

Implementing Constraint Imperative Languages 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. Constraint imperative programming languages combine declarative constraints and imperative language features into an integrated programming language. The language TURTLE supports these programming paradigms and additionally integrates functional programming with higher-order functions and algebraic data types. This paper describes the implementation of TURTLE, consisting of a compiler, a run-time system including constraint solvers and an extensive library of supporting modules.

1 Introduction

Declarative programming languages let the programmer concentrate on *what* the solution to a problem is, by specifying the properties the solution should have and letting the programming system find it. Imperative programming, on the other hand, emphasizes *how* to calculate the solution. An imperative program contains a step-wise description of the solution algorithm which finally leads to the desired result. Each of these programming paradigms has its respective advantages. Declarative programming builds on a strong mathematical foundation which simplifies program transformations (e.g. optimization) and verification. Imperative programs can often model real-world activities more naturally, because of their state-changing semantics. The available implementations of imperative languages also yield more efficient (in time and space) programs, despite the developments in compiler technology over the last decades.

Several research activities have tried to integrate declarative and imperative programming languages in order to combine the advantages of both. Constraint-imperative programming [1] is one instance of this combination. It integrates declarative constraints and imperative language constructs such as mutable data structures and assignment. Besides novel language design issues, constraint imperative languages require new techniques for an efficient implementation.

This paper reports on the implementation of the constraint imperative programming language TURTLE [2], which integrates constraints, imperative constructs and features mainly known from functional programming languages, such as higher-order functions and algebraic data types.

Martin Grabmüller: Implementing Constraint Imperative Languages with Higher-order Functions, *Second Workshop on Multiparadigm Constraint Programming Languages (Multi-CPL'03)*, pages 43–54, Kinsale, Ireland, September 2003.

This paper is organized as follows. Section 2 briefly describes the language TURTLE. The implementation of TURTLE is presented in Sect. 3. It consists of a description of the compiler, the run-time system and the TURTLE library. Section 4 relates this paper to other work and finally concludes.

2 The Programming Language Turtle

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

Imperative programming. TURTLE provides all control structures known from traditional imperative languages: conditionals, loops, functions (and procedures) and assignment statements. Variables and data structures can be modified by assignment, and input/output is performed using side-effecting functions. TURTLE supports a rich set of data types, including integers, reals, booleans, strings, characters, arrays, lists and tuples. 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 parametrized by data types and functions can be defined in terms of these parameters, resulting in 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.

Functional programming. Higher-order functions, which can receive functions as parameters and can return functions as their value, are provided for functional programming. The supported data types also include algebraic data types as known from functional programming languages. In addition to imperative loops, iteration can also be expressed by recursion, as is normally done in functional languages. TURTLE uses eager evaluation semantics to avoid conflicts with side-effects introduced by imperative programming.

Constraint programming. Four extensions were added to the imperative and functional base language: constrainable variables, constraint statements, user-defined constraints and constraint solvers.

Constrainable variables are special variables, introduced by data type annotations, e.g. a constrainable integer variable is declared with type `! int`. The values of normal variables are given by assignments, whereas the values of constrainable variables are determined by placing constraints on them. Since constrainable variables not only hold values 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 variables' values.

Constraint statements are block-structured statements which consist of (1) a constraint conjunction and (2) of a sequence of statements, called the body. The following example shows a constraint statement constraining a variable x to a value greater than zero as long as the body (which prints the value of x) is running.

```
require  $x > 0$  in io.put (! $x$ ); end;
```

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. Since a constrainable variable can only hold a single value, an arbitrary value which satisfies the constraints is chosen. When the solving process is successful, the statements in the body are executed. The variables remain bound to their 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. A second variant of the constraint statement without a body is also provided. Constraints specified with such a statement remain valid as long as the variables in the constraint do exist.

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, its body is executed.

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. When the constraints in the stores are not satisfied, the solvers are responsible for raising an exception. Constraint statements in TURTLE allow to specify the importance of individual constraints in a conjunction 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].

Currently, only linear constraints are supported by TURTLE. Lifting this restriction requires modifications both to the solvers and the compiler, because constraints are analyzed at compile time.

The code fragment in Fig. 1 illustrates the constraint extensions of TURTLE. The user-defined constraint *all_different* receives a list of constrainable variables and places inequality constraints on each pair of list elements. Line 11 declares three constrainable variables a , b and c and initializes the variables with variable objects holding different values. Line 12 invokes the user-defined constraint in

```

1  constraint all_different (l: list of lint)
2    while (tl l <> null) do
3      var ll: list of lint := tl l;
4      while (ll <> null) do
5        require hd l <> hd ll;
6        ll := tl ll;
7      end;
8      l := tl l;
9    end;
10 end;
...
11 var a: lint := var 0, b: lint := var 1, c: lint := var 2;
12 require all_different ([a, b, c]) in ... end;

```

Fig. 1. Constraint imperative example

a constraint statement and thereby ensures that the variables' values remain pairwise different while the body executes.

3 Implementation

The implementation of the TURTLE system consists of a compiler, a run-time system and of a collection of library modules. The run-time system contains two experimental constraint solvers and a garbage collector and the library provides useful utility functions and abstract data structures. This section describes each of the parts of the TURTLE programming system.

3.1 Compiler

The compiler translates TURTLE source code into object code. We will briefly describe the general structure of the compiler and then present the handling of constraints and functional programming features in more detail. The compilation of the imperative base language into machine code will not be shown, because it is rather standard.

Compiler structure. TURTLE source programs are first parsed and converted to a syntax tree, annotated with type information and fully resolved identifiers. This intermediate program representation is called high-level intermediate language (HIL). The HIL representation is converted to a low-level intermediate language (LIL) suitable for machine code generation. LIL is the language of a stack machine, designed for easy code generation and target language independence. The code emitter finally converts LIL to the target language, which in the current implementation is ANSI C. The compilation into machine code and the linking of the program modules with the run-time library is handled by a standard C compiler and object-code linker.

Compiling constraints. The handling of constraints requires the compilation of user-defined constraints and constraint statements. Constrainable variables only affect the type-checking and the compilation of the creation of variable objects and accesses to their contents. The creation of variable objects is implemented like the creation of user-defined data types and accesses are translated to simple fetch instructions.

User-defined constraints are compiled in the same way as normal functions, except that constraint statements contained in their bodies do not need to invoke the constraint solvers to check their stores for satisfiability. This is because user-defined constraints can only be invoked from constraint statements, which will do this as soon as their statement bodies are entered.

Constraints are specified in constraint statements in TURTLE. The architecture of the system is flexible enough to integrate new constraint solvers, so it is not in advance known how constraint solvers have to handle constraints in order to efficiently solve them. Therefore, a solver-independent constraint representation is built at run-time, whenever a constraint statement requires that a constraint is added to the constraint store.

The compiler distinguishes two kinds of constraints in constraint statements: first, we have trivial constraints which do not contain any references to constrainable variables. Second, there are non-trivial constraints. Trivial constraints are translated like normal boolean expressions, followed by a test whether the result was true or false. If the result was false and the constraint was required, an exception is raised. Such constraints may appear when constraint statements are used for stating program invariants instead of using them for calculating assignments for constrainable variables. For non-trivial constraints, compilation is more complicated. Since constraints are first-class objects (they have to remain accessible to the constraint solver until the scope of the constraints is left), a representation of the constraints is built. This representation must contain references to the constrainable variables so that the constraint solver can fetch the values of these variables and can store new values into them. References to normal variables, function calls and constant subexpressions are treated as constants, so that only the results of evaluating them have to be stored in the constraint representation. This is done by evaluating these expressions as soon as the containing constraint statement is entered, and by remembering the result for inclusion in the symbolic representation. After building the representation, the constraint is tagged with its strength (0 for required constraints, and values greater than 0 for preferential (non-required) constraints) and added to the constraint store. Adding the constraint will cause the constraint solver to re-solve the store. If any required constraints in the store cannot be satisfied after adding the new constraint, the new constraint will be removed and an exception will be raised. If any preferential constraints are not satisfied, the solver tries to satisfy as many preferential constraints as possible, but without raising an exception.

The actual compilation of constraint statements consists of two phases. First, the expressions of all constraints in the constraint conjunction are partitioned into constant terms on the one hand and the constrainable variables and their

```

1  var y: int ← 4;
2  var x: !int ← var 0;
3  require 10 * x + 10 > 3 * y - 1;

```

Fig. 2. Constraint compilation example

coefficients on the other hand. In the second phase, the code for evaluating the constant terms (including function calls) is emitted as well as the code for creating the symbolic representation.

For illustration of the translation of constraint statements, we will translate the constraint statement in Fig. 2 step by step.

Line 1 declares the integer variable y . The variable x is declared in line 2 as a constrainable integer variable, so the compiler will need to translate the **require** statement as a non-trivial constraint statement. The translation of line 3 proceeds by first partitioning the terms of the constraint into constants (that includes non-constrainable variables and function calls, which are evaluated before the constraint is created) and constrainable variables, together with their coefficients. For our example, we have the constant expression

$$-10 + 3 * y - 1$$

and the constrainable variable with coefficient:

$$10 * x$$

Figure 3 shows the generated code for the example. The translation of the **require** statement consists of first pushing the constraint’s strength onto the evaluation stack. For our example, since no strength was specified, 0 (the strongest strength) is assumed (line 1). After that, an indicator for the kind of constraint (the inequality ‘>’) is pushed, followed by the number of constrainable variables, 1 in the example (lines 2–3). Then the value of the constant term (which must be evaluated before the constraint is created) is pushed onto the stack (lines 4–10), followed by all constrainable variables with their corresponding coefficients (lines 11–12). Finally, the constraint is added to the constraint store (line 13). The *add-constraint* instruction is the only instruction which interfaces to the constraint solvers (except for a *remove-constraint* instruction used at the end of constraint statement bodies) and causes the solver responsible for the constraint to build a representation from (1) the constraint strength, (2) the constraint kind, (3) the constant and (4) from the constrainable variables and their coefficients found on the stack. Whenever the solver needs to check the satisfiability of its store (because new constraints are added), it uses the internal constraint representation it has built.

Note the difference between the variables x and y . The value of y (an integer) is used for calculating the constant term of the constraint, whereas the variable object stored in x is pushed onto the stack so that the constraint solver responsible for the constraint can access the variable.

```

1  push-constant 0      // constraint strength
2  push-constant 3     // constraint kind '>'
3  push-constant 1     // number of constrainable variables
4  push-variable y     // calculate the constant term...
5  load-constant 3
6  mul
7  push
8  load-constant -11
9  add
10 push
11 push-variable x     // load the constrainable variable object
12 push-constant 10   // load the coefficient
13 add-constraint     // add the constraint to the store

```

Fig. 3. Generated code for the constraint example

User-defined constraints, which may appear as elements of the constraint conjunction, are compiled into calls to the subroutines which are created when each user-defined constraint is compiled. Since the code for adding constraints to the store is contained in the user-defined-constraint, the call simply replaces the code emitted for primitive constraints.

Before translating the constraints into code, the compiler checks whether the constraint is representable in the symbolic representation. This check is purely syntactic. Non-linear constraints or constraints with relations not supported by the constraint solvers are rejected by the compiler.

In the current implementation the constraint solvers which are responsible for individual constraints are determined at compile time, using type information: constraints on integers are handled by the finite-domain solver and constraints on reals are passed to the Indigo solver (see Sect. 3.2). A modification to choose the solver at run-time, based on the constraint kind passed with the *add-constraint* instruction, is planned as future work.

The separation of normal and constrainable variables in TURTLE has two advantages for the implementation. First, it is very easy for the compiler to analyze constraints and to generate code for creating constraint representations. Second, because constrainable variables can only be determined by constraints, and constraints are monotonically added to the constraint store in nested constraint statements, semantic problems of assignments invalidating the constraint store are avoided. This makes reasoning about constraint imperative programs much easier.

Algebraic data types. TURTLE supports the definition of user-defined algebraic data types. These data types are declared in **datatype** declarations, like the declaration of the type *tree* in the following example:

```

datatype tree = leaf(value: int) or
                node(left: tree, right: tree, key: int);

```

```

// Constructors
fun leaf (value: int): tree
fun node (left: tree, right: tree, key: int): tree
// Discriminators
fun leaf? (t: tree): bool
fun node? (t: tree): bool
// Selectors
fun value (t: tree): int
fun left (t: tree): tree
fun right (t: tree): tree
fun key (t: tree): int
// Mutators
fun value! (t: tree, value: int): ()
fun left! (t: tree, left: tree): ()
fun right! (t: tree, right: tree): ()
fun key! (t: tree, key: int): ()

```

Fig. 4. Induced signature for the *tree* data type

Using this data type declaration, the TURTLE compiler automatically generates a set of functions for creating instances of the type, for accessing the fields and for examining the variant of a given value of the type. Figure 4 shows the names and types of these generated functions. The constructor functions receive the values which will be stored into the fields of the value as parameters and create either a leaf or a node value. The discriminator functions are used to determine the variant of a given tree value, and the selectors return the values stored in the corresponding fields. The mutators modify the fields of a structured value by storing new values into the appropriate storage locations and return the unit type (). Mutator functions have been added to allow imperative programming with data structures constructed from user-defined data types.

Tail-recursion elimination. Supporting functional programming requires a proper implementation of functional programming concepts. One of these concepts is to use recursive function calls instead of loops for expressing iteration. TURTLE supports proper tail-recursion, that means that an iterative algorithm expressed as a tail-recursive sequence of function calls uses constant space, even when more than one function is involved in the recursive call chain. This is implemented by compiling each TURTLE module into one (possibly large) C function. Calls between functions in one module can then be implemented by simple C **goto** instructions instead of C function calls. This compilation scheme solves the problem of mutually recursive function calls in one module (so-called *intra-module calls*), but not between different modules (*inter-module calls*), because TURTLE also supports separate compilation.

In order to achieve proper tail-recursive function calls, even across module boundaries, the TURTLE compiler uses a compilation technique similar to the

one used in the Gambit Scheme compiler for its C back-end. Feeley et al. [4] describe how to compile languages with higher-order functions to portable C. The solution is to add a wrapper function to the run-time system whose purpose is simply to repeatedly call the C functions into which the TURTLE modules have been compiled. The C functions take a function descriptor as an argument which tells which of the TURTLE functions represented by the C function is to be called. Whenever an inter-module call is made, the C function passes a descriptor of the function to be called back to the wrapper function.

Higher-order functions. Functions are represented as *closure objects* at runtime. A closure object contains a pointer to the code of the function and the environment in effect when the closure was created. An environment holds the values of all free variables of a function. Since the free variables of top-level functions are the global variables whose addresses are fixed, there is no need to create closures for these functions. The TURTLE implementation uses a technique also from [4] for reducing the overhead of calling functions: top-level functions are not represented directly by their machine code addresses, but by statically allocated descriptors which have the same memory layout as closure objects. This makes it possible to call all functions in the same way, while avoiding to create closure objects (which have to be copied on garbage collection, see Sect. 3.2) for top-level functions.

3.2 Run-time System

TURTLE is a high-level language, supporting constraints as well as functional programming with higher-order functions. A language implementation for such a language requires a large run-time system to handle all the low-level functionality, for example constraint solving and memory management. The TURTLE run-time system is implemented as a shared library, written in C.

A TURTLE program at runtime consists of a *code section*, a *data section*, a *stack section*, a *run-time library*, *constraint solvers*, *constraint stores* and a *heap*. The code section contains the machine code of the program and the data section holds both the global variables of the TURTLE program and of the run-time system. The stack is provided by the operating system and is used by the run-time system, by the constraint solvers and for interfacing with the operating system. The TURTLE run-time contains the code of the run-time system. The constraint solvers and the constraint stores manage the active constraints and determine the values of constrainable variables. The heap stores all dynamically allocated memory and is organized in two semi-spaces for garbage collection.

Constraint solvers. Two constraint solvers are currently implemented for TURTLE. The first is a finite-domain solver over the integers. This solver is a simple backtracking implementation without any consistency checks or other optimizations and mainly serves as a proof-of-concept for the easy integration of constraint solvers into the system. The second is a solver over cycle-free linear

equalities and inequalities over the reals. It is based on the Indigo algorithm, an interval based local propagation solver [5]. Both solvers implement the run-time/constraint solver interface described in Sect. 3.1. They receive the symbolic representations of constraints on the run-time stack and create data structures for handling them on the heap. For each constrainable variable and each constraint added through this interface, the solvers create data structures for maintaining the lower/upper bounds of the variables. The finite-domain solver uses this information for simply reducing the number of instantiations it must perform, whereas the Indigo solver uses propagation for reducing the domains of the variables.

Garbage collection. The dynamic memory of the TURTLE system is maintained automatically by the garbage collector included in the run-time system. The collector employs a simple *stop©* algorithm, as described by Cheney [6]. When the garbage collector is invoked, all memory cells reachable from the machine registers and the run-time stack are copied into the second half of the heap and then all cells which were not copied are reclaimed. Since these cells are not reachable anymore, they cannot affect any future computation and may therefore be recycled.

The garbage collector is also responsible for determining whether any constrainable variables have left their scopes, and to notify the constraint solvers of that fact. This is necessary, because all constraints placed on such variables need to be removed from the stores, so that they cannot influence the program execution any more.

The Trampoline. The compilation scheme described in Sect. 3.1 for functions and user-defined constraints, where each module is compiled into a single C function requires some support in the run-time system. The run-time system contains a short function (called *trampoline*) which calls the TURTLE functions requiring inter-module calls. The function is simply a loop which repeatedly calls the function contained in the global program counter. This function is also responsible for repeatedly checking whether an interrupt (user interrupt or operating system signal) has occurred, and for calling the TURTLE interrupt handler.

3.3 Library Modules

A library containing often-used data structures and functions is very important if a language is intended to be used in practice. Therefore, a standard library for TURTLE has been designed and implemented in the reference implementation.

The library provides a range of useful data types, such as trees and hash tables, support modules for the built-in data types for list, array, string and number manipulation and of course functions for imperative in- and output. Additionally, some low-level library modules have been included for interfacing with the operating system, such as for process management and network programming. The library makes intensive use of TURTLE features such as the module system,

parametrized modules and higher-order functions. Thus the library implementation was very useful in debugging the compiler and testing the language design. The structuring of the library was inspired by the design of the “Bibliotheca Opalica”, the standard library of the functional language Opal [7].

4 Related Work and Conclusion

This paper describes the implementation of the constraint imperative programming language TURTLE. It combines well-known techniques for implementing imperative and functional languages with new compilation schemes for the constraint extensions supported by TURTLE: constrainable variables, constraint statements, user-defined constraints and the interface between the user program and the constraint solvers. We will now relate our implementation to other work and draw a conclusion.

The constraint imperative programming language Kaleidoscope [8] combines object-oriented and constraint programming. The implementation of its compiler and run-time system [9, 10] is based on a translation of all imperative source language constructs (assignments etc.) into primitive constraints, which are in turn handled by constraint solvers. Kaleidoscope is thus based on a constraint solving virtual machine on which imperative programming is modelled. The approach we have taken in designing and implementing TURTLE is the opposite: we have started with an imperative language and added constraints on top of it. Apt et al. [11] have designed an extension of Modula-2 with non-determinism, based on backtracking. This language, called Alma-0, has been implemented, but the proposed extensions to a constraint imperative language have not [12], making it difficult to compare it to our approach. Other work comparable to ours are constraint libraries for imperative languages, such as ILOG [13] for C++ or JACK [14] for Java. Their advantage is the easy integration into existing programs written in imperative languages, but their disadvantage is the semantic gap between their constraint solving capabilities and the imperative execution model of their underlying languages.

The implementation presented in this paper has been used to implement various example programs, ranging from simple constraint imperative programs solving crypto-arithmetic puzzles to a working web server and a front-end (scanner and parser) for the TURTLE language. The performance of the functional and imperative part of the language is quite satisfactory, for some simple test programs the TURTLE implementation yields programs which are by a factor of 10 slower than comparable programs written in C. This overhead is partly due to the automatic memory management and to the trampoline technique necessary for tail-recursive inter-module calls, which are not available in C. The constraint part of the language was not measured against other systems, because the solvers are very weak and cannot compete with any reasonable constraint programming system.

As we have shown, many techniques for implementing imperative and functional languages can be transferred to the implementation of integrated pro-

gramming languages and seamlessly combined. The combination of imperative and functional language constructs in TURTLE is already efficiently usable, and with the integration of more powerful constraint solvers, constraint imperative programming with higher-order functions will also be usable in practice.

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. Feeley, M., Miller, J.S., Rozas, G.J., Wilson, J.A.: Compiling higher-order languages into fully tail-recursive portable C. Technical Report 1078, Département d’informatique et de recherche opérationnelle, Université de Montréal (1997)
5. Borning, A., Anderson, R., Freeman-Benson, B.: The Indigo algorithm. Technical Report 96-05-01, Dept. of Computer Science and Engineering, University of Washington (1996)
6. Cheney, C.J.: A non-recursive list compaction algorithm. *Communications of the ACM* **13** (1970) 677–678
7. Pepper, P.: Funktionale Programmierung in OPAL, ML, HASKELL und GOFER. 2nd edn. Springer (2003)
8. 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
9. Lopez, G., Freeman-Benson, B.N., Borning, A.: Implementing constraint imperative programming languages: the Kaleidoscope’93 virtual machine. In: *Proceedings of the 1994 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. (1994) 259–271
10. Lopez, G.: The Design and Implementation of Kaleidoscope, A Constraint Imperative Programming Language. PhD thesis, University of Washington, Department of Computer Science and Engineering (1997)
11. Apt, K.R., Brunekreef, J., Partington, V., Schaerf, A.: Alma-0: An imperative language that supports declarative programming. *ACM Toplas* **20** (1998) 1014–1066
12. 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
13. Puget, J.F.: A C++ Implementation of CLP. In: *Proceedings of the Second Singapore International Conference on Intelligent Systems*, Singapore (1994)
14. Abdennadher, S., Krämer, E., Saft, M., Schmauss, M.: JACK: A Java constraint kit. In: *WFLP 2001*, University of Kiel; Technical Report No. 2017 (2001)