

Constraint Imperative Programming with Higher-order Functions

Martin Grabmüller

`magr@cs.tu-berlin.de`

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Franklinstr. 28/29, 10587 Berlin, Germany

Abstract. In constraint programming, programs are written by mainly specifying the properties which must hold for a solution of a given problem. In contrast, imperative programming languages require that the programmer explicitly specifies each state transition a program must perform in order to calculate the solution. This paper describes a programming style which consists of these both views as well as on higher-order functional programming.

1 Introduction

Programming languages can broadly be classified by two main categories: declarative (or stateless) and imperative (or stateful) programming languages. Declarative languages aim at making programming mainly a task of specification; it is the job of the programming system to derive suitable algorithms and strategies for making the specification executable. Imperative languages are used to write algorithms explicitly as transformation on the program state, which consists of program variables and the state of the external world. These different paradigms are not only results of different implementations, they reflect different views on programming and on problem-solving with computers. Declarative languages concentrate on writing *what* to do, whereas imperative languages view programming as specifying *how* to do it.

Constraint programming and functional programming are prominent members of the declarative programming paradigm and have proven to have several advantages over traditional imperative languages. Their mathematical foundations and well-defined semantics offer many possibilities for program transformation and optimization as well as for correctness proofs. On the other hand, imperative languages are still more efficient for many applications and are the most widely used implementation languages. Research on programming language design has therefore led to the integration of these two programming paradigms, in order to combine the advantages of both. The term *constraint imperative programming* (CIP) has been introduced in the literature to describe the combined paradigm of constraint-based and imperative programming [1]. Constraint imperative languages have also been extended with language features mainly known

Martin Grabmüller: Constraint Imperative Programming with Higher-order Functions, Draft paper, July 2003.

from functional programming. The language TURTLE [2] combines imperative control structures, variables and data structures with constraint solving capabilities as well as with higher-order functions, algebraic data types and polymorphic modules. The programming style evolving from the combination of constraint, imperative and functional programming lets the programmer choose whichever of the available language features is best suited for a given task.

This paper is organized as follows. Section 2 gives a brief description of the programming language TURTLE. In Sect. 3, the higher-order constraint imperative programming style used for programming with TURTLE will be introduced. First, a simple program is presented which solves a particular problem, then this program will be generalized using imperative and functional modularization techniques. The advantages of this programming style will also be discussed in that section. Finally, Sect. 4 will summarize this paper and refer to related work.

2 The Programming Language Turtle

This section gives a short survey on the programming language TURTLE [2], used to write the example programs in Sect. 3. The language was designed by starting with an imperative base language, extended with higher-order functions and a rich type system. Then, several language extensions for constraint programming were added. We will first describe the base language and then discuss the constraint programming features.

TURTLE provides all control structures known from traditional imperative languages: conditionals, loops, functions (and procedures) and assignment statements. Additionally, TURTLE supports a rich set of data types, including integers, reals, booleans, strings, characters, arrays, lists and tuples. Higher-order functions, which can receive functions as parameters and can return functions as their value are provided for functional programming. TURTLE also has a module system for encapsulating the declaration of functions, variables and data types using explicit import/export relations between modules. Modules can be parameterized by data types and functions can be defined in terms of these parameters, therefore yielding polymorphic functions. The TURTLE implementation comes with a set of library modules which make extensive use of this feature, e.g. for providing functions to handle lists of arbitrary element types.

The imperative/functional base language is extended with constraint solving capabilities by the addition of four language elements, to be described next. The syntax and the usage of these language extensions will be illustrated in the next section.

Constrainable variables are special variables, introduced by data type annotations, e.g. a constrainable integer variable is declared with type `! int`. Normal variables are given values by assignments, whereas the values of constrainable variables are determined by placing constraints on them. Since constrainable variables not only hold a value but also need to store additional information for use with the constraint solvers, they are actually represented by *variable objects*,

which are explicitly created and must be dereferenced to obtain the variable's value.

Constraint statements are block-structured statements which consist of (1) a constraint conjunction and (2) of a sequence of statements, called the body. When a constraint statement starts to execute, the constraints in the constraint conjunction are added to the constraint store and the built-in constraint solver tries to satisfy the constraints by assigning suitable values to the constrainable variables appearing in the constraints. When the solving process is successful, the statements in the body are executed. The variables remain bound to these values during the execution of the body, and the constraints are removed from the constraint store when the statement is left.

If the constraint solver detects that the constraints are not satisfiable, an exception is raised which must either be handled by the program or otherwise terminates execution.

User-defined constraints abstract over constraints similar to functions, which abstract over individual expressions or statements. User-defined constraints can contain arbitrary statements, but their main purpose is to place constraints on one or more of their parameters. When a user-defined constraint invocation appears in a constraint conjunction, the body of the user-defined constraint is executed. An example for such a user-defined constraint is *all_different(L)*, which constrains all elements of the list *L* to be pairwise different. This constraint will be used in the example program in Sect. 3.

Constraint solvers are built into the run-time system of TURTLE and are responsible for maintaining their associated constraint stores. Whenever constraints are added to the store, the solvers must satisfy their stores by calculating assignments for the constrainable variables. The addition and removal of constraints to and from the stores as well as the solving process is initiated by the execution of constraint statements.

Constraint statements in TURTLE allow to specify how important individual constraints in a conjunction are by so-called *strength annotations*. The constraint solvers will try to satisfy the most important constraints, even if that means that less important ones will be violated. This treatment of preferential constraints is called *constraint hierarchies* [3] and is very useful for dealing with over-constrained problems.

3 Constraint Imperative Programming with Higher-order Functions

This section will give a detailed example on how programming is done with a higher-order constraint imperative programming language. First, we will explain a simple program which solves a specialized problem, then the program will be generalized using functional and imperative language constructs.

3.1 Basic Constraint Imperative Programming

Figure 1 presents a constraint imperative solution (written in TURTLE syntax) to the famous crypto-arithmetic puzzle $SEND+MORE=MONEY$. The task is to assign a digit between 0 and 9 to each letter, where all letters must have pairwise distinct digits and the number represented by $SEND$ plus the number represented by $MORE$ must give the number represented by $MONEY$.

After the module header, which states the name of the module and the imported module *io*, two user-defined constraints are defined. The user-defined constraint *all_different* has a list of constrainable integers as its input and places inequality constraints on each pair (L_i, L_j) of list elements:

$$all_different(L) \equiv \bigwedge_{i \neq j} L_i \neq L_j$$

The constraint *all_different* illustrates how user-defined constraints are typically used: the complex constraint represented by the user-defined constraint is decomposed into simpler constraints, in this case into a number of inequalities. Since the constraints are added one after the other to the constraint store, they are handled as a conjunction of simple constraints.

The user-defined constraint *domain* is used to restrict the domain of a constrainable variable by specifying a lower and upper bound for the allowed values.

$$domain(v, min, max) \equiv min \leq v \leq max$$

The main program first declares the constrainable variables, one for each letter of the puzzle. The variables are all initialized to the value 0, but the actual value is irrelevant in this example, because this value will be overridden immediately with a constraint statement. The expressions `var 0` creates a variable object and initializes its value slot to 0.

In the constraint statements, the two user-defined constraints are used to specify the domain of all constrainable variables (0 to 9) and to ensure that all variables get different values. The last constraint in the constraint conjunction specifies the relation between the individual variables as required by the puzzle.

The body of the constraint statement simply outputs the values calculated for the variables. Because the constrainable variables hold variable objects, their values must be extracted using the `!` operator.

3.2 Generalization

A generalized version of the program in Sect. 3.1 is the crypto-arithmetic puzzle solver we describe in this section. This program is capable of solving puzzles of arbitrarily many rows, where each row can have arbitrarily many columns. The program is presented in several parts, where each part will be commented on individually.

The generalized program makes extensive use of the TURTLE standard library. Besides the input/output module, which was also used in Fig. 1, modules

```

module sendmore;
import io;

constraint all_different (l: list of !int)
  while tl l  $\neq$  null do
    var ll: list of !int  $\leftarrow$  tl l;
    while ll  $\neq$  null do
      require hd l  $\neq$  hd ll;
      ll  $\leftarrow$  tl ll;
    end;
    l  $\leftarrow$  tl l;
  end;
end;

constraint domain (v: !int, min: int, max: int)
  require v  $\geq$  min and v  $\leq$  max;
end;

fun main(args: list of string): int
  var s: !int  $\leftarrow$  var 0;
  var e: !int  $\leftarrow$  var 0;
  var n: !int  $\leftarrow$  var 0;
  var d: !int  $\leftarrow$  var 0;
  var m: !int  $\leftarrow$  var 0;
  var o: !int  $\leftarrow$  var 0;
  var r: !int  $\leftarrow$  var 0;
  var y: !int  $\leftarrow$  var 0;
  require domain (s, 0, 9) and domain (e, 0, 9) and domain (n, 0, 9) and
    domain (d, 0, 9) and domain (m, 1, 9) and domain (o, 0, 9) and
    domain (r, 0, 9) and domain (y, 0, 9) and
    all_different ([s, e, n, d, m, o, r, y]) and
    (s * 1000 + e * 100 + n * 10 + d) +
    (m * 1000 + o * 100 + r * 10 + e) =
    (m * 10000 + o * 1000 + n * 100 + e * 10 + y)
  in
    io.put ("s = "); io.put (!s); io.nl ();
    io.put ("e = "); io.put (!e); io.nl ();
    io.put ("n = "); io.put (!n); io.nl ();
    io.put ("d = "); io.put (!d); io.nl ();
    io.put ("m = "); io.put (!m); io.nl ();
    io.put ("o = "); io.put (!o); io.nl ();
    io.put ("r = "); io.put (!r); io.nl ();
    io.put ("y = "); io.put (!y); io.nl ();
  end;
end;

```

Fig. 1. SEND+MORE=MONEY Program

for manipulating strings, integers and lists of several element types are used. The modules *lists* and *listmap* are imported several times with different type parameters. The exported functions of these modules will be described below when they are used.¹

```

module crypto;
import io, strings, lists<string>, lists<char>, lists<!int>, compare,
    listsort<char>, listmap<string, !int>, listmap<char, !int>,
    ints;

```

The general puzzle solver used the same user-defined constraints *all_different* and *domain* as the program in Sect. 3.1, therefore, they are left out of this example.

The following function is a utility routine for calculating all letters occurring in all strings in a list of strings, with duplicates removed. The function *letters* illustrates the use of functions defined in other modules and overload resolution in combination with higher-order functions. The function *index* (exported from module *lists*) expects as parameters a value and a list of the same element type as the value type, and a comparison function. This function is expected to return a value less than zero if its first argument is less than the second, equal to zero if the arguments are equal and a value greater than zero otherwise. Since the module *compare* exports several functions *cmp* (one for each basic data type), the overload resolution mechanism must find out which of these matches the type expected by function *index*, which in turn depends on the type of the parameters to *index*.

```

fun letters (s: list of string): list of char
  var c: list of char ← [];
  var i: int;
  var t: string;
  while s ≠ null do
    i ← 0;
    t ← hd s;
    while i < sizeof t do
      if lists.index (t[i], c, compare.cmp) < 0 then
        c ← t[i] :: c;
      end;
      i ← i + 1;
    end;
    s ← tl s;
  end;
  return c;
end;

```

¹ Note that **string** is a reserved word whereas the other type names are normal identifiers. Therefore the different typography.

Function *letter_variable* looks up character *c* in a list of letters and returns the variable in list *vars* with the same index. It is used to find variable objects associated with individual letters of a puzzle.

```

fun letter_variable (c: char, letters: list of char, vars: list of !int): !int
  while letters  $\neq$  null do
    if hd letters = c then
      return hd vars;
    end;
    letters  $\leftarrow$  tl letters;
    vars  $\leftarrow$  tl vars;
  end;
  return var 0;
end;

```

The user-defined constraint *constrain* is responsible for placing constraints on the constrainable variables corresponding to the individual letters such that the puzzle is correctly specified. Since it is a rather long function, we will split the discussion into several parts.

The input to the constraint is a list of strings which represent the puzzle. The last element of the list is the result of the addition, and the other elements are the operands. The parameter *letters* is a list of all letters of the puzzle whereas *letter_vars* is a list of the corresponding variable objects. The variable *max_len* stores the length of the longest operand string. The higher-order function *map* from module *lists* is used to calculate this maximum value and for generating as many variable objects as there are rows in the puzzle. This list of variables is stored in *row_vars*. *pow* is calculated to be the number of values any of the rows can take.

```

constraint constrain (strs: list of string, letters: list of char,
  letter_vars: list of !int)
  var max_len: int  $\leftarrow$  0;
  var row_vars: list of !int  $\leftarrow$ 
    listmap.map (fun (s: string): !int
      if sizeof s > max_len then
        max_len  $\leftarrow$  sizeof s;
      end;
      return var 0;
    end, strs);
  var pow: int  $\leftarrow$  ints.pow (10, max_len);

```

The next part of the user-defined constraint states the fundamental constraints that the domain of the variables representing the letters is the interval $[0, 9]$ and that the letter variables must be pairwise different. The row variables are constrained to be non-negative and to be less than the maximum row value

calculated above. This is an optimization to be exploited by the constraint solver, because this information is redundant.

```

var rv: list of !int ← row_vars;
lists.foreach (fun (l: !int)
               require domain (l, 0, 9);
               end, letter_vars);
require all_different (letter_vars);
rv ← row_vars;
while rv ≠ null do
  require hd rv ≥ 0 and hd rv < pow;
  rv ← tl rv;
end;

```

The nested loop in the following piece of code generates the constraints which state the relation between the individual letters of a row and the row variable. The outer loop traverses the list of strings and the list of rows in parallel, the inner loop runs through the individual strings, generating the constraints describing the order of magnitude of each letter.

```

rv ← row_vars;
while strs ≠ null do
  var s: string ← hd strs;
  strs ← tl strs;
  var i: int ← 1;
  var run: !int ← letter_variable (s[0], letters, letter_vars);
  require run > 0;
  require run < pow;
  while i < sizeof s do
    var r: !int ← var 0;
    require r ≥ 0 and r < pow;
    var lv: !int ← letter_variable (s[i], letters, letter_vars);
    require run * 10 + lv = r;
    run ← r;
    i ← i + 1;
  end;
  require run = hd rv;
  rv ← tl rv;
end;

```

For example, the puzzle

$$\begin{array}{r}
 A\ B \\
 +\ C\ D \\
 +\ E\ F \\
 \hline
 =\ G\ H
 \end{array}$$

will generate the following constraints

$$A * 10 + B = r_1 \wedge C * 10 + D = r_2 \wedge E * 10 + F = r_3 \wedge G * 10 + H = r_4$$

for the three rows, where r_i are the row variables.

The last step is the generation of the constraint between all the rows. Since the number of rows is not known in advance, the constraint is actually built up from several such constraints.

Reconsider the example puzzle above and the constraints which were generated for the rows (excluding the last, which is handled specially because it represents the result).

$$r_1 + r_2 = a_1 \wedge a_1 + r_3 = a_2$$

The variables a_i are introduced because the n -ary addition is represented as $n-1$ binary additions. Adding the constraint

$$a_2 = r_4$$

completes the user-defined constraint.

```

rv ← row_vars;
var avar: !int ← hd rv;
rv ← tl rv;
while tl rv ≠ null do
  var av: !int ← var 0;
  require av ≥ 0 and av < pow;
  var aa: !int ← hd rv;
  require avar + aa = av;
  avar ← av;
  rv ← tl rv;
end;
require avar = hd rv;
end;

```

The main program is straightforward. It reads in a number of lines from the keyboard. Then it prints out a pretty-printed version of the puzzle and generates a sorted list of all letters of the puzzles. It creates a list of variable objects and requires the user-defined constraint *constrain* described above. When a solution has been found, it is printed out.

```

fun main(args: list of string): int
  var strs: list of string ← io.get ();
  var l: list of char ← listsort.sort (letters (strs), compare.cmp);
  lists.foreach (io.put, l); io.nl ();
  var letter_vars: list of !int ← listmap.map (fun (c: char): !int
    return var 0;

```

```

    end, l);
  require constrain (strs, l, letter_vars);
  while l ≠ null do
    io.put (hd l); io.put (" = "); io.put (!(hd letter_vars)); io.nl ();
    l ← tl l; letter_vars ← tl letter_vars;
  end;
  return 0;
end;

```

3.3 Discussion

The availability of higher-order functions simplifies a lot the task of generating and manipulation lists of constrainable variables. Functions are very useful for modularization, and higher-order functions enable the programming of generic and reusable utility routines. Even more of the imperative loops and explicit manipulation of variables could have been removed by using more higher-order functions, but since the program was intended to present imperative programming as well, they were left in.

Imperative features such as side-effecting input and output is an advantage of the constraint imperative language combination. In declarative languages, it is very tedious to insert input/output to an existing program, because this addition may require that the program is converted, e.g. to monadic style. Another advantage of the imperative paradigm is that they can be implemented very efficiently.

In comparison to a truly imperative solution, the constraint imperative version is shorter since the whole solving process has been delegated to the constraint solver. Much of the code in the example is for user interaction and could not be removed by using another programming paradigm.

Unfortunately, the constraint solvers currently available in the TURTLE implementation are very weak. It is possible to solve very small puzzles, but because of the exponential complexity of the problem and the solvers' weakness, we could not test larger examples.

A limitation of the language design is that only the first solution found is assigned to the variables in a constraint statement. The addition of nondeterminism could lift this restriction. This is a topic of future work.

The main advantage of constraint imperative programming with higher-order functions is the flexibility it provides to the programmer. Imperative, constraint-based and functional programming are seamlessly integrated into a single programming language, and whatever paradigm is most suited for solving a given problem can be used.

Additionally, the programming style eases the transition from mainly imperative solutions to solutions which use the declarative paradigm wherever it is better suited, thus leading to shorter and clearer solutions. This transition will be illustrated by the following code fragment (taken from the user-defined constraint *constrain* in Sect. 3.2), which performs two tasks: First, it generates a

list of constrainable variables, second, it calculates the maximum length of a list of strings.

```

var row_vars: list of !int ←
  listmap.map (fun (s: string): !int
    if sizeof s > max_len then
      max_len ← sizeof s;
    end;
    return var 0;
  end, strs);

```

This solution is not as clear as it could be, due to the interweaving of the functional *map* and the guarded assignment. When a program was written in this style, a programmer might clean it up by converting it step-by-step into a more declarative solution:

```

var row_vars: list of !int ←
  listmap.map (fun (s: string): !int
    return var 0;
  end, strs);
max_len ← listreduce.reduce (ints.max, 0,
  listmap.map (fun (s: string): !int
    return sizeof s;
  end, strs);

```

The usage of higher-order functions like *reduce* in this example reduces the notational overhead of writing imperative loops and avoids the effect that imperative programmers often mangle different tasks into the same loop, simply to reduce the amount of code written.

4 Related Work and Conclusion

In this paper, a new programming style called *constraint imperative programming with higher-order functions* has been described. We have introduced the constraint imperative programming language TURTLE. The new programming style has been demonstrated by first describing a program solving a particular problem and then extending the program to a generalized version. The extension makes use of all of the three programming paradigms supported by the language TURTLE, namely constraints, higher-order functions and imperative variables and control structures.

The combination of non-deterministic and imperative programming has been demonstrated by Radensky [4], who extended Pascal with nondeterminism and by Apt and Schaerf [5], who extended Modula-2 with failing/successful statements and backtracking. These languages do not support higher-order functions, nor constraint programming (for Alma-0, an extension with constraint programming is planned [6]). Constraint facilities have also been integrated into existing imperative languages by using constraint libraries, such as ILOG Solver [7] for

C++ or JACK [8] for Java. The disadvantage of constraint libraries is that they cannot add fundamental language features, like higher-order functions with a convenient syntax. Siskind and McAllester have merged non-determinism and constraint programming [9] with Common Lisp, which also has imperative assignments and higher-order functions. This approach has possibly most similarities to ours.

Constraint imperative programming with higher-order functions is an extension of constraint imperative programming, but it can also be regarded as an instance of multiparadigm programming, integrating three major programming paradigms. Further research in the area of multiparadigm programming and languages is a topic of future research.

References

1. Freeman-Benson, B.N.: Constraint Imperative Programming. PhD thesis, University of Washington, Dept. of Computer Science and Engineering (1991)
2. Grabmüller, M.: Constraint Imperative Programming. Diploma Thesis, Technische Universität Berlin (2003)
3. Borning, A., Freeman-Benson, B., Wilson, M.: Constraint hierarchies. *Lisp and Symbolic Computation* **5** (1992) 223–270
4. Radensky, A.: Toward integration of the imperative and logic programming paradigms: Horn-clause programming in the Pascal environment. *ACM SIGPLAN Notices* **25** (1990) 25–34
5. Apt, K.R., Schaerf, A.: Search and imperative programming. In: Conference Record of POPL 97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, New York, NY (1997) 67–79
6. 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
7. Puget, J.F.: A C++ Implementation of CLP. In: *Proceedings of the Second Singapore International Conference on Intelligent Systems*, Singapore (1994)
8. Abdennadher, S., Krämer, E., Saft, M., Schmauss, M.: JACK: A Java constraint kit. In: *WFLP 2001*, University of Kiel; Technical Report No. 2017 (2001)
9. Siskind, J.M., McAllester, D.A.: Nondeterministic lisp as a substrate for constraint logic programming. In Fikes, R., Lehnert, W., eds.: *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Menlo Park, California, AAAI Press (1993)