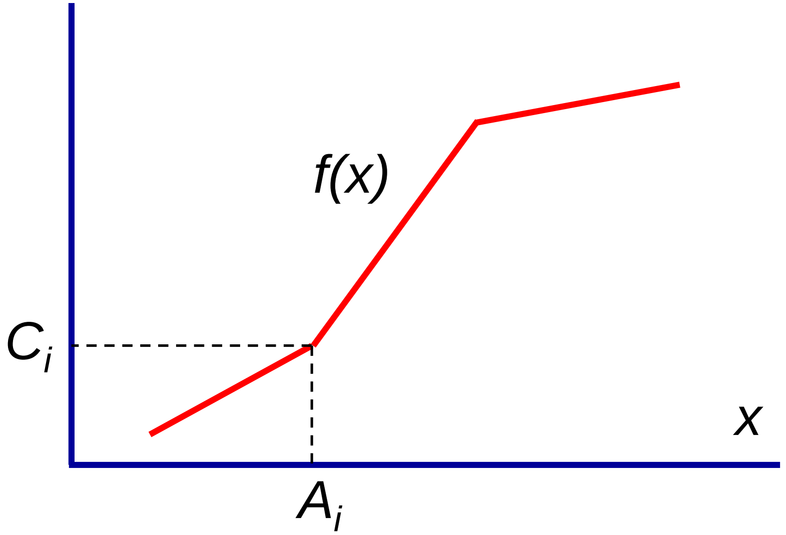
Solver generates the **locally ideal** model

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

$$(a_{i+1} - a_i)\delta_{i+1} \le \overline{x}_i \le (a_{i+1} - a_i)\delta_i, \quad \delta_i \in \{0,1\}, \; i = 1, \ldots, n-1$$

We need to declare auxiliary variables δ_i, x_i

Piecewise linear



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

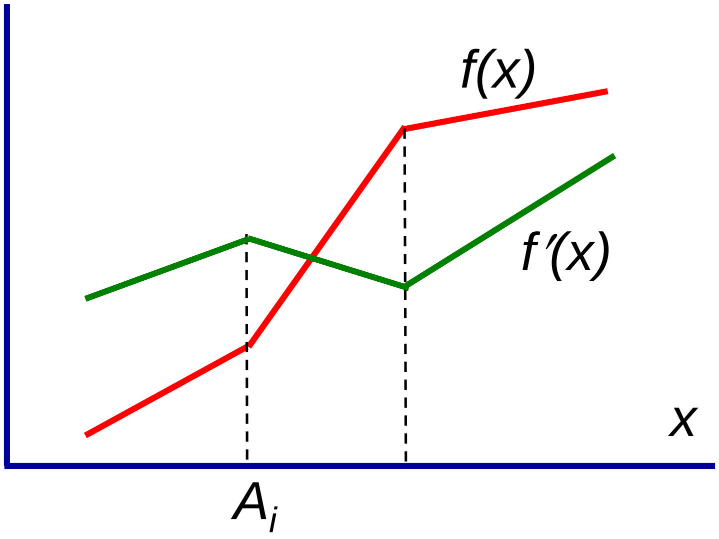
```
x is howmuch output
index in {1..n}
data A,C{index}
z is howmuch cost
piecewise(x,z,A,C)
```

piecewise constraint induces solver to declare a new index set that associates **index** with **A,** and use it to declare δ_i, x_i

```
xbar[i] is howmuch output.A(index i)
delta[i] is whether lastpositive output.A(index i)
```

Both declarations create predicates inherited from **output** and **A**

Piecewise linear



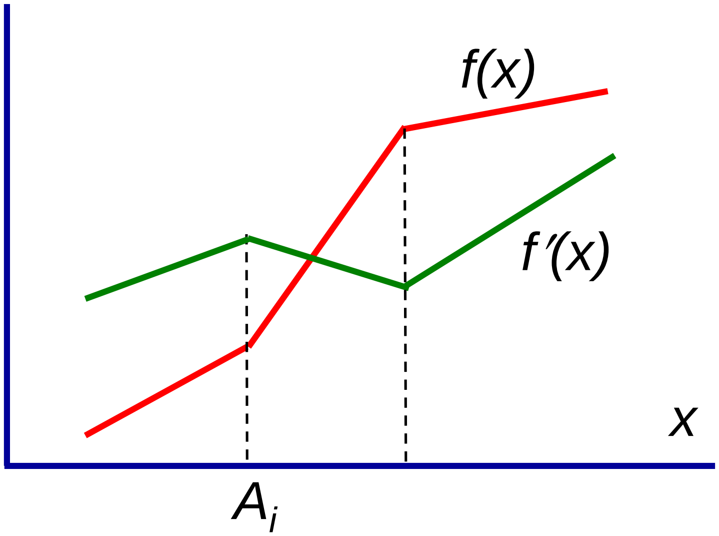
Suppose there is another piecewise function on the same break points

```
x is howmuch output
index in {1..n}
data A,C{index}
z is howmuch cost
piecewise(x,z,A,C)
data C′{index}
z′ is howmuch profit
piecewise(x,z′,A,C′)
x′[i] is howmuch cost output.A(index i)
delta′[i] is whether lastpositive output.A(index)
```

Piecewise linear

Suppose there is another piecewise function on the same break points

```
x is howmuch output
index in {1..n}
data A,C{index}
z is howmuch cost
piecewise(x,z,A,C)
data C′{index}
z′ is howmuch profit
piecewise(x,z′,A,C′)
x′[i] is howmuch cost output.A(index i)
delta′[i] is whether lastpositive output.A(index)
```



Because new piecewise constraint is associated with the same x and A, solver again creates **output.A**.

The solver creates variables δ_i' and x_i' with same types as δ_i and x_i and so identifies them.

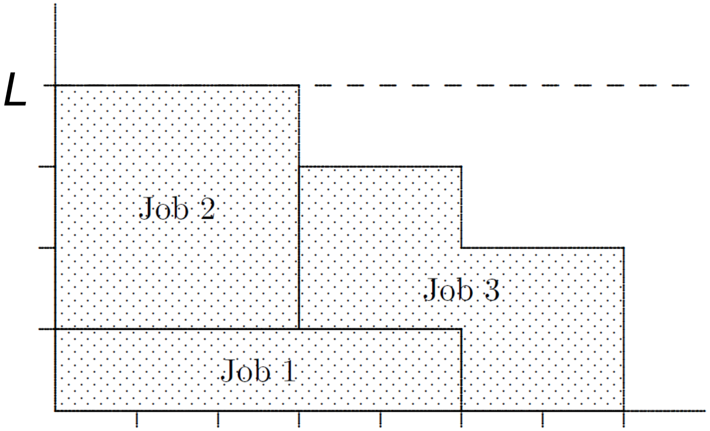
Interval variables

$$\text{cumulative}\left(x, D, R, L\right)$$

$$x_j \subseteq W_j, \text{ all } j$$

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

```
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)
```



Interval variables

$$\text{cumulative}(x, D, R, L)$$

$$x_j \subseteq W_j, \text{ all } j$$

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

```
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

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

$$\varphi_{jt} \ge \delta_{jt'}, \text{ all } t, t' \text{ with } 0 \le t - t' < D_j, \text{ all } j$$

```
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

$$\text{cumulative}\left(x, D, R, L\right)$$

$$x_j \subseteq W_j, \text{ all } j$$

$$\left|x_j^{\text{end}} - x_k^{\text{end}}\right| \geq T_0, \text{ all } j, k, \; j \neq k$$

```
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| >= T0
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

$$\text{cumulative}(x, D, R, L)$$

$$x_j \subseteq W_j, \text{ all } j$$

$$\left| x_j^{\text{end}} - x_k^{\text{end}} \right| \geq T_0, \text{ all } j, k, \ j \neq k$$

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| >= T0
delta[j,t] is whether running.start sched(job j, time t)
phi[j,t] is whether running sched(job j, time t)
```

Solver generates

$$\varepsilon_{jt} + \varepsilon_{kt'} \leq 1, \text{ all } t, t' \text{ with } 0 < t' - t < T_0, \text{ all } j, t \text{ with } j \neq k$$

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

Interval variables

Variables δ_{jt} and ε_{jt} are related by an offset.
Solver associates **running.*end*** in declaration of ε_{jt}
with **running.*start*** in declaration of δ_{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 δ_{jt} and ε_{jt} are related by an offset.
Solver associates `running.end` in declaration of ε_{jt}
with `running.start` in declaration of δ_{jt} and deduces

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

Solver also associates `running.end` in declaration of ε_{jt}
with `running` in declaration of ϕ_{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.

$$\min \sum_i D_{is_i}$$

$$\text{alldiff}(x),\ \text{circuit}(s)$$

$$x_i < x_j,\ \text{all } i, j \text{ with } \text{prec}_{ij} = 1$$

$$s_i \in \text{Succ}_i$$

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

TSP with Side Constraints

$$\min \sum_i D_{is_i}$$

$$\text{alldiff}(x),\ \text{circuit}(s)$$

$$x_i < x_j,\ \text{all } i, j \text{ with } \text{prec}_{ij} = 1$$

$$s_i \in \text{Succ}_i$$

Traveling salesman problem with missing arcs and precedence constraints.

`city, position in {1..n}`

`data D{city, city}` Distances

`data Prec{city, city}` `Prec[i,j]=1` if *i* must precede *j*

`data Succ{city}` `Succ[j]` = set of successors of city *j*

Two variable systems:

```
x[i] is which position ordering(city i)
s[i] is successor city ordering(city i) subset Succ[i]
```

TSP with Side Constraints

Traveling salesman problem with missing arcs and precedence constraints.

$$\min \sum_i D_{is_i}$$

$$\text{alldiff}(x),\ \text{circuit}(s)$$

$$x_i < x_j,\ \text{all } i, j \text{ with prec}_{ij} = 1$$

$$s_i \in \text{Succ}_i$$

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

Two variable systems:

```
x[i] is which position ordering(city i)
s[i] is successor city ordering(city i) subset Succ[i]
```

Precedence constraints require **x** variables

```
prec{city i, city j | Prec[i,j] = 1}: x[i] < x[j]
```

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

TSP with Side Constraints

Traveling salesman problem with missing arcs and precedence constraints.

$$\min \sum_i D_{is_i}$$

$$\text{alldiff}\left(x\right),\ \text{circuit}\left(s\right)$$

$$x_i < x_j,\ \text{all } i, j \text{ with } \text{prec}_{ij} = 1$$

$$s_i \in \text{Succ}_i$$

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

Two variable systems:

```
x[i] is which position ordering(city i)
s[i] is successor city ordering(city i) subset Succ[i]
```

Precedence constraints require **x** variables

```
prec{city i, city j | Prec[i,j] = 1}: x[i] < x[j]
```

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

```
min sum {city i} D[i,s[i]]   Objective function
```

TSP with Side Constraints

The solver can give **`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)
```

TSP with Side Constraints

The solver can give **`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)`

TSP with Side Constraints

The solver can give **`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 **`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)`**.
Solver generates cutting planes in **`w`**-space and **`s`**-space.

TSP with Side Constraints

The solver can give **`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)`**.
Solver generates cutting planes in **`w`**-space and **`s`**-space.

The **`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] = which city ordering(position k)
```

The solver generates the channeling constraints between **`y[k]`** and **`x[i]`** = which position is city *i*

TSP with Side Constraints

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

```
y[k] = which city ordering(position k)
```

The solver generates the channeling constraints between **`y[k]`** and **`x[i]`** = which position is city *i*

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

```
min 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!*