Turtle: A Constraint Imperative Programming Language

Martin Grabmüller and Petra Hofstedt

Technische Universität Berlin
Fakultät IV Elekrotechnik und Informatik
{magr,ph}@cs.tu-berlin.de

Abstract. Ideally, in constraint programs, the solutions of problems are obtained by specifying their desired properties, whereas in imperative programs, the steps which lead to a solution must be defined explicitly, rather than being derived automatically. This paper deals with the design and implementation of the programming language Turtle, which integrates declarative constraints and imperative language elements in order to form a more powerful programming paradigm suitable for solving a wide range of problems.

1 Introduction

Programming languages can be divided into two main categories: imperative and declarative languages. Imperative programming languages are statement-oriented and it is the task of the programmer to specify explicitly which steps a computer program has to perform in order to find the solution to a given problem. In declarative languages, the properties of the desired solutions are specified and it is mainly up to the programming system (compiler and run-time system) to find a way for obtaining the solutions.

Even though the advantages of declarative programming—e.g., more effective development and increased program correctness—are widely recognized, imperative programming languages are still used in the industry as well as in academia. This is partly due to the fact that imperative programming languages are still more efficient than declarative ones for many applications, and partly because traditionally, software projects were started with imperative languages and there is a need to still maintain them. One way to make use of the advantages of both programming paradigms is to integrate them into one language, so that whatever paradigm is better suited can be used to perform a given task.

The combination of constraint-based and imperative programming has been called *constraint imperative programming* (CIP) in literature [1]. The object-oriented programming language Kaleidoscope [2] is one of the first examples of a language in that paradigm, but other languages or systems also share properties of CIP [3, 4].

This paper describes the design of the higher-order constraint imperative programming language Turtle[1]. Even though constraint imperative programming

[1] Note that turtles normally live longer than most other animals.

as originally introduced by [1] refers to object-oriented programming languages, we will use a broader definition of CIP. In the following text, all languages which have both constraint and imperative features (including object-oriented ones) will be called constraint imperative programming languages.

The rest of this paper is organized as follows: We present the higher-order constraint imperative programming language Turtle in Sect. 2. This is done by first defining the imperative part of the language which is then enhanced by introducing four essential language extensions. Section 3 touches the semantics of Turtle by a brief introduction of the Turtle Abstract Machine (TAM) and sketches the implementation of our CIP language carried out based on this machine description. Finally, in Sect. 4, we discuss Turtle with respect to related work and draw a conclusion.

2 Turtle – A Constraint Imperative Language

Turtle was developed by first designing a base language which includes the essential features of typical traditional imperative languages and language constructs known from functional programming languages. This base language was then enriched with four concepts necessary for extending it to a constraint imperative language. This section describes both the base language and the constraint extensions and presents an example for constraint imperative programming with Turtle.

2.1 The Base Language

The imperative subset of the Turtle language consists of all programming constructs necessary for imperative, structured programming. These constructs include imperative control structures, variables, functions, a rich type system allowing the definition of new data types, a module system for structuring programs with polymorphic modules and higher-order functions.

All necessary control structures of imperative languages, such as conditionals, loops, function calls, expressions and assignments are available in Turtle. Fig. 1 summarizes the syntax of the imperative part of the language. t stands for type expressions, which are used to denote data types. Statements s do not produce values, but are instead executed for their side effects. Statements are declarations, assignments, function calls and control structures. Expressions, denoted by e, are evaluated in order to produce values, and always appear as parts of statements. Note that functions can be both declared by statements as well as created by expressions which result in function values.

Constants are named c in Fig. 1 and can be integer, boolean, real, character or string constants. Variables v represent storage locations into which values can be stored by assignment statements, and from which values can be fetched. The notation $e_1[e_2]$ stands for array accesses.

Functions f are like functions or procedures in other imperative languages: they receive input values as parameters and return values as their results. The calling convention in Turtle is call-by-value.

$$
\begin{array}{lll}
t & ::= & t_{simple} \mid t_{complex} \\
t_{simple} & ::= & int \mid bool \mid real \mid char \\
t_{complex} & ::= & n < t_1, \ldots, t_n > \mid \textbf{string} \mid \textbf{array of } t \mid \textbf{list of } t \mid (t_1, \ldots, t_n) \\
s & ::= & \textbf{var } v : t; \mid f(e_1, \ldots, e_n); \mid e_1 := e_2; \\
 & \mid & \textbf{datatype } n < n_1, \ldots, n_l >= v_1(l_{11} : t_{11}, \ldots, l_{1n} : t_{1n}) \textbf{ or } \ldots \\
 & & \quad \textbf{or } v_m(l_{m1} : t_{m1}, \ldots, l_{mn} : t_{mn}); \\
 & \mid & \textbf{fun } f(v_1 : t_1, \ldots, v_n : t_n) : t \quad s; \ldots \textbf{ end}; \\
 & \mid & \textbf{while } c \textbf{ do } s; \ldots \textbf{ end}; \mid \textbf{return } e; \\
 & \mid & \textbf{if } c \textbf{ then } s_1; \ldots \textbf{ else } s_2; \ldots \textbf{ end}; \\
e & ::= & c \mid v \mid f(e_1, \ldots, e_n) \mid e_1[e_2] \mid \textbf{fun } (v_1 : T_1, \ldots, v_n : T_n) : t \quad s; \ldots \textbf{ end}
\end{array}
$$

Fig. 1. Base Language Syntax

The type system has primitive data types like integer or real numbers, boolean values, characters and strings. TURTLE supports tuples, arrays and lists of arbitrary base types. User-defined algebraic, recursive data types are introduced by **datatype** declarations, which declare data types n, possibly parameterized by other types, with several variants v_i, where each variant has a number of fields l_{ij}. The compiler automatically generates functions for creating values of user-defined types (constructor functions) and for selecting and modifying the fields stored in the values (selector and mutator functions). Since a data type can have one or more variants, functions for examining the variant a given value has (discriminator functions) are also generated.

In TURTLE, programs are composed of modules which define data types, functions and variables. Some or all of the definitions in a module can be exported, so that other modules of the same program can import and use them. By these means, various principles of software engineering, such as encapsulation, information hiding and the definition of interfaces between modules can be expressed in TURTLE.[2]

In addition to the imperative language discussed above, some functional programming concepts have also been integrated: functions in TURTLE are higher-order, so that functions can be arguments to and results from function calls and can be stored into arbitrary data structures. This feature was added because it proved very useful for code reusability and abstraction in other programming languages besides purely functional languages, for example in Scheme [5]. Another important feature—not specific to functional programming, but most common in these languages—is the possibility to make modules polymorphic. This means that modules can be parametrized by data types and that the types and functions defined in these modules can operate on any data type which is specified when importing them.

[2] The syntax for defining modules has been omitted from Fig. 1 for clarity.

t ::= **!** t_{simple}
s ::= **constraint** $f(v_1 : T_1, \ldots, v_n : T_n)$ $s; \ldots$ **end**;
| **require** $c_1 : p_1 \wedge \ldots \wedge c_n : p_n$ **in** $s; \ldots$ **end**;
| **require** $c_1 : p_1 \wedge \ldots \wedge c_n : p_n$;
e ::= **var** e | **!** e | **constraint** $(v_1 : T_1, \ldots, v_n : T_n)$ $e; \ldots$ **end**

Fig. 2. Constraint Extension Syntax

2.2 Constraint Programming Extensions

The extension of an imperative language to a constraint imperative one by adding declarative constraints allows for cleaner and more flexible programs, as will be shown in Sect. 2.3.

Four concepts have been found to be necessary for extending an imperative language to a constraint imperative one: constrainable variables, constraint statements, user-defined constraints and constraint solvers. These extensions are required because it must be possible to place constraints on variables, to abstract over constraints and finally, to solve the constraints. The syntactic extensions to the base language are shown in Fig. 2.

Constrainable variables are variables whose values can be determined by placing constraints on them. In TURTLE, constrainable variables are declared by giving them constraint types. This is done by annotating the type of the variable with an exclamation mark in the variable definition. Constrainable variables are distinguished from normal, imperative variables, because they behave differently. Normal variables are set to values by assignment statements, whereas constrainable variables are determined by constraints. So the former are imperative, whereas the latter are declarative. The separation into two classes of variables makes the semantics cleaner and it is easier to program with them, because the use and behaviour of each variable can be deduced from the type of the variable. Normal variables cannot be determined by constraints, and if such a variable appears in a constraint, its value is taken as a constant when the constraint is solved.

To make use of abstraction mechanisms like functions and abstract data types, it must be possible to pass constrainable variables to functions or user-defined constraints or to store them into data structures. Each constrainable variable is associated with a *variable object* which is explicitly created by a **var** expression and holds the value of the variable. By using variable objects, it is possible to share these objects between several constrainable variables and to place the variable object in different data structures, for example, for constraining several data structures at once. When the value of a constrainable variable is needed, it has to be extracted by the dereferencing operator **!**. This is shown in Fig. 2 as the expression extension **!** e. [3]

[3] The type system forbids expressions like **!!!** 7, which are meaningless.

Constraint statements are the statements which allow the programmer to place constraints on constrainable variables. In TURTLE, only one such statement exists: the **require** statement. This statement normally consists of a constraint conjunction $c_1 : p_1 \wedge \ldots \wedge c_n : p_n$ and a body $s; \ldots$ The body is a sequence of statements and the constraints in the conjunction are enforced as long as the body executes. Enforcing means that the built-in constraint solver tries to satisfy the constraint conjunction by adding it to the constraints in the constraint store. All constraints of enclosing constraint statements are contained in the store. When the solver is able to solve the conjunction of the store together with the newly added constraints, each constrainable variable gets assigned a value such that the values of all constrainable variables form a solution of the constraint store. When the body of the constraint statement is left, all constraints added for that statement are removed from the store. Because sometimes constraints must stay in effect when the scope of a constraint statement is left (e.g. when constraints are to be placed on global variables), a variant of the **require** statement without a body is provided. The constraints defined with such a statement stay in effect as long as the variables they constrain exist. Both versions of the **require** statement are shown in Fig. 2; the third new production for s refers to the user-defined constraints discussed below.

User-defined constraints. In the same way as functions abstract over statements, user-defined constraints abstract over constraints. A user-defined constraint is similar to a function, but its purpose is not to collect a sequence of statements which might be executed in more than one place, but to collect constraints which might be needed in several constraint statements. An example for this is a user-defined constraint *all_different(L)* which constrains the variables in the list L to be pairwise different. Whenever a program needs to specify that a set of variables must be pairwise different, this user-defined constraint may be used in the constraint conjunction of a constraint statement instead of spelling out all the necessary inequalities. In TURTLE, user-defined constraints are defined similar to functions, but declared with the keyword **constraint** instead of **fun**. User-defined constraints cannot be called like normal functions, they may only be invoked from constraint statements. Functions, on the other hand, may be called from constraint statements. The returned value is then used in the constraint as a constant value, just as normal variables are treated in constraints. Calling user-defined constraints results in additional (simpler) constraints being placed onto their parameters, they do not produce values.

Constraint solvers. The last component necessary for CIP is a constraint solver capable of solving the constraints appearing in the constraint imperative program. The solver is responsible for checking whether a constraint conjunction specified in a constraint statement can be satisfied together with the current constraint store, for adding the constraints to it and for determining values for the variables on success. It must also remove constraints from the store whenever the scope of a constraint statement is left.

One feature which is not strictly necessary for CIP, but very useful when modelling real-life problems is the possibility to define constraint hierarchies [6].

With a constraint solver supporting constraint hierarchies, it is possible to assign strengths to individual constraints which represent their importance. The solver respects these strengths when solving the constraint store and tries to satisfy the most important constraints, even if that means that less important ones (preferential constraints) are left unsatisfied. Especially for over-constrained problems, such as those appearing in graphical applications, this is very useful.

In TURTLE, constraint hierarchies can be specified by assigning strengths to the individual constraints appearing in constraint statements. Syntactically, this is done by annotating the constraints with symbolic strengths (preferences) (p_i in Fig. 2). When no such annotation is given, the strength defaults to the strongest strength, called *mandatory*. Constraints with this strength always get satisfied by the solvers and if that is not possible, an exception is raised.

The constraint programming model currently implemented in TURTLE only allows a single solution for the constraints, so that the solvers only determine the first (not necessarily best) solution. This restriction could be lifted by adding backtracking to the TURTLE semantics, as with Alma-0 [7]. This is a topic of future consideration.

2.3 Programming In Turtle – An Example

This section presents an example of CIP in TURTLE. It will demonstrate the use of the constraint features of the language. More detailed examples may be found in [8]. Consider the example program in Fig. 3. The program is intended to calculate the margins, column width and inter-column gap for a simple layout problem. A program like this could be part of a larger typesetting system. The variables *lm* and *rm* represent the left and right margin of the paper, *gap* stands for the gap between the two columns and *pw* is the page width. *col* represents the width of both the left and the right column.

The program starts by declaring the variables as constrainable variables of type *real* and initializes the variables. In this example, all variables are initialized to the value zero, because the actual values will later be determined by constraints. After the variable declarations, a constraint statement is used to specify the constraints on the variables. Some of the constraints are not annotated with a constraint strength, so that they will be assigned the strongest strength, *mandatory*. *lm*, *rm* and *pw* are specified to take fixed values, because these values are normally fixed for a printer or paper size. The gap between the columns is constrained to lie in the interval [0.5, 2.0], with a *medium* preference at 0.5. This preference ensures that the gap will be as narrow as possible.

Another preferential constraint, but with more important strength *strong*, is placed on the variable *col*. This constraint tries to hold the value for *col* below or equal to 7.0. Together with the preferential constraint on *gap*, this will result in a narrow gap as long as *col* is smaller than 7.0. The gap will only get wider when the column width exceeds 7.0. When that happens, the constraint with strength *strong* will override the constraint labelled with *medium*. The constraint $col <= 7.0$ will not be violated until *gap* reaches its upper bound 2.0 in order to satisfy the overall constraint conjunction.

```
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. 3. Layout Example Program

The results calculated for this example are as follows. *gap* will get its maximum value 2.0, and the *medium* constraint on *gap* will be violated as well as the *strong* constraint on *col*, so that the latter variable will get the value 7.5. *lm, rm* and *pw* will receive the values 2.0, 2.0 and 21.0, respectively, so that the main constraint $gap + lm + 2.0{*}col + rm = pw$ is satisfied by $2.0{+}2.0{+}2.0{*}7.5{+}2.0 = 21.0$.

Finally, in the body of the constraint statement, the values of all variables are printed out. The variables need to be dereferenced using the **!** operator, because the values of the variable objects stored in the constrainable variables shall be printed, not the variable objects themselves.

The advantage of using a constraint imperative language instead of a traditional imperative one is that the problem specification (the constrainable variables and the constraints) is expressed declaratively, and it can be used to solve a variety of problems: it can be used to calculate the column width for a given paper size as well as for calculating the optimal paper size when given a specific column width.

3 Semantics and Implementation

In this section we describe the TURTLE Abstract Machine (TAM), which provides a formal semantics of TURTLE. Furthermore, we briefly sketch on the implementation of our constraint imperative language which has been carried out based on this machine description.

3.1 The Turtle Abstract Machine

The TURTLE Abstract Machine provides an operational semantics for μTURTLE which is a subset of TURTLE. Every TURTLE program can be converted into an equivalent μTURTLE program by a straightforward syntactical transformation, where various syntactic entities are converted into simpler equivalent constructs, e.g. tuple assignment is converted into multiple single-assignments. For a detailed description see [8].

The TAM is basically a standard stack machine for imperative languages equipped with registers for maintaining the computation environment, and extended with instructions for managing the constraint store. Its design ideas inherit from [9] and [10].

There are four memory areas and a set of registers in the TAM. The *memory* is divided into

- the *Code* area containing the machine instructions of the executing program,
- the *Stack* for storing intermediate values produced during the evaluation of expressions and function calls,
- the *Store* which holds all dynamic data structures created during execution (environments for storing local variables, closures for handling higher-order functions, continuation records for storing the machine state at function calls, and a chain of exception handlers, i.e. code addresses for resuming execution when an exception occurs plus corresponding continuations), and
- the constraint store *CStore*. The TAM itself accesses this store only by abstract machine instructions, whereas the integrated solvers are responsible for maintaining some representation of the constraints in the CStore.

The *register set* contains an accumulator *acc* for storing intermediate values during program execution like results of function calls and primitive operations, the stack pointer *sp* which points to the top of the *Stack*, and the program counter *pc* which always points to the next TAM instruction in the *Code* area to be executed. The registers *cont*, *env*, and *ex* are used for access to the chain of continuation records, to the environment holding the local variables of the currently executing function and a reference to the enclosing environment, and to a list of exception handlers, resp.

The TAM *instruction set* can be divided into classes: Various *load and store instructions* fetch values from memory locations or constants to the accumulator or store the accumulator value to memory. There are *instructions for closure and environment creation*, *instructions for function call and return*, *branching instructions* to control the program flow, and an instruction to halt the machine. The machine has several *instructions for constraint and constraint store management*, i.e. for creation of constraint run-time representations from the constraint specifications and for the addition and removal of constraints from and to the constraint store, resp. Finally, *instructions for exception handling* allow to add and delete entries to and from the exception handling chain and to raise an exception. These instructions are, among other things, necessary to handle the violation of constraint stores (cf. Example 1).

```
1   var x: !int := var 0;
2   var y: int := 2;
3   require x < y and
4               x = 2 : strong in
6     f(!x,y);
7   end;
```

Fig. 4. Cut-out of a TURTLE program example

The translation of a μTURTLE program into TAM code is performed according to a translation scheme given in [8]. We go not into detail wrt. this scheme but discuss an example in the following. The execution of a μTURTLE program requires the preparation of the machine followed by the interpretation of the program's TAM instructions until a `halt` instruction is executed.

Example 1. Let us demonstrate the translation of a cut-out of a μTURTLE program into TAM instructions and their interpretation by the abstract machine.

In Fig. 4, a constrainable variable x and another simple (non-constrainable) variable y are declared. The variable x is initialized with 0, the value 2 is assigned to y. The constraints $x < y$ and $x = 2$ are required to hold during the function call f(!x,y).

Note, that since x is a constrainable variable, the behavior of the expression $x < y$ is different from one, where x and y are both simple variables. In the latter case at run-time it would be simply tested whether the current values of the variables fulfill the test, while in the case of a constraint the constrainable variable can be (re)set such that the corresponding constraint (and possibly further ones) is (are) satisfied. The first constraint has the strongest strength, i.e. it is *mandatory.* Its satisfaction by the solver is, thus, stronger desired than that of the second constraint with strength *strong.*[4]

Using the translation scheme the abstract machine code of Fig. 5 is produced. Lines 1-6 and 7-8 correspond to the initialization of the constrainable variable x to 0 and of the variable y to 2. For the constrainable variable x a function call is generated which creates a variable object (with initial value 0) at run-time (lines 1-6). For the variable y the constant 2 is loaded to the accumulator and afterwards stored into the *Store* at a corresponding location. At this, $\mathcal{L}$ is a function transforming variables to location descriptors at compile-time.

The remaining lines correspond to the **require** statement which involves a management of the constraint store. Lines 9, 10 and 11, 12 create symbolic representations of the two constraints (including their strengths) and convey references to them from the accumulator into corresponding *Store* locations of newly created variables t_1 and t_2. Note that the constraint strengths are translated to non-negative integer constants. The strongest strength *mandatory* is

[4] Since the constraint solver cannot satisfy both constraints at once, it will ignore the second constraint such that x will remain 0 finally.

1		load-constant 0
2		push
3		save-continuation l_0
4		load-variable rt_make_var
5		call
6	l_0:	store-variable $\mathcal{L}[\![x]\!]$
7		load-constant 2
8		store-variable $\mathcal{L}[\![y]\!]$
9		make-constraint $(x < y)$, 0
10		store-variable $\mathcal{L}[\![t_1]\!]$
11		make-constraint $(x = 2)$, 2
12		store-variable $\mathcal{L}[\![t_2]\!]$
13		add-constraints $\mathcal{L}[\![t_1]\!], \mathcal{L}[\![t_2]\!]$
14		handle l_1
15		$\mathcal{T}[\![\mathrm{f}(!x,y)]\!]$
16		remove-constraints $\mathcal{L}[\![t_1]\!], \mathcal{L}[\![t_2]\!]$
17		unhandle
18		jump l_2
19	l_1:	remove-constraints $\mathcal{L}[\![t_1]\!], \mathcal{L}[\![t_2]\!]$
20		raise
21	l_2:	...

Fig. 5. TAM Code for the TURTLE Code of Fig. 4

0 while numerically greater values represent weaker strengths, i.e. 2 represents *strong* here. Line 13 adds the constraints to the constraint store and raises an exception if the newly built store cannot be satisfied. The expression $\mathcal{T}[\![\mathrm{f}(!x,y)]\!]$ at line 15 represents the body of the **require** statement whose translation is left out in this example.[5] Line 16 removes the constraints from the store again and resolves it at this. Constraints must be removed from the store not only when the body finished normally as discussed above but also when an exception is raised during the execution of the body. Thus, in line 14 an exception handler is set up which removes the constraints before re-raising the exception (lines 19 and 20). Line 17 removes the exception handler when execution finishes without causing an exception to be raised.

Note that in case of **require** statements without body, the creation of an exception handler is not necessary as well as at the compilation of user-defined constraints (not shown in this example) because user-defined constraints can only be invoked from constraint statements which will resolve the constraint store anyway.

As shown in the example, adding the constraint to the store automatically causes the constraint solver to resolve the store, i.e. to check whether the constraint store together with the newly added constraints is satisfiable. In that case, the store is modified accordingly. If any required constraint in the store is violated, the newly added constraint is removed from the store and an exception is raised. If any preferential constraint is not satisfied, the solver tries to satisfy as many preferential constraints as possible, but no exception is raised. When constraints are removed, the solver resolves its store, too.

3.2 The Turtle Compiler and the Run-Time System

In this section, we shortly sketch on the TURTLE compiler and the run-time system of TURTLE. At this, we will consider the internal representation of constraints in more detail.

[5] $\mathcal{T}$ stands for the translation function.

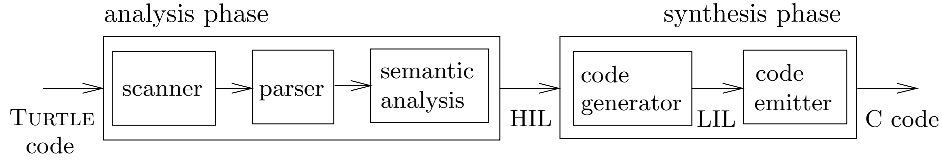

Fig. 6. Structure of the TURTLE complier

The TURTLE compiler is implemented as a library which exports a single function translating the input file to either an object file (for library modules) or to an executable (for main programs). Even if currently only a command line compiler is available, the library can be linked into any application, so it could be used for implementing an integrated development environment.

As usual the compiler is composed of an analysis and a synthesis part (cf. Fig. 6). When the analysis phase consisting of the scanner, the parser and a semantic analysis part (context checking, overload resolution etc.) is finished, the compiler has produced an internal intermediate tree-like representation of the source program which is syntactically and semantically (wrt. the static semantics) correct. Every identifier has been unambiguously related to the entity it stands for and all nodes are annotated with the types of the expressions they represent. We call this representation high-level intermediate language code (HIL code). At this stage, machine-independent optimizations on the HIL code may follow optionally, but they have not been implemented currently. Afterwards, the HIL code is passed as input to the synthesis phase of the compiler.

In this phase the HIL representation is translated to the target code, which in the case of the current TURTLE implementation is the programming language ANSI C. In a first step the HIL code is transformed by the code generator into low-level intermediate language code (LIL code) which has a form similar to assembler language, but is machine independent. The LIL representation reflects the TAM machine code generated according to Sect. 3.1 and puts thus the intended operational semantics into practice. However, for practical reasons LIL has more instructions than the TAM language so that the generation of efficient code is easier and it is possible to handle restrictions of real machines, e.g. checking for heap overflow which is not necessary in the abstract machine with conceptionally unbounded memory. After machine-independent optimizations (like peep-hole optimization), in the second step of the synthesis the LIL code is translated into C code, which is then compiled into machine code by a C compiler.

Example 2. Figure 7 shows a part of the LIL code representation of Example 1 concerning the initialization of the variables and the creation of a constraint representation. Lines 1-5 and 6, 7 realize the variable declarations and correspond to the lines 1-6 and 7, 8 in Fig. 5. The variable y is a normal variable which is initialized to the value 2 (lines 6, 7), whereas x is initialized with a constrainable

```
1   load-int #0                       % var x: !int := var 0;
2   push
3   make-int-variable
4   variable-set
5   store env(0, 1)
6   load-int #2                       % var y: int := 2;
7   store env(0, 2)
8   load-int #0                       % x < y : 0
9   push
10  load env(0, 2)
11  push
12  load env(0, 1)
13  push
14  load-int #1
15  push
16  add-int-constraint #4, #1
    ...
25  resolve-int-constraint
```

Fig. 7. LIL code representation of the TURTLE code of Fig. 4

variable object. This object contains a value slot which is initialized by the operand of the var expression, i.e. 0, and a pointer to a solver-specific data structure holding the information necessary for constraint solving.

While the creation of constraint representations can be kept very abstract in the TAM code (lines 9, 10 and 11, 12 in Fig. 5), for the actual compiler implementation this needs a more detailed treatment, e.g. lines 8-16 realize the creation of the representation of the constraint $x < y$. The add-constraints instruction in the TAM code at line 13 is realized by a corresponding instruction for every constraint (e.g. in line 16) and one final resolve-int-constraint instruction per **require** statement.

The above example already gives a first impression about how the internal representation of constraints is realized. Let us consider this in more detail. Trivial constraints which do not contain any references to constrainable variables are translated like normal boolean expressions followed by a test whether the result was true or false. In the latter case and if the constraint was required, an exception is raised.

More interesting is the compilation of non-trivial constraints. Since in TURTLE it must be possible to transfer constraints between the user program and the constraint solvers, the compiler must build a representation of the constraint. This representation must contain the strength of the constraint and references to the constrainable variables so that the solver can fetch their values and store new values into them.

The translation of non-trivial constraints partitions them into constants (including non-constrainable variables and function calls) which are evaluated be-

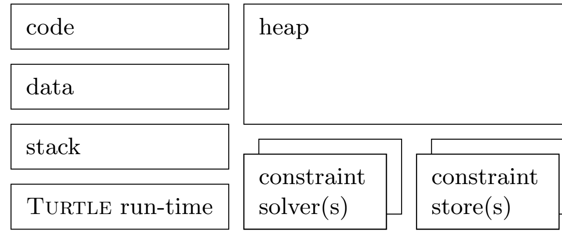

Fig. 8. TURTLE program components

fore the constraint is created and into constrainable variables together with their coefficients. When translating a constraint[6] of a **require** statement first its strength is pushed onto the evaluation stack followed by the constant term and the constrainable variables with their correspondent coefficients. Finally, an indicator for the kind of constraint is pushed onto the stack as well as the number of constrainable variables.

Example 3. Consider again the creation of the constraint representation for $x < y$ in Fig.7, lines 8-16.

The instructions 'load-int #0' and 'push' load strength '0' of this *mandatory* constraint onto the stack. This is followed by pushing the constant term 'env(0,2)' (representing y), the constrainable variable stored in 'env(0,1)' (representing x) and its coefficient '#1' (lines 10-15). Note the difference between y and x. The value of y is used for calculating the constant term of the constraint (i.e. y itself), whereas the value of x (which is a variable object) is pushed onto the stack so that the constraint solver responsible for the constraint can access the variable. Finally, a '>'-constraint (coded by '#4') with one ('#1') constrainable variable is added to the constraint store by 'add-int-constraint #4, #1'.

Figure 8 shows the parts of a TURTLE program when it is executed. It consists of the compiled code, the data section holding global variables of the TURTLE program, the stack necessary for the run-time and the operating system, the run-time system which provides basic functionality like in-/output, memory management and operating system interfaces, the heap and possibly several constraint solvers. The heap stores dynamically created data structures, the environments and continuations of active TURTLE functions. The constraints are held in their according constraint stores which are manipulated by the associated constraint solvers only.

Currently two experimental solvers for treating constraints for which the compiler cannot generate efficient straight-line code are integrated: a solver for cycle-free linear real arithmetics based on the Indigo algorithm [11] which is

[6] The described translation based on a partitioning into constants and constrainable variables used for the constraints of the currently integrated solvers will differ in general if other solvers are integrated.

able to handle constraint hierarchies and a finite-domain solver over integers. Note, that the incorporated solvers are quite weak and the integration of more powerful and efficient solvers is a task of future work. The architecture of our system however is open for the inclusion of new solvers and the interface between the (imperative) run-time system and the constraint solvers is cleanly designed and flexible.

The constraint hierarchy support in TURTLE is restricted to the syntax necessary for specifying them and the interface between the run-time system and the constraint solvers which attaches strengths to the symbolic constraint representation. It is the task of the constraint solvers to make use of the strength information (as the Indigo solver does) or to ignore it (as in the currently implemented finite domain solver).

Even if the current TURTLE compiler implementation is an experimental one, it is quite usable as demonstrated by many TURTLE example programs including a web server and a TURTLE compiler front end (scanner, parser and abstract syntax tree implementation). The programs run with reasonable memory consumption and speed, i.e. purely imperative benchmark examples run slower for a factor of about 10 wrt. equivalent C programs, and provide enough debugging information to program comfortably.

4 Related Work and Conclusion

In this paper, we presented the design and implementation of the constraint imperative programming language TURTLE.

The name *constraint imperative programming* was introduced by Borning and Freeman-Benson [1, 12] and originally only referred to object-oriented languages with constraint programming extensions. We have extended this notion to include all languages with both imperative and constraint concepts. A further example of this paradigm is the language Alma-0 [7], which currently amalgamates concepts of pure logic and imperative languages but which is intended to be extended to a CIP language [3].

TURTLE's concept of constrainable variables and the idea to separate them from normal variables is similar to the "variables" and "unknowns" proposed for extensions of Alma-0 [3]. Our user-defined constraints perform a similar task as *constraint constructors* in the language Kaleidoscope [2]. Constraint statements similar to ours can also be found in that language. However, in contrast to Kaleidoscope, the separation of constrainable variables gives more control over the handling of constraints. Additionally, TURTLE extends the CIP paradigm by integrating higher-order functions into the constraint imperative framework, thus making the language much more flexible and powerful. The algebraic data types of TURTLE come from the purely functional language Opal [13].

Besides the approach to create new programming languages which integrate both imperative and constraint programming, constraint libraries, like the constraint library ILOG [14] for C++ or JACK [4], a library for programming in Java, have been developed. Constraint libraries have the advantage of easier

integration into existing programs, whereas in CIP languages, constraints and imperative constructs are more tightly combined, which makes the semantics cleaner and helps with optimization.

The language TURTLE and its implementation are practical and the language has been used for implementing both imperative and constraint programs of different sizes. The major drawback of the current implementation is the lack of powerful and efficient constraint solvers, which is a topic of future work. The language design itself is flexible and expressive enough, such that TURTLE can be used for solving problems for which good imperative solution techniques are known as well as those for which declarative solutions are preferable.

References

1. Freeman-Benson, B.N.: Constraint Imperative Programming. PhD thesis, University of Washington, Dept. of Computer Science and Engineering (1991)
2. 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
3. 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
4. Abdennadher, S., Krämer, E., Saft, M., Schmauss, M.: Jack: A Java constraint kit. In: WFLP 2001, University of Kiel; Technical Report No. 2017 (2001)
5. Kelsey, R., Clinger, W., Rees, J., et al.: Revised^5 report on the algorithmic language Scheme. ACM SIGPLAN Notices **33** (1998) 26–76
6. Borning, A., Freeman-Benson, B., Wilson, M.: Constraint hierarchies. Lisp and Symbolic Computation **5** (1992) 223–270
7. Apt, K.R., Brunekreef, J., Partington, V., Schaerf, A.: Alma-0: An imperative language that supports declarative programming. ACM Toplas **20** (1998) 1014–1066
8. Grabmüller, M.: Constraint Imperative Programming. Diploma Thesis, Technische Universität Berlin (2003)
9. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs. 2nd edn. MIT Press (1996)
10. Wilson, P.R.: An introduction to Scheme and its implementation. World Wide Web (2003) `http://www.cs.utexas.edu/users/wilson/schintro/schintro_toc.html`, last visited: 2003-02-24.
11. 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)
12. Freeman-Benson, B.N., Borning, A.: The design and implementation of Kaleidoscope'90, a constraint imperative programming language. In: Proc. of the IEEE Computer Society 1992 Int'l Conference on Computer Languages. (1992) 174–180
13. Pepper, P.: Funktionale Programmierung in OPAL, ML, HASKELL und GOFER. 2nd edn. Springer (2003)
14. Puget, J.F.: A C++ Implementation of CLP. In: Proceedings of the Second Singapore International Conference on Intelligent Systems, Singapore (1994)