The Constraint Imperative Programming Language Turtle

Martin Grabmüller

Fachgebiet Übersetzerbau und Programmiersprachen
Fakultät IV Elektrotechnik und Informatik
Technische Universität Berlin
magr@cs.tu-berlin.de

Abstract. The goal of declarative programming is to provide languages and implementations which let the programmer write programs by specifying what the properties of the desired solutions should be. In imperative languages, the calculation steps leading to the solutions must be programmed explicitly. The programming language TURTLE combines traditional imperative language constructs and declarative constraints. This blending of paradigms is called *constraint imperative programming* in literature, and TURTLE is one instance of this multiparadigm approach.

1 Introduction

Constraints are conditions in a mathematical sense and are well suited for expressing restrictions on values and relations between objects. Constraint solvers are program components which can recognize violations of constraints and may be able to re-establish them by modifying the objects or adapting the relations participating in these constraints. Constraint programming [1] is a declarative programming paradigm based on constraints, enriched with mechanisms for controlling the constraint solving process.

The advantages of declarative programming have been generally acknowledged, but imperative programming languages are still the most widely used programming languages in industry. They still have the advantages of faster implementations and better tool support.

The programming language TURTLE [2] combines traditional imperative language constructs and declarative constraints. This blending of paradigms is called *constraint imperative programming* in literature [3].

Other work on integrating constraints and imperative programming includes Kaleidoscope [4], an object-oriented language which allows to place constraints on object attributes, and the Alma project, which aims at designing a language which integrates traditional procedural programming with constraints and nondeterminism [5].

This paper is organized as follows: in Sect. 2, we introduce the principles behind the design of TURTLE. Section 3 presents a problem which can be well solved by constraint imperative programming and show the solution in TURTLE. The implementation of TURTLE is sketched in Sect. 4 and Sect. 5 describes some areas of future work and finally concludes.

2 Language Design

The design of the language was started by first defining a base language which includes the basic elements of imperative programming languages: variables, assignments, conditionals, loops and subroutines. Additionally, this base language includes some language features known from functional programming, such as polymorphic data types, algebraic data types and higher-order functions.

In order to support constraint programming, TURTLE was then extended with four new language elements: *constrainable variables*, *constraint statements*, *user-defined constraints* and *constraint solvers*. This section will describe each of these features in detail.

2.1 Constrainable variables

Constrainable variables are not set by assignment statements like normal variables, but instead their values are determined by constraints. Variables must be declared as being constrainable by assigning them special types. Because of the separation between these two kinds of variables, the use and semantics of variables can be deduced from their types. The following code fragment declares a constrainable variable of type *real* and initializes it to the value 0.

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

The values of constrainable variables are not stored directly in the variable, but instead in a *variable object*. This object contains additional information for maintaining constraints in addition to the value. Because of this indirection, constrainable variables can be shared between different data structures, so that constraints can be placed on several data objects at once. To fetch the value of a constrainable variable from its variable object, a dereferencing operation is necessary, which is expressed by the prefix operator **!**:

var *y*: real := **!***x*;

The separation of normal and constrainable variables has the advantage that the purpose of each variable is visually apparent and can be verified by the type checker in the compiler. The disadvantage is less flexibility in the use of variables.

2.2 Constraint statements

Constraints are placed on constants and constrainable variables with *constraint statements* (see Fig. 1). A constraint statement consists of a constraint conjunction and a sequence of statements. The constraints are satisfied by assigning suitable values to their variables as long as the statements in the body are executed. The constraints in the conjunctions can individually get assigned priorities, so-called *constraint strengths*. These priorities specify the importance of the respective constraints and the solver is free to violate less important constraints

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

Fig. 1. Constraint statement

```
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]);
```

Fig. 2. User-defined constraint

(such as *weak*) if that is necessary to satisfy stronger ones. When no strength is given, the strongest strength is assumed.

Additionally, the body of a constraint statement can be omitted. In this case, the constraints stay active as long as the variables mentioned in the constraints do exist. For global variables, constraints will stay in effect until the program terminates.

2.3 User-defined constraints

User-defined constraints abstract over constraints, just like subroutines abstract over statements. Constraint conjunctions, which are needed in several constraint statements throughout the program, can be collected and be referenced by a name in order to access the conjunction without the need to specify all individual constraints wherever they are needed. Figure 2 shows a user-defined constraint which constrains all elements of a list of integers to be pairwise different. Whenever this user-defined constraint appears in a constraint statement (as illustrated in the last line of the example), the *all_different* constraint will be executed, thus placing inequality constraints on each pair of list elements. Of course, user-defined constraints can be called in more than one place with different parameters, thus increasing modularity and reusability of constraints.

User-defined constraints can thus be used to create powerful and generally useful constraint libraries to be included in several application programs.

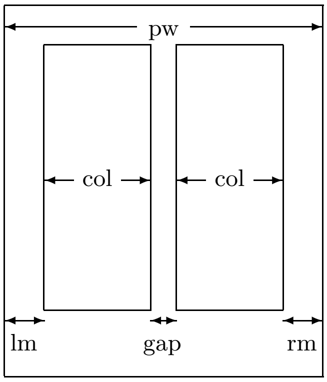

Fig. 3. Layout problem

2.4 Constraint solvers

The constraints specified by constraint statements must be checked, and assignments for constrainable variables must be calculated. This is the task performed by one or more *constraint solvers* integrated with the programming language implementation. The solvers maintain all constraints in *constraint stores*. The number and kind of the used constraint solvers is not specified by the language, it depends on the problems which are to be solved. The TURTLE system implements a generic interface between compiled code and the constraint solvers.

The current implementation of TURTLE includes a solver for finite domain constraints and one for real numbers, and other solvers (for other domains or stronger solvers) can easily be integrated.

3 Example

For illustrating constraint imperative programming with TURTLE, the following problem is to be solved: for a typesetting system, the layout of a two-column page is to be calculated (see Fig. 3). The page has width *pw*, and on the left and right of the texts are the left margin *lm* and right margin *rm*. Two equally-sized columns of text (*col*) are separated by a *gap*. The following equation is expected to hold:

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

The page width is constrained to be equal to 21 and the margins must both be 2. Additionally, the size of the gap must lie in the interval $[0.5, 2]$, but should be as narrow as possible. The size of the column should be smaller than 7, as long as the other constraints are not violated.

Figure 4 shows how these constraints are specified in TURTLE. First, constrainable variables of all variables of the problem are created and initialized to 0, because this initial value is not needed here. Then the constraint statement enforces the constraints informally specified above. In the body of the constraint statement, the values for all the variables are printed.

```
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;
```

Fig. 4. Layout program

The solving process determines values for the variables as follows:

- Because the 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, so $gap = 2.0$.
- The *medium* constraint on *gap* to keep it as narrow as possible cannot be fulfilled.
- *col* has to be enlarged in order to fulfill the main constraint: $col = 7.5$.
- Priority of $gap = 0.5$ is weaker than $col <= 7.0$, so the former constraint is violated.
- Since $gap <= 2.0$ is required, the constraint on *col* is violated, too.

Finally, the following assignments to the variables are calculated:

$$lm = rm = 2 \wedge pw = 21 \wedge gap = 2 \wedge col = 7.5$$

4 Implementation

The implementation of the TURTLE language consists of a compiler, a run-time system (including constraint solvers and a garbage collector) and a library of

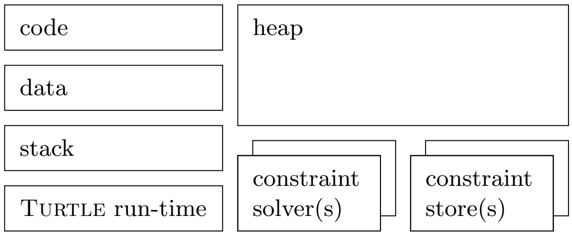

Fig. 5. TURTLE program components

modules, written in TURTLE (operating system interfaces, networking, data type manipulation, etc.).

The execution model is based on an imperative stack machine, extended to handle constraints. The various components of a running TURTLE program are shown in Fig. 5. The *code* section contains the program code of the TURTLE program, *data* contains global data of the TURTLE program as well as the run-time system. The *stack* is only used by the run-time system, because activation records and local data of TURTLE functions is kept in the *heap*, which contains all dynamically allocated data structures. The heap is managed by a copying garbage collector. One or more constraint solver(s) with associated constraint store(s) maintain active constraints and perform the constraint solving task.

Constraints are internally represented in a symbolic form, which is created when a constraint is enforced by a constraint statement. This symbolic representation is passed to the constraint solver which tries to satisfy it together with already active constraints. If the constraint cannot be satisfied, an exception is raised. Exceptions can be handled by the TURTLE program.

The TURTLE implementation described here has been used to implement various example programs, ranging from short constraint imperative demonstration programs to larger imperative programs (a frontend of a TURTLE-compiler including a scanner and a parser as well as a functional web server).

5 Future Works and Conclusion

One problem with the current implementation of TURTLE is the weakness of the constraint solvers. The integration of stronger constraint solvers is necessary to make the system practical for larger problems. Another direction for future research is to use constraints not only for calculation of values, but also for directing the control flow of the program. The inclusion of nondeterminism is a possible approach and would be well suited for solving search problems. The work on constraint imperative programming is only a small fragment of the larger area of multiparadigm programming, which also includes the combinations of functional, logic, imperative and other paradigms, such as object- or aspect-oriented programming.

In this paper, we presented the design of the constrain imperative programming language TURTLE. The design is based on a typical imperative programming language enriched with four language features for constraint programming: constrainable variables, constraint statements, user-defined constraints and constraint solvers. A programming system, consisting of a compiler, a run-time system and library modules has been implemented. The resulting programming language is practical and several imperative as well as constraint imperative example programs of varying size have been developed to demonstrate the different language features.

References

1. Marriot, K., Stuckey, P.J.: Programming with Constraints. MIT Press (1998)
2. Grabmüller, M.: Constraint Imperative Programming. Diploma Thesis, Technische Universität Berlin (2003)
3. Freeman-Benson, B.N.: Constraint Imperative Programming. PhD thesis, University of Washington, Dept. of Computer Science and Engineering (1991)
4. Lopez, G., Freeman-Benson, B., Borning, A.: Kaleidoscope: A constraint imperative programming language. In Mayoh, B., Tyugu, E., Penjaam, J., eds.: Constraint Programming: Proc. 1993 NATO ASI Parnu, Estonia, Springer (1994) 305–321
5. Apt, K.R., Schaerf, A.: The Alma project, or how first order logic can help us in imperative programming. In: Correct System Design. Number 1710 in LNCS, Springer (1999) 89–113