Expressive Models for Monadic Constraint Programming

Pieter Wuille Tom Schrijvers

ModRef'10
St. Andrews, September 6, 2010

FD-MCP?

FD-MCP

FD-MCP:

- is a CP system for Finite-Domain (FD) problems
- is a subsystem of MCP, a Haskell CP framework
- provides an EDSL for writing FD problems

Why an EDSL for CP Modelling?

EDSL

An EDSL (Embedded Domain Specific Language) is

- more than an API: includes abstraction and syntactic sugar
- still embedded in host language, and able to interact with it

Advantages

The result allows advantages of both:

- Concise notation
- Declarative syntax (not a sequence of function calls)
- Full language feature set
- Directly usable results

Haskell and MCP

Haskell

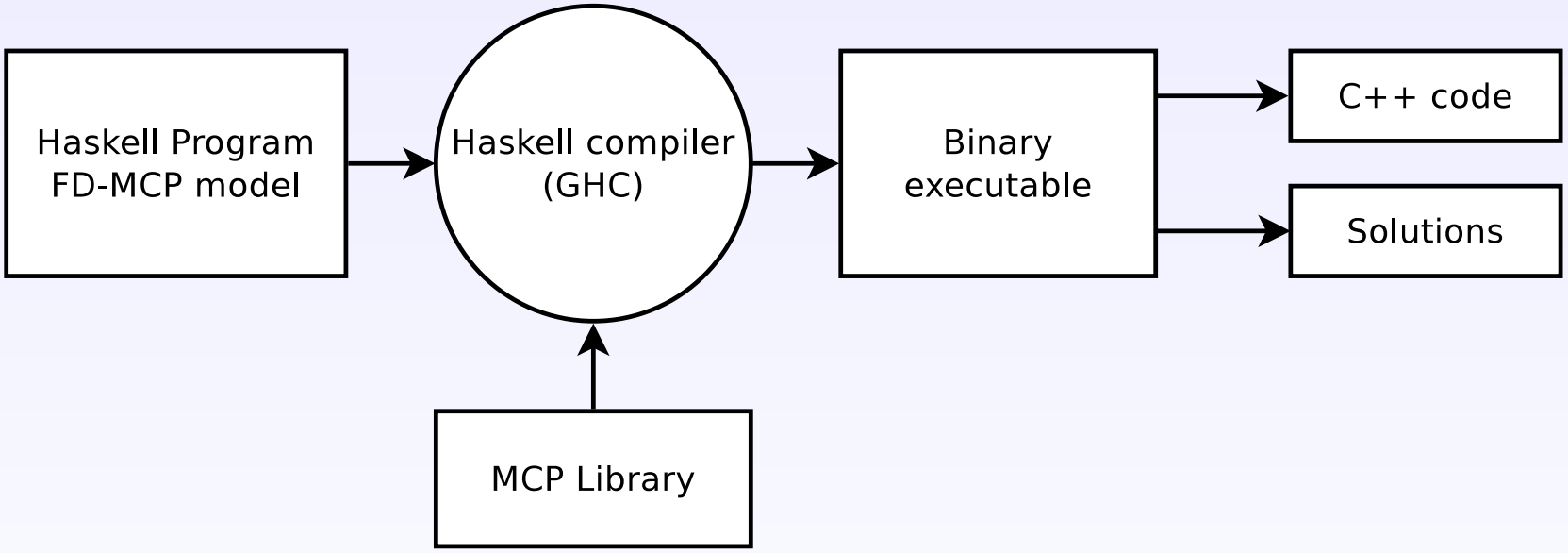
Haskell:

- Lazy, purely functional programming language
- Support for first-class and higher-order functions
- Uses monads to order stateful operations
- Supports user-defined operators and overloading through type classes

MCP

- Framework for CP in Haskell
- Does not fix variable domain, solver backend, search strategy, ...

Structure


Haskell Program
FD-MCP model
Haskell compiler
(GHC)
Binary
executable
C++ code
Solutions
MCP Library

Expressions: example

Example ($x > 5 \wedge x < 10 \wedge x^2 = 49$)

```
model = exists $ \x -> do      -- request a variable x
    x @> 5                     -- state that x>5
    x @< 10                    -- state that x<10
    x*x @= 49                  -- state that x*x=49
    return x                   -- return x
```

Expressions

Expressions

- Everything is written as expressions
- Constraints are equivalent to boolean expressions
- New variables are introduced by passing a function that takes an expression representing the new variable as argument, to `exists`

Parameters: example

Example ($x > 5 \wedge x < 10 \wedge x^2 = p$)

```
model p = exists $ \x -> do  -- request a variable x
    x @> 5                   -- state that x>5
    x @< 10                  -- state that x<10
    x*x @= p                 -- state that x*x=p
    return x                 -- return x
```

Parameters

- Problem classes are written as functions that take an expression as parameter
- Known values can be passed at runtime, to obtain a problem instance
- Model functions can be compiled as-is to C++ code

Higher-order constructs: examples

Example ($a + b + c + d = 10 \wedge a^2 + b^2 + c^2 + d^2 = 30$)

```
model = exists $ \arr -> do
    size arr @= 4
    csum arr @= 10
    csum (cmap (\x -> x*x) arr) @= 30
    return arr
```

Higher-order constructs

Higher-order constructs

- Use equivalents of typical higher-order functions as primitives:
  - *cmap f* $[a_1, a_2, a_3, \ldots]$: $[f(a_1), f(a_2), f(a_3), \ldots]$
  - *cfold f i* $[a_1, a_2, a_3, \ldots]$: $\ldots f(f(f(i, a_1), a_2), a_3) \ldots$
- To build typical CP higher-order constructs on top of
  - *forall c*: *fold* $(\wedge)$ *true c*
  - *csum c*: *cfold* $(+)\,0\,c$
  - *count v c*: *cfold* $(p\,i \rightarrow p + (i = v))\,c$
  - ...

Monadic bind

Monadic bind

- Boolean expressions can be used as solver actions that enforce their truth
- Solver actions can be combined using monadic bind
- Haskell provides syntactic sugar for this

These are equivalent:

```
model = exists $ \x -> do
  x @> 5
  x @< 10
```

```
model = exists (\x ->
  (x @> 5) @&& (x @< 10))
```

Building of expression tree

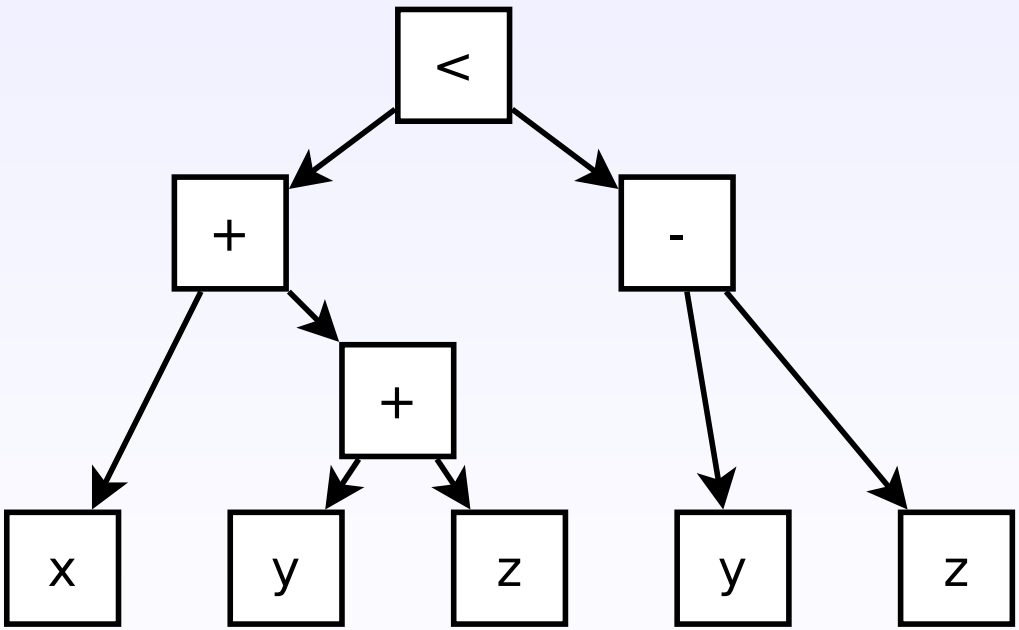
Building of expression tree

- The EDSL: Haskell functions and operators
- Syntactic sugar for boolean, integer and array expressions
- Models are monadic actions that introduce variables and post boolean expressions
- Evaluate at runtime to an expression tree

Building of expression tree

```
x+y+z @< z-y
```

```
Less (Plus x (Plus y z)) (Minus z y)
```



Expression tree simplifications

Simplifications

- Simple pattern matching on the tree
- Applies some mathematical identities
- Attempts to minimize variable references and tree nodes

Expression tree simplifications: examples

- $X + 0 \rightarrow X$
- X - X $\rightarrow$ 0
- $X + X \rightarrow 2*X$
- $(a + (b + X)) \rightarrow (a+b) + X$
- size [a] $\rightarrow$ 1
- ...

Conversion to Constraint Network Graph

For optimization purposes:

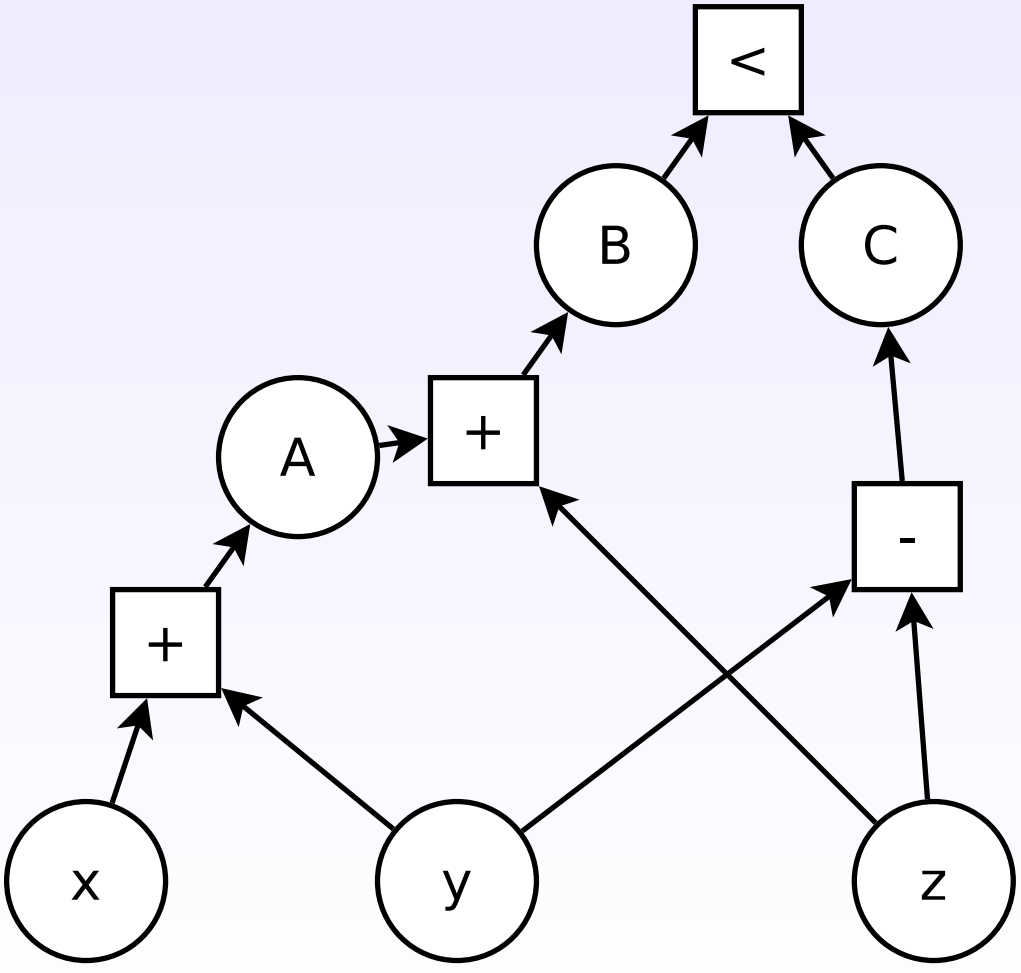
- We need information about a constraint's variables.
- We need information those variables' constraints.
- ...
- Syntax tree does not make this explicit

So we:

- We merge identical leaf nodes together, resulting in a graph
- ...or even whole identical subtrees (CSE)
- We turn higher-order constructs without flattening into subgraphs

Conversion to Constraint Network Graph

```
x+y+x @< z-y
```



Graph-based optimizations

Graph-based optimizations

Certain subgraphs can be recognized and replaced:

- A fold that sums values can become a sum
- A fold that sums equalities against a constant can become a count
- A fold that sums expressions can become a sum of a map

Mapping to solver-specific constraints

So far:

What we have

- A graph representation of the problem (class)
- Possibly still parametrized
- Compact, not flattened
- Independent of the solver's supported constraints

Next: mapping to solver-specific constraints

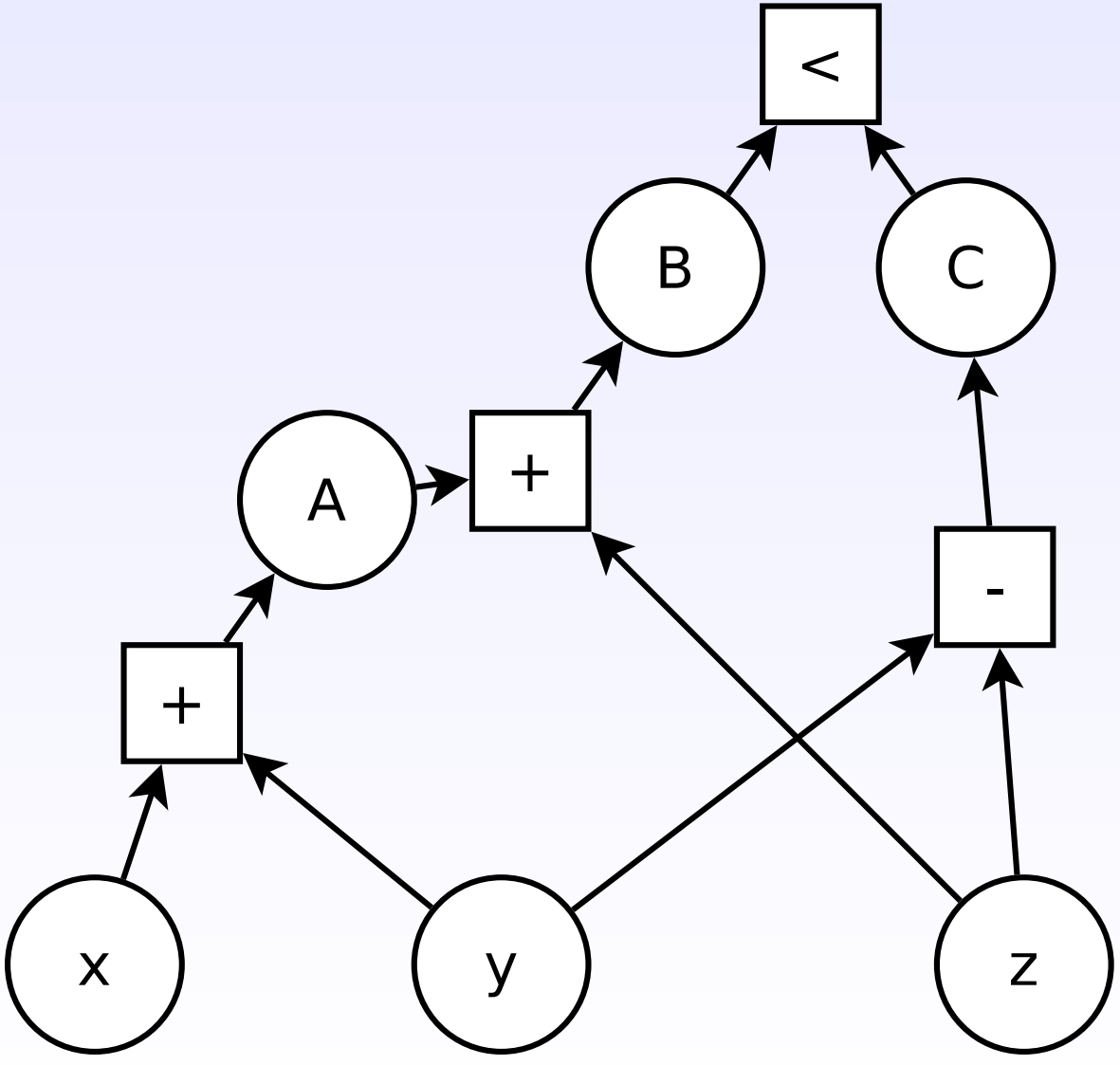
Mapping to solver-specific constraints

Annotation algorithm

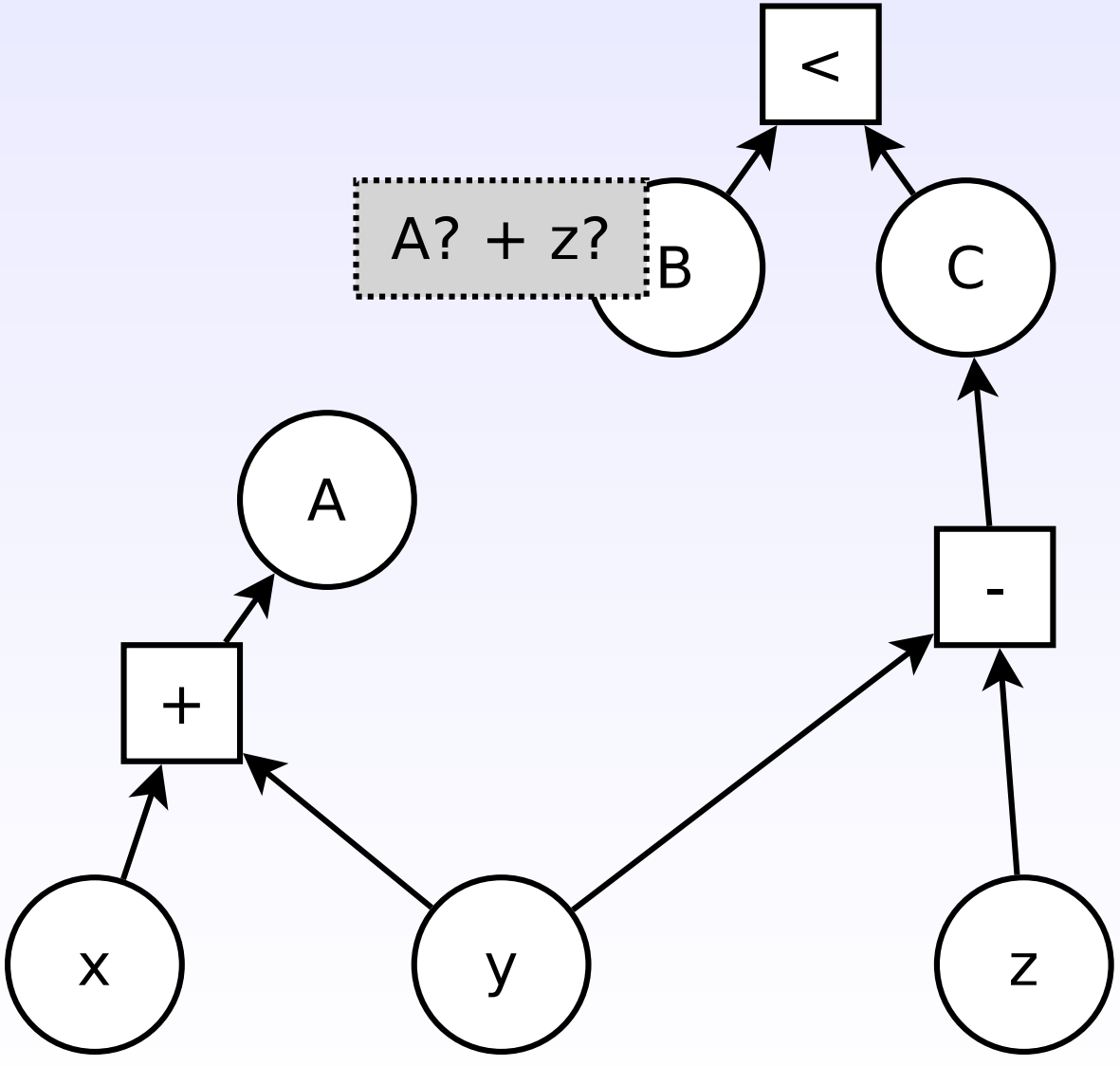
- Try to write nodes in function of other nodes, absorbing edges
- Start with options that may produce simple results
- Work recursively, but eager (no backtracking)
- Store resulting information in annotations on nodes

When all nodes are annotated, the remaining edges are translated to constraints

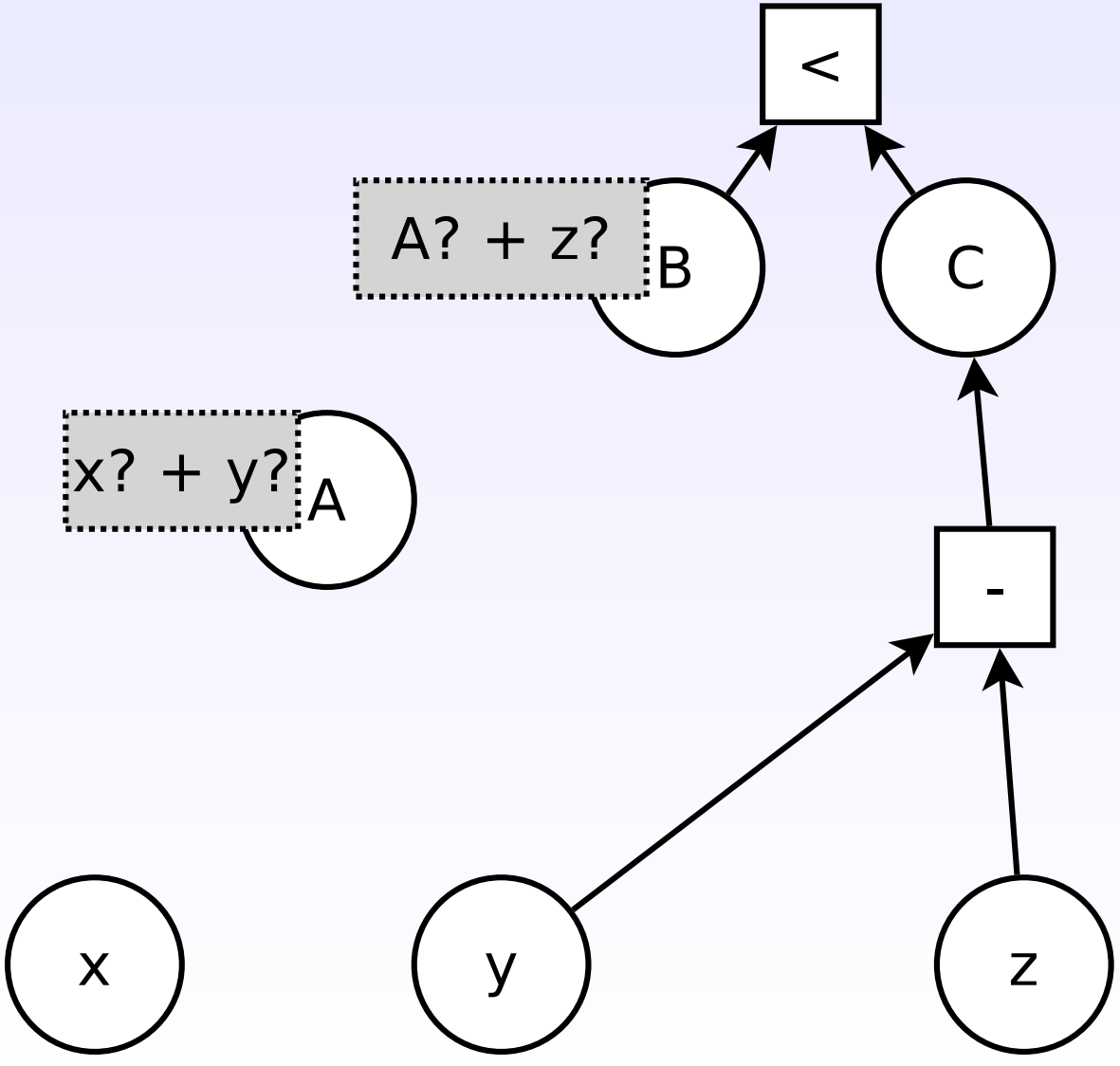
Mapping to solver-specific constraints



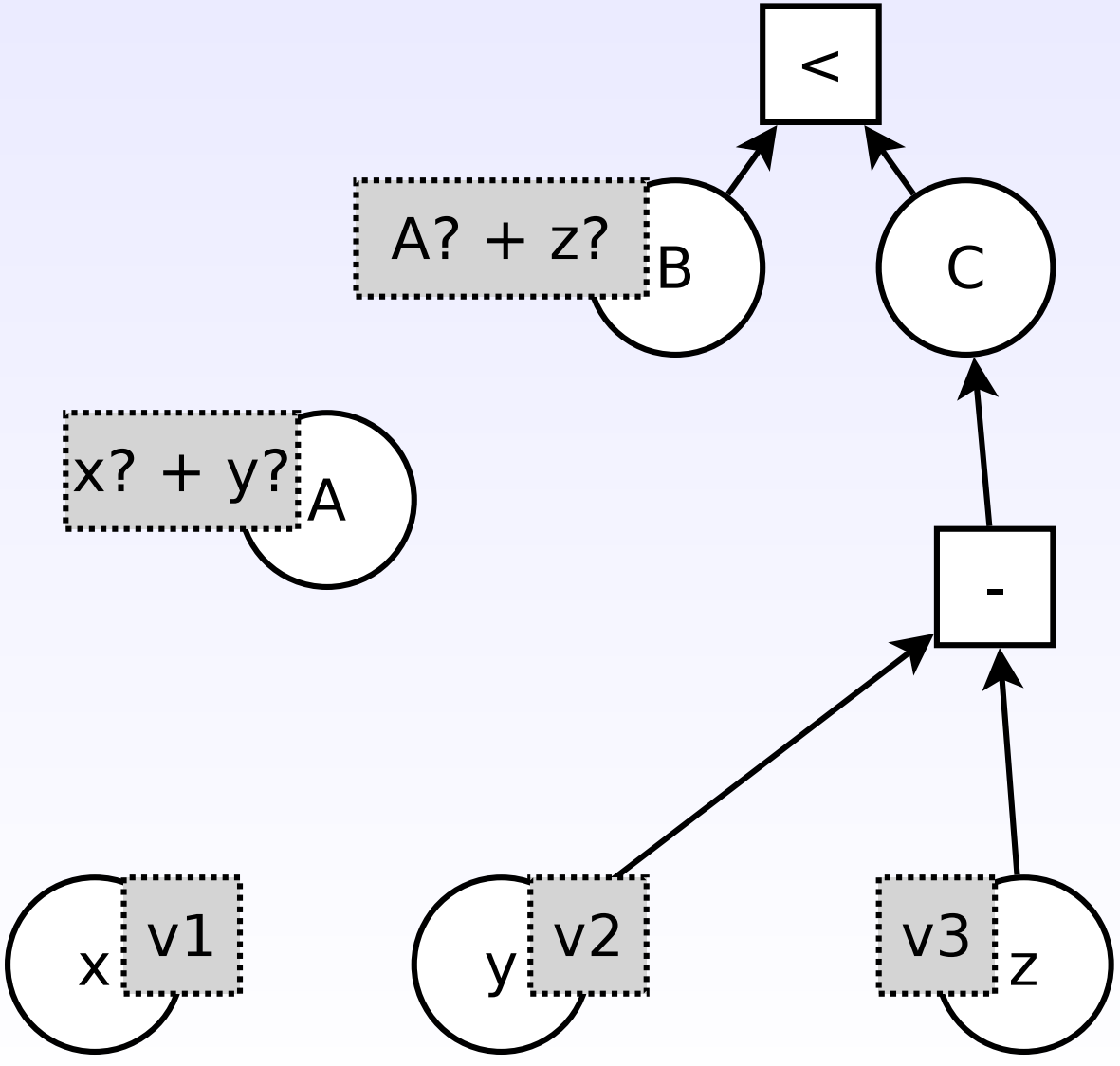
Mapping to solver-specific constraints



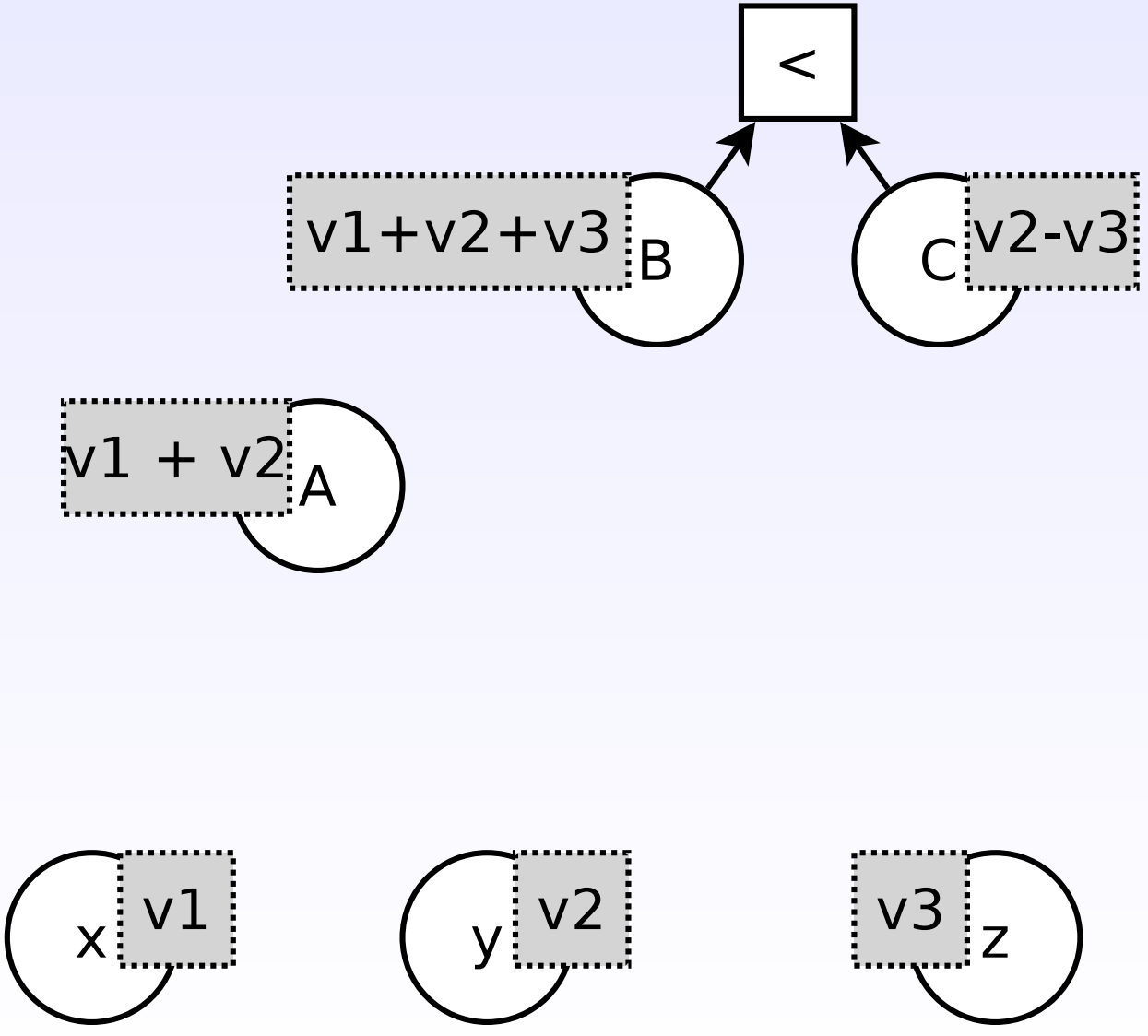
Mapping to solver-specific constraints



Mapping to solver-specific constraints



Mapping to solver-specific constraints



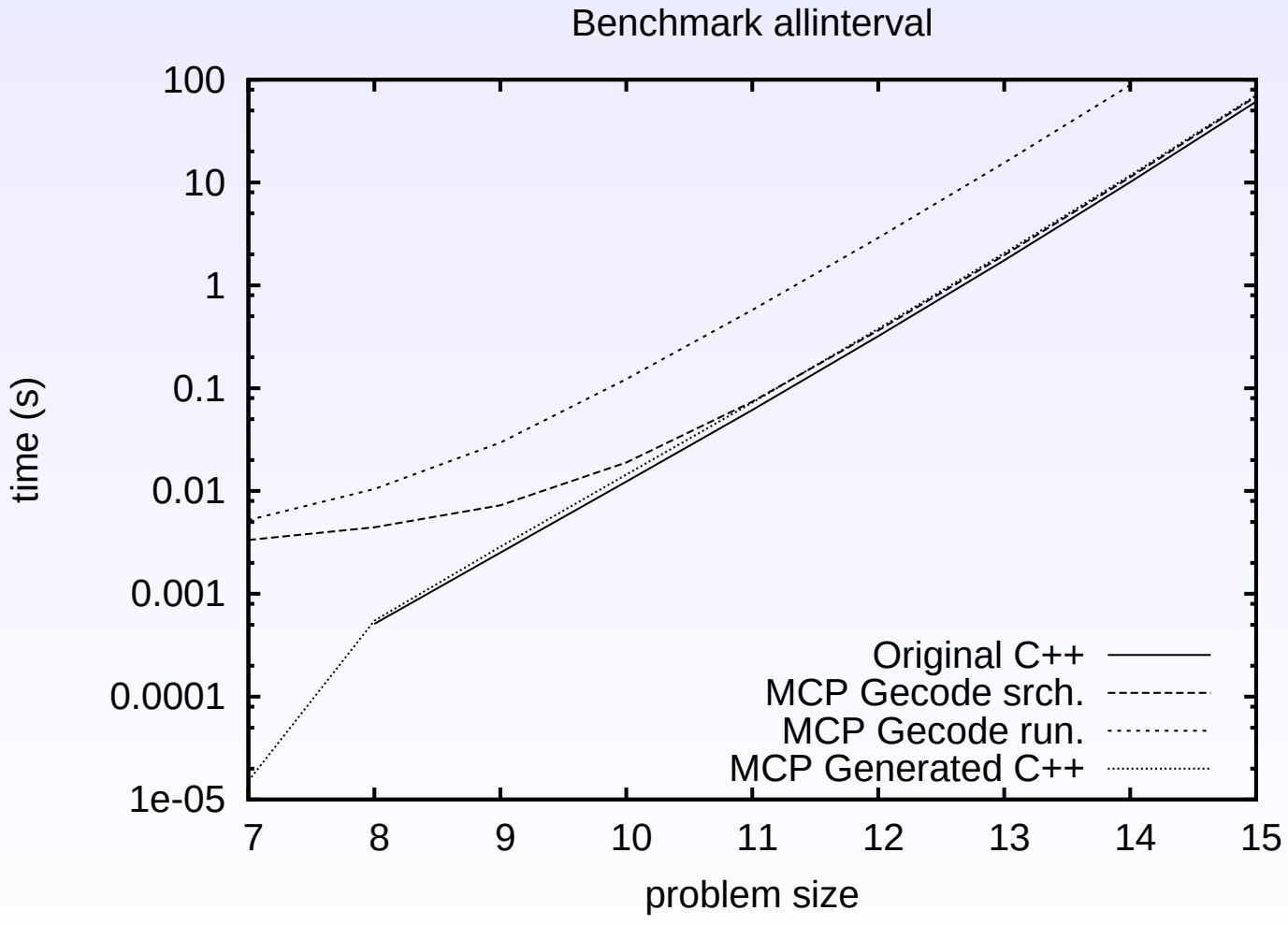
Mapping to solver-specific constraints

"Linear" is not only possible annotation:

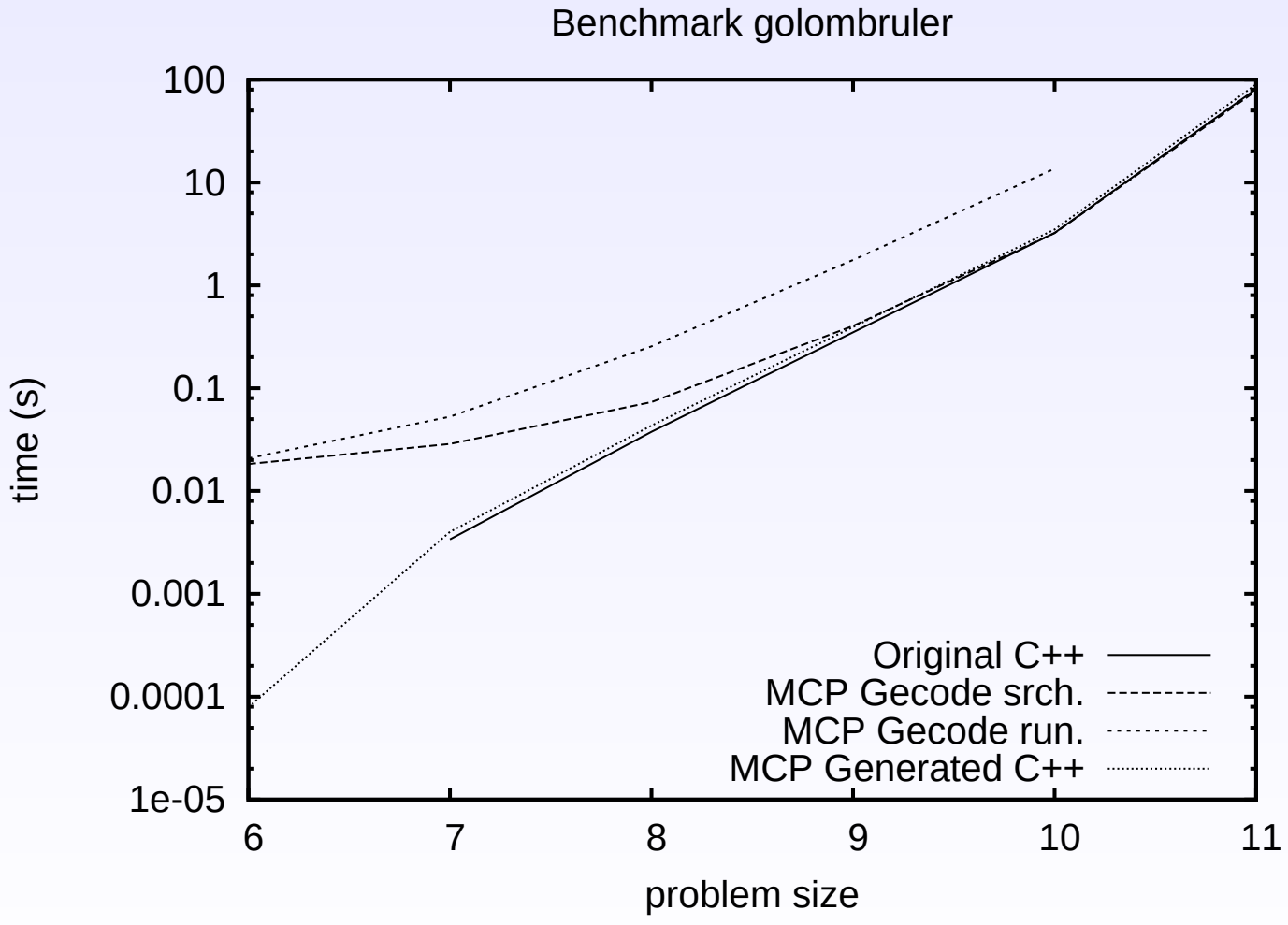
Supported annotations

- Sizes of array variables
- Constant values (integers, arrays, booleans)
- Conditionals
- ...

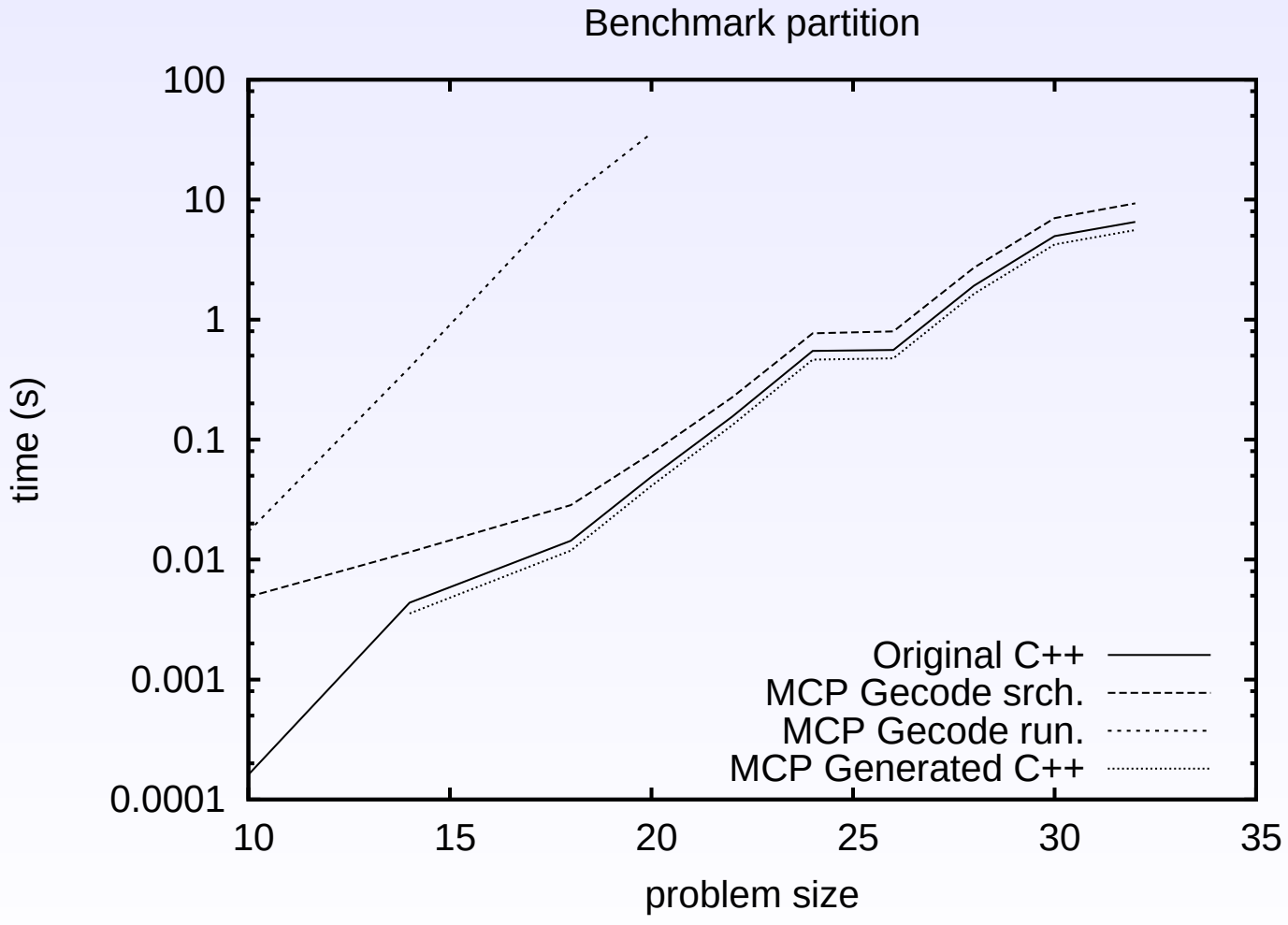
Evaluation


Benchmark allinterval
time (s)
100
10
1
0.1
0.01
0.001
0.0001
1e-05
7
8
9
10
11
12
13
14
15
problem size
Original C++
MCP Gecode srch.
MCP Gecode run.
MCP Generated C++

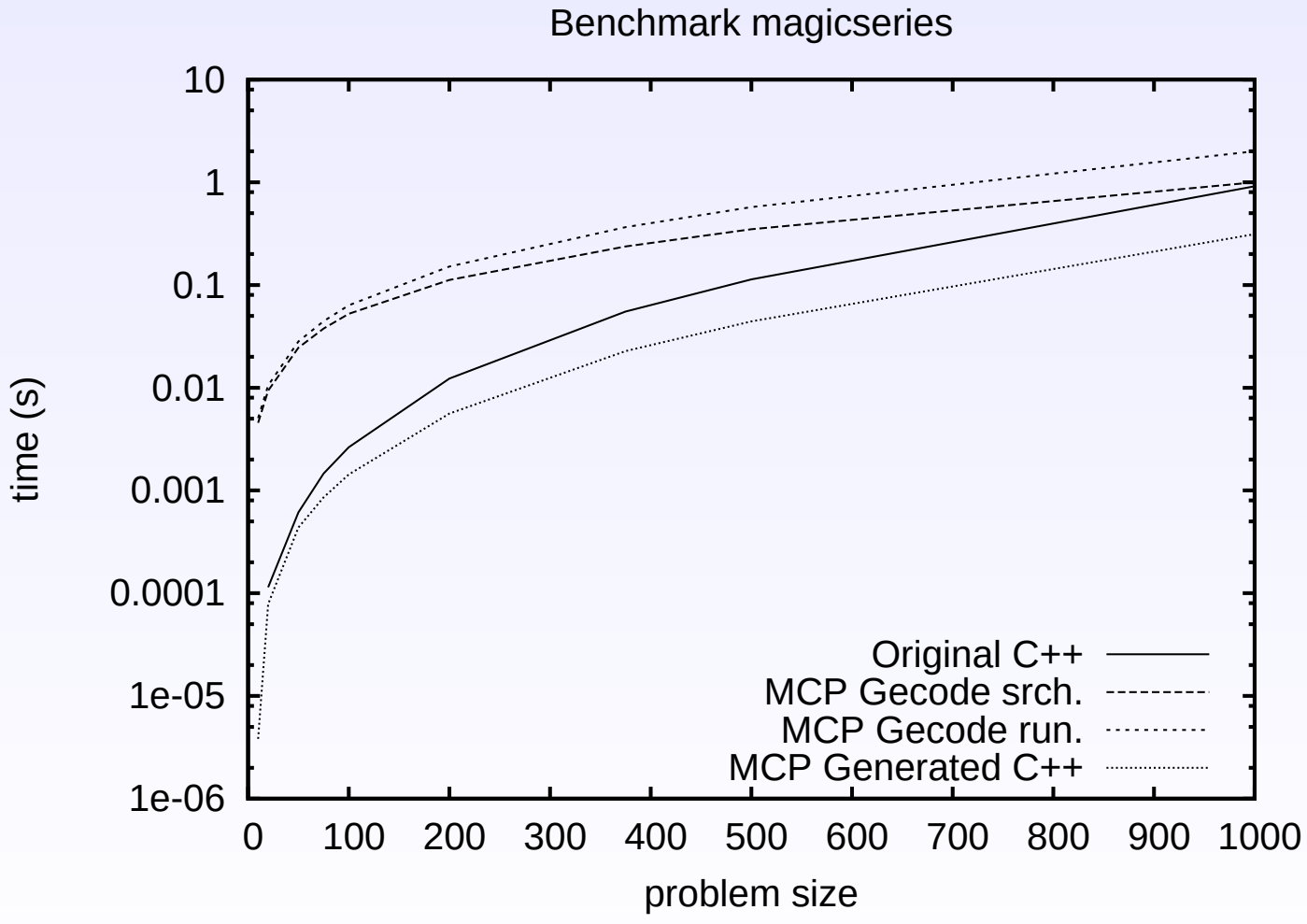
Evaluation


Benchmark golombruler
time (s)
100
10
1
0.1
0.01
0.001
0.0001
1e-05
6
7
8
9
10
11
problem size
Original C++
MCP Gecode srch.
MCP Gecode run.
MCP Generated C++

Evaluation


Benchmark partition
time (s)
100
10
1
0.1
0.01
0.001
0.0001
10
15
20
25
30
35
problem size
Original C++
MCP Gecode srch.
MCP Gecode run.
MCP Generated C++

Evaluation


Benchmark magicseries
time (s)
10
1
0.1
0.01
0.001
0.0001
1e-05
1e-06
0
100
200
300
400
500
600
700
800
900
1000
problem size
Original C++
MCP Gecode srch.
MCP Gecode run.
MCP Generated C++

Future work

Future work

- Extend system to labelling and search
- Code generation for search
- Further optimizations
- More benchmarks

The end

Any questions?