PETER J. STUCKEY, GUIDO TACK, GRAEME GANGE, JIP DEKKER

FROM CLP(R) TO MINIZINC THERE AND BACK AGAIN

- Three simultaneous advances : 1986-1987
- Alain Colmerauer builds Prolog III which includes arithmetic and sequence constraints (as well as term constraints)
- Pascal Van Hentenryck builds CHIP which includes finite domain constraints
- Joxan Jaffar and Jean-Louis Lassez define a semantics for logic programming extended with constraints

CLP(R)

- CLP(R) was a CLP language built by
  - Joxan Jaffar, Spiro Michaylov, Roland Yap and myself

- One of the first 3 CLP languages: Prolog III, CHIP, CLP(R)

- Incremental linear arithmetic + term solving
- Nonlinear arithmetic solving implemented by delay
- Powerful simplification/projection of constraints ("dump")

- Interesting applications:
  - generative architecture, options trading analysis

THE MOST USED CLP(R) PROGRAM?

- Mortgage simulator:
  - P principal of the loan (the amount borrowed)
  - T number of time periods
  - R repayment amount (each time period)
  - I interest rate (applied each time period)
  - B balance owing at the end of the loan

```
mg(P,T,R,I,B) :- T = 0, B = P.
mg(P,T,R,I,B) :- T >= 1, mg(P*(1+I) - R, T-1, I, B).
```

THE MOST USED CLP(R) PROGRAM?

- What can you do with it?
- If I borrow 1000 for 10 years at 10% repaying 150 per year how much do I owe finally?
  - `mg(1000, 10, 150, 10/100, B)`
  - `B = 203.129`
- How much can I borrow repaying 150 to owe nothing at the end?
  - `mg(P, 10, 150, 10/100, 0)`
  - `P = 921.685`
- What is the relationship between the principal, repayment and balance (10 years at 10%)
  - `mg(P, 10, R, 10/100, B)`
  - `P = 0.3855*B + 6.1446 * R`
  - Notice the simplification and projection of constraints

THE RISE OF CLP

- Constraint Programming is a Powerful Paradigm
  - state-of-the-art solutions to many problems

- CHIP: finite domain constraints
- Eclipse Prolog: finite domain and interval constraints

- Sicstus Prolog: finite domain constraints
- SWI-Prolog: finite domains, Booleans, rationals, floats
- Ciao Prolog: rationals, reals, finite domains

- Optimization Programming Language (OPL)
  - ILOG modelling language for finite domain constraints
- A modelling language
  - not a programming language
- State the constraints of the problem
  - Let the underlying solve ILOG Solver solve the problem
- Programmable search
  - but no meta-programming capabilities

G12 PROJECT

- Group 12 of the periodic table

- Zinc a solver independent modelling language
- Cadmium a model rewriting language
- Mercury an existing LP language for solver implementation

- A new approach to Constraint Programming
  - Note Zinc like OPL is not CLP
  - But Mercury is a full PL

- HAL then G12 (Zinc) then ...
- 2006 conference workshop
  - "Future of Constraint Programming"
  - Many people complained about a lack of a standard for CP
-
  -
  -
-
  -
  -

- HAL then G12 (Zinc) then ...
- 2006 conference workshop
  - "Future of Constraint Programming"
  - Many people complained about a lack of a standard for CP
- CPAIOR2007 we submitted
  - "MiniZinc: a Standard Modelling Language for CP"
  - Rejected!
-
  -
  -

- HAL then G12 (Zinc) then ...
- 2006 conference workshop
  - "Future of Constraint Programming"
  - Many people complained about a lack of a standard for CP
- CPAIOR2007 we submitted
  - "MiniZinc: a Standard Modelling Language for CP"
  - Rejected!
- CP2007 we submitted
  - "MiniZinc: Towards a Standard Modelling Language for CP"
  - Accepted

MINIZINC SUCCESSES

- MiniZinc is designed to be solver-independent
- Allowed easy comparison of
  - Constraint Programming solvers
  - Mixed Integer Programming solvers
  - Boolean SATisfiability solvers
  - SAT Modulo Theory solvers
  - Constraint-based Local Search solvers

- Many uses worldwide (in communities far from CP)
- 2500 downloads per month

- The mortgage example in MiniZinc

```
int: T;                  % time period
var 0.0..10000.0: I;     % interest rate
var 0.0..10000.0: R;     % repayment
var 0.0..10000.0: P;     % principle
var 0.0..10000.0: B;     % balance

array[1..T] of var float: mortgage;
constraint
   balance = mortgage[T] /\
   mortgage[1] = P + (P * (1+I)) - R /\
   forall(i in 2..T) ( mortgage[i] =  mortgage[i-1] * (1+I) - R )
;

output ["P = \(P); T = \(T); I = \(I); R = \(R); B = \(B)\n"];
```

Query 1

```
T = 10;
P = 1000.0;
I = 10/100;
R = 150;
```

```
P = 1000.0; T = 10; I = 0.01; R = 150; B = 203.129
```

Query 2

```
T = 10;
B = 0.0;
I = 10/100;
R = 150;
```

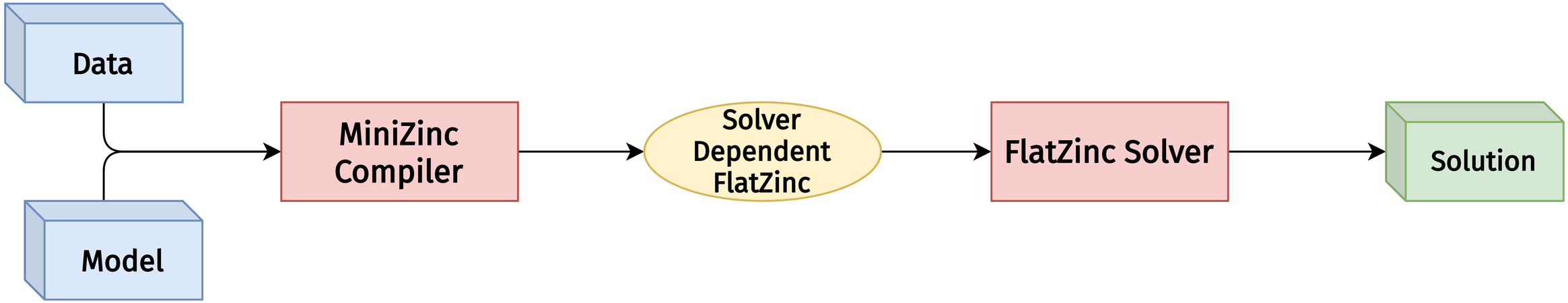
```
P = 921.685; T = 10; I = 0.01; R = 150; B = 0.0
```

- A more usual example: job shop scheduling

```
int: n_machines;                                % The number of machines.
int: n_jobs;                                    % The number of jobs.
set of int: JOB = 1..n_jobs;
set of int: TASK = 1..n_machines;
set of int: MACHINE = 1..n_machines;
array[JOB,TASK] of MACHINE: m;                  % machine for each task
array[JOB,TASK] of int: d;                      % duration for each task
int: max_duration = sum(array1d(d));

array[JOB,TASK] of 0..max_duration: s;          % start time for each task;
var 0..max_duration: makespan;                  % end of all tasks
constraint forall(j in JOB, t in TASK)(s[j,t] + d[j,t] <= makespan);
constraint forall(j in JOB, t in 1..n_machines-1)
                 (s[j,t] + d[j,t] <= s[j,t+1]);
constraint forall(ma in MACHINE)
                 (disjunctive([ s[j,t] | j in JOB, t in TASK where m[j,t] = ma],
                              [ d[j,t] | j in JOB, t in TASK where m[j,t] = ma]));
solve minimize makespan;
```

MINIZINC PROCESS



A VICTIM OF ITS OWN SUCCESS

- MiniZinc was designed for CP models
  - a small number of highly complex global constraints
  - e.g. rotating workforce scheduling: 2 constraints
- MiniZinc is now used for MIP and SAT models
  - hundreds of thousands of simple constraints

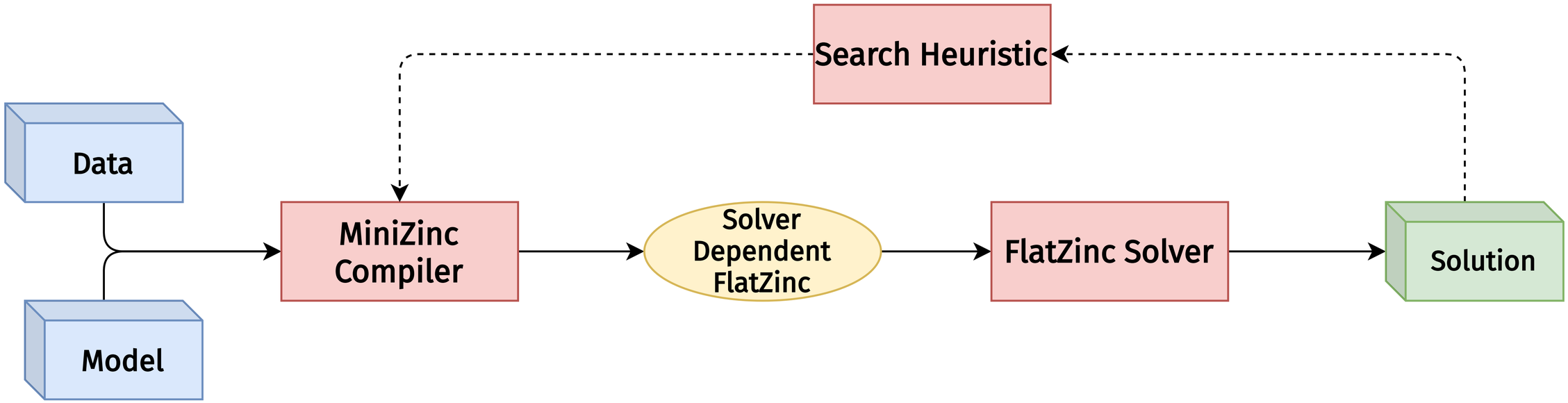
- MiniZinc was designed to solve NP-hard problems
  - solve time is LARGE, even for small models
- MiniZinc is used to define easy problems
  - solve time is instantaneous, even for LARGE models

- PROBLEM: translation time is too slow

INCREMENTALITY

- Many discrete optimization problems
  - repeatedly solve a very similar problem
- e.g. lexicographic minimization (x,y,z)
  - find solution (a,b,c)
  - add constraint $(x,y,z) < (a,b,c)$
  - repeat until no solution
- e.g. large neighbourhood search
  - find a solution
  - fix most of the parts of the solution
  - search for a better solution
  - repeat until resources exhausted.

INCREMENTAL MINIZINC



- MiniZinc is frequently used for incremental solving processes
- Recompilation of the entire model on each small change

THE ANSWER

MINIZINC 3.0

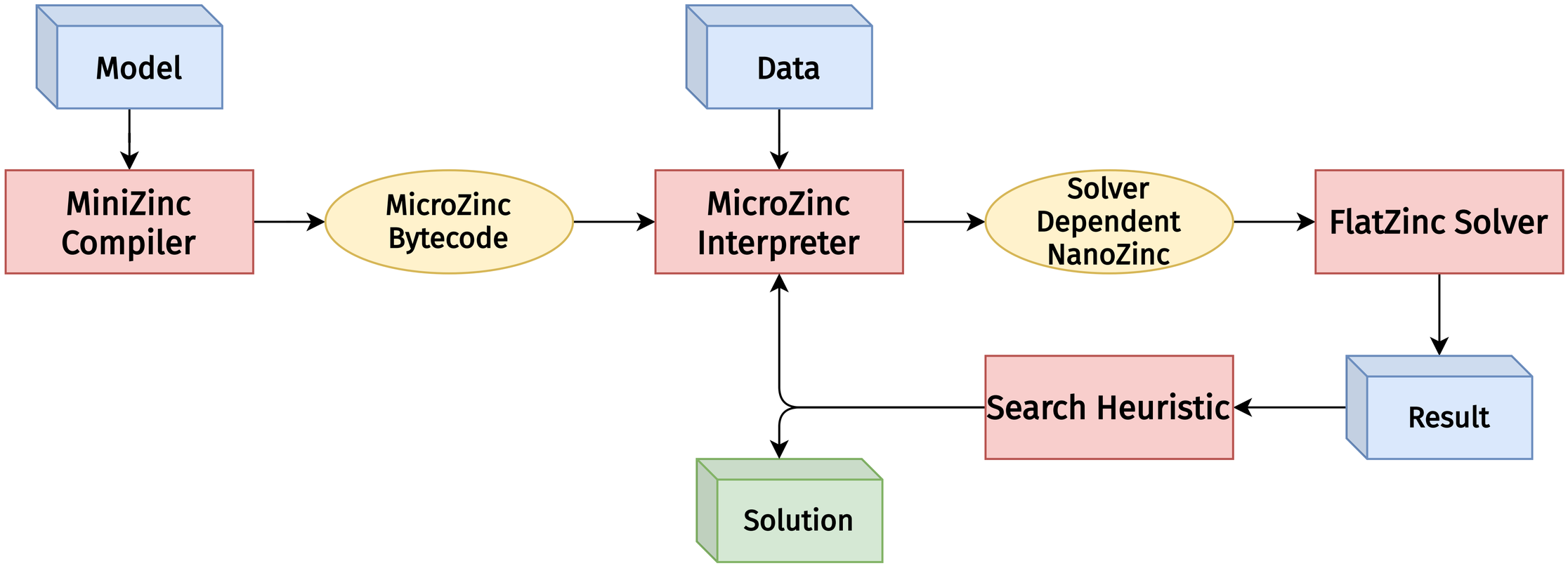
A NEW MINIZINC SYSTEM

- Compilation is not easy (we did this for Zinc)
  - MiniZinc is a rewriting system that relies on problem data

- MiniZinc 3.0: An efficient incremental rewriting engine

- MiniZinc compiles to
- MicroZinc (advanced features removed by program transformation)
  - rewrites to
- NanoZinc (a list of variables and defining constraints)
  - outputs to
- FlatZinc (for a solver)

A NEW ARCHITECTURE



FROM MINI TO MICRO

- We introduce a minimal subset of MiniZinc called MicroZinc containing:
  - (Pure and total) Functions/Calls
  - Let expressions
  - If-then-else expressions (with decidable *at interpretation time* conditions)
  - Array comprehensions
- Through rewriting all MiniZinc models can be expressed as MicroZinc

- Variable declarations with attached constraints
  - `var` <set> : <varname>

    └── <defining-constraints>
- Flat constraints (each subexpression is named)
  - builtins: `int_minus(x,3)`
  - MicroZinc predicates: `abs(u,v)`

CONSTRAINT PROPAGATION

- The rewriting applies constraint propagation rules to
  - simplify expressions
  - tighten bounds on variables
  - eliminate redundant constraints
  - eliminate dead variables/constraints (by reference counting)
  - substitute one variable for another for x = y

- It also performs common subexpression elimination
- Crucial for solving performance

EXAMPLE

- constraint x* (y div z) + y + 1 >= abs(y - z); var -10..10: x; var 0..10: y; var -10..-1: z;

```
var -10..10: x;
var 0..10: y;
var -10..-1: z;
var 1..20: u;
└── constraint u = y - z;
var 1..20: v;
└── constraint v = abs(u);
var -20..0: w;
├── constraint z != 0;
└── constraint w = y div z;
var -200..200: t;
└── constraint t = x*w;
var bool: top
└── constraint t + y + 1 >= v;
```

FlatZinc

```
var -10..10: x;
var 0..10: y;
var -10..-1: z;
var 1..20: u;
constraint int_minus(y,z,u);
var 1..20: v;
constraint int_abs(v,u);
var -20..0: w;
constraint int_neq(z,0);
constraint int_div(y,z,w);
var -200..200: t;
constraint int_mul(x,w,t);
constraint int_lin_ge([1,1,-1],[t,y,v],-1);
```

EXAMPLE

- constraint x* (y div z) + y + 1 >= abs(y - z); var -10..10: x; var 0..10: y; var -10..-1: z;

```
var -10..10: x;
var 0..10: y;
var -10..-1: z;
var 1..20: u;
└── constraint u = y - z;
var 1..20: v;
└── constraint v =  u;
var -20..0: w;
├── constraint z != 0;
└── constraint w = y div z;
var -200..200: t;
└── constraint t = x*w;
var bool: top
└── constraint t + y + 1 >= v;
```

since u > 0

EXAMPLE

- constraint x* (y div z) + y + 1 >= abs(y - z); var -10..10: x; var 0..10: y; var -10..-1: z;

```
var -10..10: x;
var 0..10: y;
var -10..-1: z;
var 1..20: u;
└── constraint u = y - z;
var 1..20: v;
└── constraint v =  u;
var -20..0: w;
├── constraint z != 0;
└── constraint w = y div z;
var -200..200: t;
└── constraint t = x*w;
var bool: top
└── constraint t + y + 1 >= u;
```

since u > 0

substitution in linear

EXAMPLE

- constraint x* (y div z) + y + 1 >= abs(y - z); var -10..10: x; var 0..10: y; var -10..-1: z;

```
var -10..10: x;
var 0..10: y;
var -10..-1: z;
var 1..20: u;
└── constraint u = y - z;
var 1..20: v;
└── constraint v =  u;
var -20..0: w;
├── constraint z != 0;
└── constraint w = y div z;
var -200..200: t;
└── constraint t = x*w;
var bool: top
└── constraint t + y + 1 >= y - z;
```

since u > 0

substitution in linear

EXAMPLE

- constraint x* (y div z) + y + 1 >= abs(y - z); var -10..10: x; var 0..10: y; var -10..-1: z;

```
var -10..10: x;
var 0..10: y;
var -10..-1: z;
var 1..20: u;
└── constraint u = y - z;
var 1..20: v;
└── constraint v =  u;
var -20..0: w;
├── constraint z != 0;
└── constraint w = y div z;
var -200..200: t;
└── constraint t = x*w;
var bool: top
└── constraint t + 1 >= -z;
```

since u > 0

simplification in linear

EXAMPLE

- constraint x* (y div z) + y + 1 >= abs(y - z); var -10..10: x; var 0..10: y; var -10..-1: z;

```
var -10..10: x;
var 0..10: y;
var -10..-1: z;
var 1..20: u;
└── constraint u = y - z;
var 1..20: v;
└── constraint v =  u;
var -20..0: w;
├── constraint true;
└── constraint w = y div z;
var -200..200: t;
└── constraint t = x*w;
var bool: top
└── constraint t + 1 >= -z;
```

since u > 0

since z < 0

simplification in linear

EXAMPLE

- constraint x* (y div z) + y + 1 >= abs(y - z); var -10..10: x; var 0..10: y; var -10..-1: z;
  - var -10..10: x;
  - var 0..10: y;
  - var -10..-1: z;
  - ~~var 1..20: u;~~
    - ~~constraint u = y - z;~~
  - ~~var 1..20: v;~~
    - ~~constraint v = u;~~
  - var -20..0: w;
    - ~~constraint true;~~
    - constraint w = y div z;
  - var -200..200: t;
    - constraint t = x*w;
  - var bool: top
    - constraint t + 1 >= -z;

since u is not referred to outside its defining constraints

EXAMPLE

- constraint x* (y div z) + y + 1 >= abs(y - z); var -10..10: x; var 0..10: y; var -10..-1: z;
  - `var -10..10: x;`
  - `var 0..10: y`
  - `var -10..-1: z;`
  - `var -20..0: w;`
    - `constraint w = y div z;`
  - `var -200..200: t;`
    - `constraint t = x*w;`
  - `var bool: top`
    - `constraint t + 1 >= -z;`
- Now suppose we discover that x = 0

EXAMPLE

- constraint x* (y div z) + y + 1 >= abs(y - z); var -10..10: x; var 0..10: y; var -10..-1: z;
  - `var 0..0: x;` since x = 0
  - `var 0..10: y`
  - `var -10..-1: z;`
  - `var -20..0: w;`
    - `constraint w = y div z;`
  - `var -200..200: t;`
    - `constraint t = x*w;`
  - `var bool: top`
    - `constraint t + 1 >= -z;`
- Now suppose we discover that x = 0

EXAMPLE

- constraint x* (y div z) + y + 1 >= abs(y - z); var -10..10: x; var 0..10: y; var -10..-1: z;
  - `var 0..0: x;` since x = 0
  - `var 0..10: y`
  - `var -10..-1: z;`
  - `var -20..0: w;`
    - `constraint w = y div z;`
  - `var -200..200: t;`
    - `constraint t = 0*w;` since x = 0
  - `var bool: top`
    - `constraint t + 1 >= -z;`
- Now suppose we discover that x = 0

EXAMPLE

- constraint x* (y div z) + y + 1 >= abs(y - z); var -10..10: x; var 0..10: y; var -10..-1: z;
  - `var 0..0: x;` since x = 0
  - `var 0..10: y`
  - `var -10..-1: z;`
  - `var -20..0: w;`
    - `constraint w = y div z;`
  - `var -200..200: t;`
    - `constraint t = 0;` since x = 0
  - `var bool: top`
    - `constraint t + 1 >= -z;`
- Now suppose we discover that x = 0

EXAMPLE

- constraint x* (y div z) + y + 1 >= abs(y - z); var -10..10: x; var 0..10: y; var -10..-1: z;
  - `var 0..0: x;` since x = 0
  - `var 0..10: y`
  - `var -10..-1: z;`
  - `var -20..0: w;`
    - `constraint w = y div z;`
  - `var 0..0: t;` since t = 0
    - `constraint t = 0;` since x = 0
  - `var bool: top`
    - `constraint t + 1 >= -z;`
- Now suppose we discover that x = 0

EXAMPLE

- constraint x* (y div z) + y + 1 >= abs(y - z); var -10..10: x; var 0..10: y; var -10..-1: z;
  - `var 0..0: x;` since x = 0
  - `var 0..10: y`
  - `var -10..-1: z;`
  - `var -20..0: w;`
    - `constraint w = y div z;`
  - `var 0..0: t;` since t = 0
    - `constraint t = 0;` since x = 0
  - `var bool: top`
    - `constraint 1 >= -z;` since t = 0
- Now suppose we discover that x = 0

EXAMPLE

- constraint x* (y div z) + y + 1 >= abs(y - z); var -10..10: x; var 0..10: y; var -10..-1: z;
  - `var 0..0: x;` since x = 0
  - `var 0..10: y`
  - `var -1..-1: z;` from linear
  - `var -20..0: w;`
    - `constraint w = y div z;`
  - `var 0..0: t;` since t = 0
    - `constraint t = 0;` since x = 0
  - `var bool: top`
    - `constraint 1 >= -z;` since t = 0
- Now suppose we discover that x = 0

EXAMPLE

- constraint x* (y div z) + y + 1 >= abs(y - z); var -10..10: x; var 0..10: y; var -10..-1: z;
  - `var 0..0: x;`
  - `var 0..10: y`
  - `var -1..-1: z;`
  - ~~`var -20..0: w;`~~
    - ~~`constraint w = y div z;`~~
  - ~~`var 0..0: t;`~~
    - ~~`constraint t = 0;`~~
  - `var bool: top`
    - ~~`constraint 1 >= -z;`~~
- Now suppose we discover that x = 0

- Calls without definition are untouched
- They create no FlatZinc output

```
predicate lex_min_{i}(array[int] of var int:x) =
      lex_less(x,sol(x)) /\
      lex_min_{i+1}(x);
```

- Original NanoZinc has no defn of `lex_min_0` and
  - `var bool: top;`

```
├── constraint ...
└── constraint lex_min_0(x);
```

- Each iteration adds a definition of `lex_min_{i}`

BACKTRACKING

- In fact we only need the LAST `lex_less` constraint
- Backtrack to base model, use

```
predicate lex_min(array[int] of var int:x) =
      lex_less(x,sol(x));
```

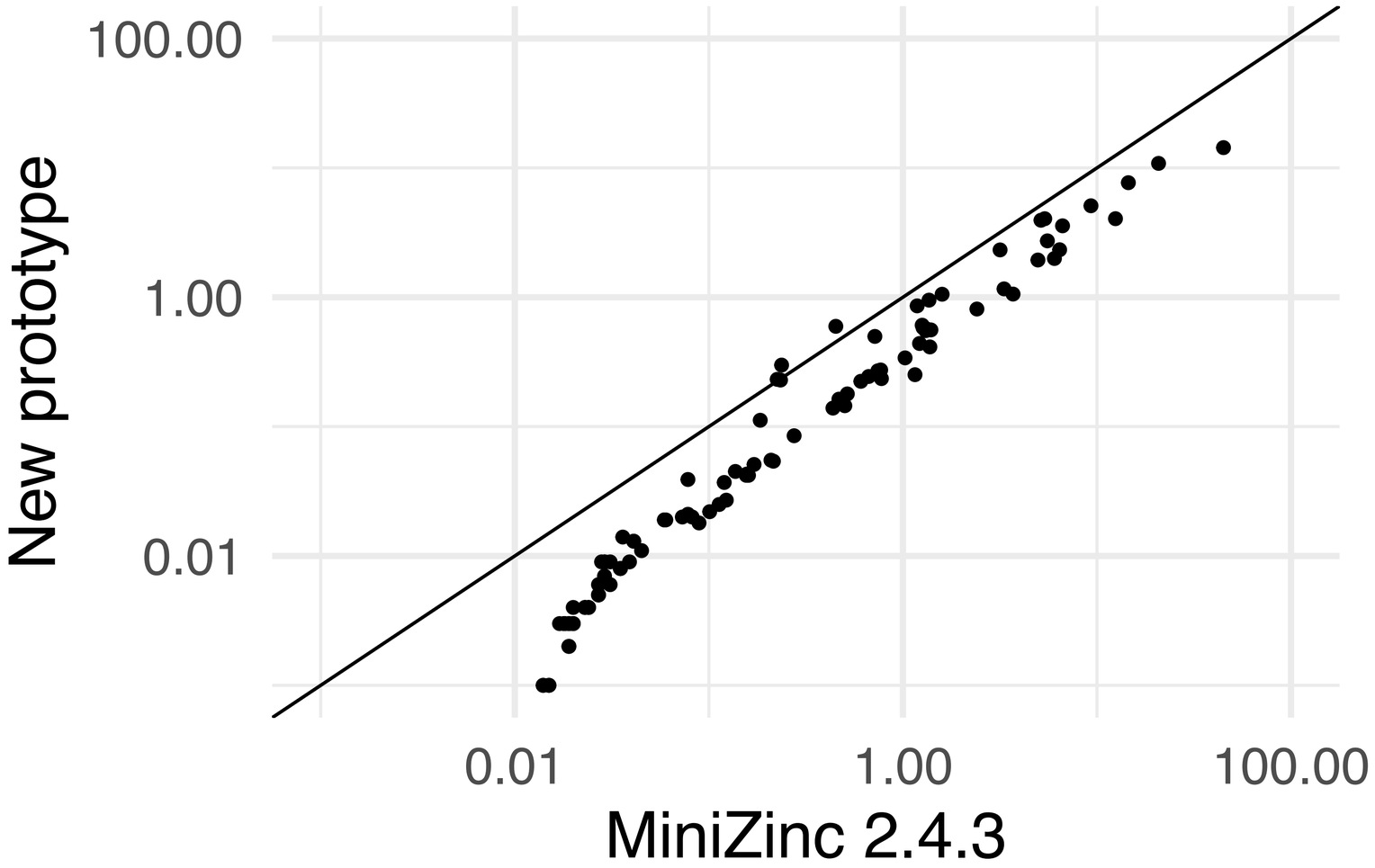
- Original NanoZinc has no defn of `lex_min_0` and
- Since the last solution `sol(x)` changes, we simply repeat

- So our incremental interface also supports backtracking

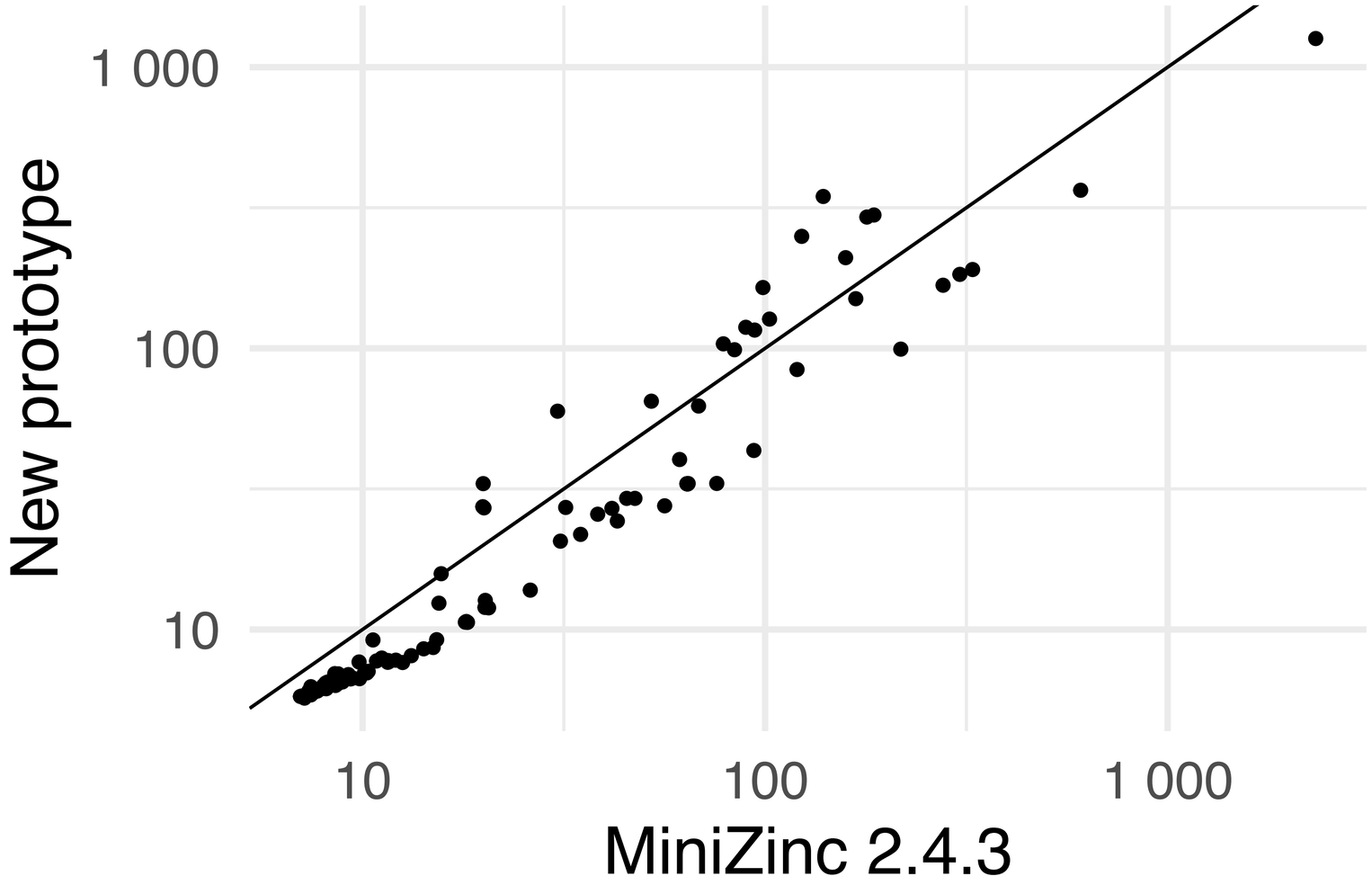
PRELIMINARY RESULTS

- The new compiler works
  - much more optimization and debugging required
  - not ready for release by a long way

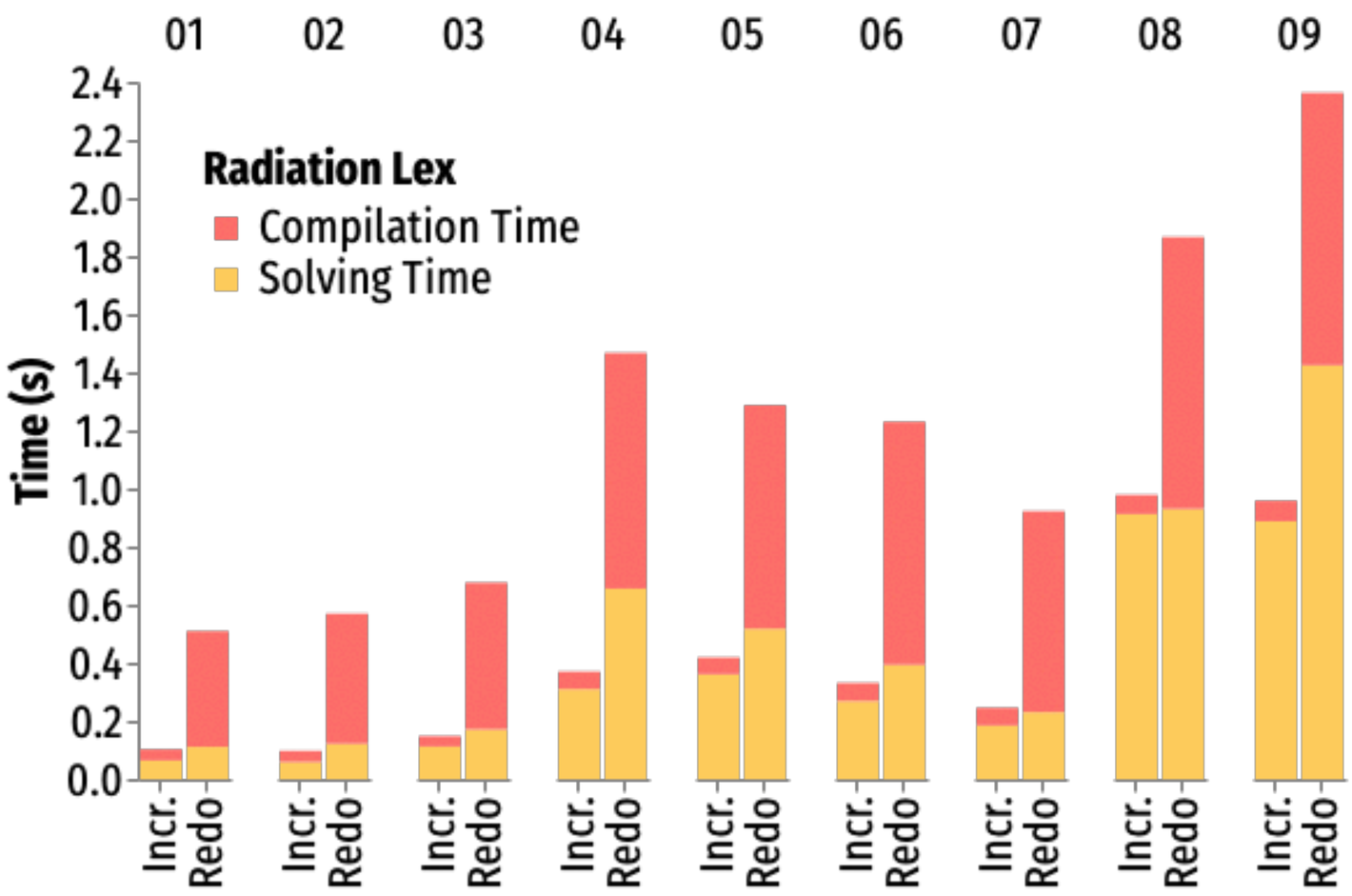
MODEL COMPILATION COMPARISON



MEMORY COMPARISON



LEXICOGRAPHIC SOLVING EXAMPLE



CONCLUSIONS

- CP modelling languages lost much of the expressiveness of CLP
- But incremental compilation for MiniZinc in some sense recreates a CLP solver
  - incremental compiler is a propagation solver
  - continuations for incrementality are analogous to CLP predicates
  - backtracking is supported
- Advantages
  - more efficient compile time (particularly for incremental use)
  - produces more efficient FlatZinc
  - incremental solvers can take advantage of incrementality
- Lessons:
  - once in a while rebuild from scratch
  - old good ideas have a habit of reappearing

QUESTIONS