Expressive Models for Monadic Constraint Programming

Pieter Wuille  Tom Schrijvers

ModRef’10
St. Andrews, September 6, 2010
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, ...

---

### Introduction

### The FD-MCP Language

### Translation process

### Evaluation

### Future work
Expressions: example

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

\[
\text{model} = \text{exists } \ x -> \text{do}
\]
\[
\begin{align*}
    x & @> 5 \\
    x & @< 10 \\
    x*x & @= 49 \\
    \text{return } x
\end{align*}
\]

-- request a variable \(x\)
-- state that \(x>5\)
-- state that \(x<10\)
-- state that \(x*x=49\)
-- return \(x\)
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 \text{exists}
Example ($x > 5 \land x < 10 \land x^2 = p$)

```plaintext
model p = exists $ \forall 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 \((a + b + c + d = 10 \land a^2 + b^2 + c^2 + d^2 = 30)\)

\[
\text{model} = \exists \ arr \rightarrow \begin{aligned}
&\text{size} \ arr @= 4 \\
&\text{csum} \ arr @= 10 \\
&\text{csum} (\text{cmap} (\ x \rightarrow \ x \times x) \ arr) @= 30 \\
&\text{return} \ arr
\end{aligned}
\]
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 \((\land)\) true \(c\)
- **csum** \( c : \) cfold \((+)\) 0 \(c\)
- **count** \( v c : \) cfold \((p i \rightarrow p + (i = v))\) \(c\)
- \(\ldots\)
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

- 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

\[ x + y + z \triangleleft z - y \]

\( \text{Less} \ (\text{Plus} \ x \ (\text{Plus} \ y \ z)) \ (\text{Minus} \ z \ y) \)
Expression tree simplifications

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

- $X + 0 \rightarrow X$
- $X - X \rightarrow 0$
- $X + X \rightarrow 2X$
- $(a + (b + X)) \rightarrow (a+b) + X$
- 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

\[ x + y + x \preceq z - y \]
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

A? + z? \rightarrow B < C

x? + y? \rightarrow A - C

x y z
Mapping to solver-specific constraints
Mapping to solver-specific constraints

\[ v_1 + v_2 + v_3 \] (B) < \[ v_2 - v_3 \] (C)

\[ v_1 + v_2 \] (A)
“Linear” is not only possible annotation:

Supported annotations

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

Benchmark allinterval

- Original C++
- MCP Gecode search
- MCP Gecode run
- MCP generated C++
Evaluation

Benchmark golombruler

- Original C++
- MCP Gecode srch.
- MCP Gecode run.
- MCP Generated C++

The graph shows the performance of different implementations on the Golombruler benchmark. The x-axis represents the problem size, while the y-axis shows the time in seconds. The graph compares the performance of the original C++ implementation with the translated implementations using the MCP language and Gecode framework.
Evaluation

Benchmark partition

- Original C++
- MCP Gecode search
- MCP Gecode run
- MCP Generated C++

X-axis: problem size
Y-axis: time (s)
Evaluation

Benchmark magicseries

- Original C++
- MCP Gecode search
- MCP Gecode run
- MCP Generated C++
Future work

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