
The Constraint Imperative Programming Language Turtle

Martin Grabmüller

`magr@cs.tu-berlin.de`

Fachgebiet Übersetzerbau und Programmiersprachen

Fakultät IV – Elektrotechnik und Informatik

Technische Universität Berlin



Introduction

- Constraint imperative programming
- Language design: base language and constraint extensions
- CIP with Turtle: an example
- Implementation
- Future works



Constraint Imperative Programming (1)

- Imperative programming is statement oriented
- Constraint programming is declarative
- Constraint imperative programming (CIP) combines these programming paradigms
- Goal: combine advantages of both



Constraint Imperative Programming (2)

- Kaleidoscope (object oriented, with constraints)
 - Object attributes can be constrained
 - User-defined *constraint constructors* can be defined on objects
- Alma-0 (procedural, nondeterministic)
 - Failing/successful statements
 - Backtracking
- Constraint libraries (constraints as abstract data types)
 - Support creation of constraints and searching
 - Pre-processors for extended imperative languages



Base Language (1)

- Variables and assignments
- Conditionals and loops
- Subprograms
- Higher-order functions
- Lists, arrays and algebraic data types
- Parameterized modules



Base Language (2)

```
module main;
```

```
import io, listsort<string>, compare;
```

```
fun main (argv: list of string): int
```

```
    var l: list of string := io.get (io.open ("main.t"));
```

```
    io.put (listsort.sort (l, compare.cmp));
```

```
    return 0;
```

```
end;
```



Constraint Extensions

- Constrainable variables
- Constraint statements
- User-defined constraints
- Constraint solvers



Constrainable Variables

```
var x: !real := var 0;
```

- Values of constrainable variables are determined by constraints
- Are declared with special types
- Contain *variable objects* encapsulating their values (constructed by **var** expressions)
- Can thereby be shared between data structures



Constraint Statements (1)

```
require  $x = 2$  in  
    io.put (!x);    // prints "2"  
end;
```

- Constraint conjunctions are added to the stores of responsible constraint solvers
- Constraints are solved (variables are assigned)
- Statements in the statement body are executed
- Constraints are removed at the end of the body



Constraint Statements (2)

```
require  $x = 2$  and  $y \geq x$  and  $y = 0$  : weak in  
  io.put (!y);    // prints "2"  
end;
```

- Individual constraints can have different priorities
- Constraints can form *constraint hierarchies*
- Less important constraints may be violated in order to satisfy more important ones
- Constraint solvers minimize the resulting error



User-defined Constraints (1)

```
constraint between (v: !int, min: int, max: int)  
  require v >= min and v <= max;  
end;
```

- User-defined constraints make constraint conjunctions reusable
- Like primitive constraints (e.g relations like \leq), can be used in constraint statements
- Parameters can be constrainable variables or values



User-defined Constraints (2)

```
constraint all_different (l: list of !int)
  while (tl l <> null) do
    var ll: list of !int := tl l;
    while (ll <> null) do
      require hd l <> hd ll;
      ll := tl ll;
    end;
    l := tl l;
  end;
end;
...
require all_different ([a, b, c]);
```

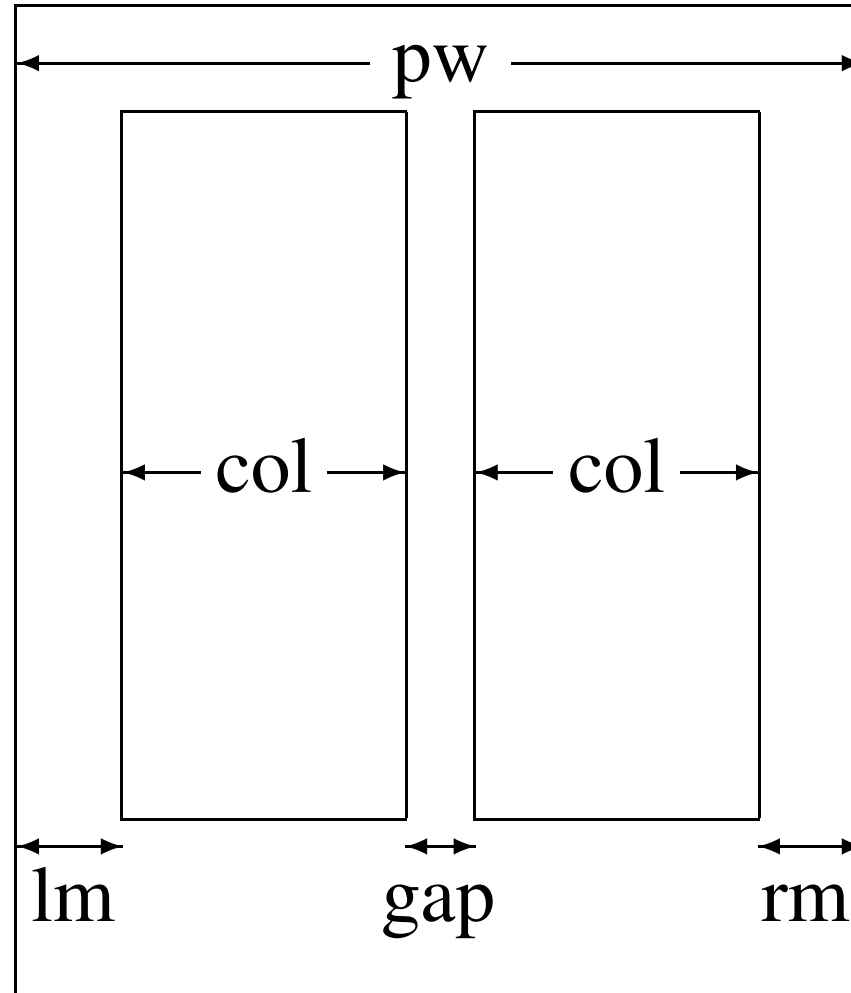


Constraint Solvers

- Manage the set of active constraints in constraint stores
- Check the satisfiability of the stores
- Calculate assignments for constrainable variables
- Raise exceptions when constraint conjunctions are not satisfiable
- New solvers can easily be integrated



Example – Layout Calculation



$$pw = lm + 2col + gap + rm$$



Example Program

```
module layout;
import io;
fun main (args: list of string): int
    var lm: !real := var 0.0;
    var rm: !real := var 0.0;
    var gap: !real := var 0.0;
    var pw: !real := var 0.0;
    var col: !real := var 0.0;
    require lm = 2.0 and rm = 2.0 and pw = 21.0 and
        gap >= 0.5 and gap <= 2.0 and gap = 0.5 : medium and
        col <= 7.0 : strong and gap + lm + 2.0 * col + rm = pw in
        io.put ("lm="); io.put (!lm); io.nl ();
        io.put ("rm="); io.put (!rm); io.nl ();
        io.put ("gap="); io.put (!gap); io.nl ();
        io.put ("pw="); io.put (!pw); io.nl ();
        io.put ("col="); io.put (!col); io.nl ();
    end;
    return 0;
end;
```



Layout Calculation

- Constraints for lm , rm und pw have highest priority and must get satisfied:
 $lm = 2.0 \wedge rm = 2.0 \wedge pw = 21.0$
- gap may be at most 2.0: $gap = 2.0$
- col has to be enlarged in order to fulfill the main constraint: $col = 7.5$
- Priority of $gap = 0.5$ is weaker than $col \leq 7.0$, so the former constraint is violated
- Since $gap \leq 2.0$ is required, the constraint on col is violated, too



Implementation

- Compiler, run-time system and library
- Implementation is based on imperative stack machine
- Constraint solvers are separate program components
- Constraints are passed at run-time in symbolic form to constraint solvers
- Automatic storage management for data and constraints



Future works

- Integration of stronger constraint solvers
- Using constraints for directing the control flow
- Nondeterminism, e.g for search problems
- General research on multiparadigm languages



Thank you

