

Constraints, Predicates, Functions and a Turtle

Stephan Frank, Martin Grabmüller, Petra Hofstedt and André Metzner
{sfrank, magr, ph, ame}@cs.tu-berlin.de

Fakultät IV – Elektrotechnik und Informatik
Technische Universität Berlin, Germany

Abstract. This paper briefly introduces the current work on multiparadigm programming languages and systems at Berlin University of Technology. Starting from a general approach for the cooperation and coordination of constraint solvers and its implementation *Meta-S*, we briefly discuss how to integrate different host languages into the framework building at this multiparadigm constraint languages. Further current research directions concern the functional logic programming language *Brooks* which allows the program compilation according to different narrowing strategies and the constraint imperative language *TURTLE*.

1 Introduction

This paper reports on the current work on multiparadigm programming languages and systems at Berlin University of Technology.

We started with a general approach for the combination of constraint domains and the cooperation and coordination of constraint solvers which is briefly reintroduced in Sect. 2. *Meta-S* is a practical implementation of this approach equipped with an extensible strategy description language. We sketch on this system in Sect. 3.

Interestingly, our general cooperation framework offers a further promising perspective concerning the building of multiparadigm programming languages. Considering language evaluation mechanisms as constraint solvers it is possible to integrate different host languages into the cooperation framework. This approach allows to build multiparadigm constraint languages customized for particular problem requirements for comfortable modeling and problem solving. In Section 4 we discuss this idea.

Further current research directions of our group concern the functional logic programming language *Brooks* which allows the program compilation according to different narrowing strategies and the constraint imperative language *TURTLE* introduced in Sect. 5 and 6, resp.

2 Cooperating Solvers

In [Hof00] a framework for cooperating constraint solvers has been introduced. It allows the integration of arbitrary solvers providing typical interface functions

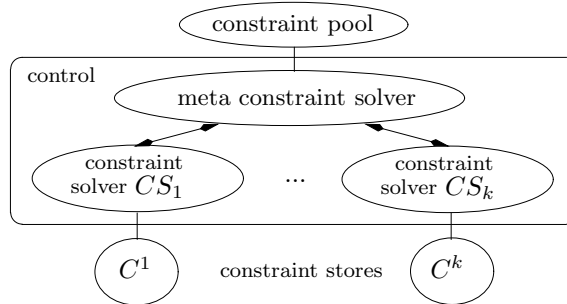


Fig. 1. General architecture for cooperating constraint solvers.

and the handling of hybrid constraints which none of the single solvers is able to handle alone.

Figure 1 shows the architecture of the system. It consists of a number of constraint solvers CS_ν , $\nu \in L^1$, with associated constraint stores C^ν , a meta constraint solver, and a constraint pool. The meta solver coordinates the work of the individual solvers. Their stores hold the constraints which have been propagated previously. The pool is a set containing the constraints which still have not been considered. Given a constraint conjunction to solve, the pool initially contains all these constraints. Step by step the constraints from the pool are propagated by the associated solvers to their stores. For information exchange between the solvers the stores are projected which may yield new information for other solvers in form of new constraints. These are put back into the pool and propagated later. This procedure continues until no more information exchange takes place. In this case, the contents of the stores and of the pool express whether the initially given constraint conjunction was unsatisfiable or not and provide restrictions of the solution space via projections of the stores, in particular cases even solutions of the initial problem. In [Hof00] the system is formally described and it is shown how to define cooperation strategies for the solvers.

Of particular importance for the generality of the approach are the requirements for the interfaces of the solvers to integrate into the system. For every solver CS_ν , first, a function $tell_\nu$ used to propagate constraints from the pool to its store C^ν , and, second, a set of functions $proj_{\nu \rightarrow \mu}$, $\mu \in L$, which provide information from a store C^ν for another solver CS_μ for information interchange is required. The formal definition of the requirements for the interface functions as well as examples of its usage are given in [Hof00,Hof01].

3 Meta-S

A practical implementation and extension of this theoretical framework is Meta-S [FM02,FHM03]. It provides an extensible strategy specification language,

¹ L denotes the set of indices of constraint solvers.

including pattern matching and constraint rewriting facilities, and serves as a testbed for generic and problem specific (meta-)solving strategies, which are employed to minimize the cooperation overhead incurred.

The structure of the Meta-S system was kept highly modular to allow changes of particular modules and the addition of new components in an easy way. The three main modules are the *Base system*, the *Meta module* and any number of pluggable constraint solvers.

The Base system module provides the common substrate of the whole Meta-S system including the external and internal representations of constraints, facilities for translating between those, a syntax extension facility, a pattern matching facility for constraints, and the abstract interface between the framework and individual attached constraint solvers.

Using this abstract solver interface, we have currently interfaced several external and internal constraint solvers: linear arithmetic and interval arithmetic solvers, finite-domain solvers for rationals, floats and strings, and a Common Lisp solver toolkit that enables the generation of residuation-based solvers from Common Lisp functions.

The Meta module includes the meta-solver proper, user facilities to support the command line interface and programmatic macros to ease the integration of the meta-solver system into other applications. Here also resides the strategy framework which we will consider now in more detail.

Generic Strategies. A general strategy class reflects the common requirements of the general cycle of operations of the cooperating solver system. It can be instantiated resp. its parts can be overwritten accordingly for particular strategy requirements.

Three general strategies “eager”, “lazy” and “heuristic” have already been implemented demonstrating the flexibility of the abstract strategy class which mainly differ in the handling of constraint disjunctions.

Benchmarking these strategies with a set of typical problems made important gains of heuristic strategy over the other strategies observable [FM02]. Besides this, the experiments comment that large gains can be achieved through the incorporation of problem specific information (e.g. variable projection order) into the solving process. Thus, a strategy language was developed in order to allow the user to devise more efficient strategies based on existing generic ones and particular problem specific knowledge.

Strategy Language The strategy language allows the user to define strategies by overriding methods of a base strategy.

The language offers primitives for the propagation of sets of constraints, as well as the projection of individual or all solvers against a given set of variables, it supports positional pattern matching for selection, destructuring and rewriting of constraints, and it provides all normal Common Lisp constructs including all kinds of normal control and data flow constructs as well as abstraction features like macros since the strategy language is designed as an extension to the underlying CL system.

```

(define-strategy heuristic-solver-flow
  (heuristic-strategy)
  (:step
   (select ((eq-constraints (= t t))
           (in-constraints (in t t))
           (rest t))
          (tell-all in-constraints)
          (tell-all eq-constraints)
          (tell-all rest)
          (project-one my-linear)
          (tell-all)
          (project-all))))

```

Fig. 2. Strategy specification for the heuristic-solver-flow strategy.

An example for a strategy specification is given in Fig. 2. This strategy is used for a class of problems solved cooperatively by a linear arithmetic solver and a finite domain solver. The strategy “heuristic-solver-flow” is based on the heuristic strategy. It prioritizes domain and equality constraints over other constraints, and takes advantage of the fact that it is mostly the linear arithmetic solver (“my-linear”) which generates restrictions, whereas the finite-domain solver mostly just generates all possible combinations.

Significant performance improvements can be gained by user-defined strategies compared to the generic ones (see [FHM03]).

4 Declarative Languages as Solvers

Besides the possibility to let arbitrary constraint solvers cooperate, the introduced general cooperation framework offers a further very interesting perspective concerning the creation of multiparadigm programming languages, i.e. languages which combine different programming paradigms, such as functional, logic, constraint or imperative ones. The idea of a multiparadigm language is to increase expressiveness and problem-solving power such that the programmer can use a wide range of styles and language features from different paradigms.

Considering declarative programs together with the language evaluation mechanisms as constraint solvers we are able to integrate them into our overall framework of cooperating solvers. Within this framework it is then possible to extend the declarative languages by constraint systems and, thus, to build multiparadigm constraint languages customized for a given set of requirements for comfortable modeling and solving of many problems.

Comparing the evaluation of declarative programs on the one hand and constraint solving on the other hand, our main observations are that the evaluation of expressions in declarative languages consists of their stepwise transformation to a normal form while particular knowledge (substitutions) is collected, and that this is similar to a stepwise propagation of constraints to a store which is at

this simplified. This opens a perspective for a smooth integration of declarative languages and constraints by treating them in a unique way.

We identified *four general steps to combine a declarative language and constraints*:

1. For considering a language as a constraint solver, first of all one needs to identify the constraints of the language itself.
2. The next step concerns the extension of the language by constraints of other domains in a way such that a recursive generation of constraints during program evaluation is possible.
3. Furthermore, the original language evaluation mechanism, e.g. resolution in logic programming or narrowing for functional logic languages, must be extended by gathering constraints of the further integrated solvers. Besides this, the language evaluation mechanism itself remains unchanged.
4. Finally, it is necessary to define the interface functions *tell* and *proj* for the “language solver” according to the requirements of the general solver cooperation framework. This definition will usually base on the extended language evaluation mechanism defined in the previous step. The functions *tell* and *proj* are used to integrate the language evaluation mechanism into the framework and, thus, will reflect the stepwise evaluation of programs.

While these steps describe the strategy for building multiparadigm constraint languages on a very abstract level, in [Hof02] we applied this method at the integration of a logic language with a constraint system which yields a constraint logic programming language according to [JMMS98] and at the extension of a functional logic language with constraints.

5 Brooks

A language intended for such an integration is the functional logic programming language Brooks [Met02,HM03]. Functional logic languages combine the two main declarative paradigms and thereby bring together deterministic computations from the functional world and nondeterministic search operations from the logic world. Brooks inherits from the languages Curry [HAK⁺02] and BABEL [MNRA92], but in contrast allows the integration of different narrowing strategies.

Considering languages as constraint solvers as described in the previous Sect. 4, programs are not evaluated as a whole, but in a stepwise fashion under the strategic control of the meta solver. At this it is not a priori clear how meta solving and evaluation mechanisms interact. Thus, deviating from other existing functional logic languages, Brooks offers different narrowing strategies. This provides one basis for the integration of the language into the solver cooperation framework.

In the following, we briefly present the language Brooks and touch its implementation and evaluation strategies.

As usual for functional and functional logic languages a program consists of data type definitions and function definitions. Brooks utilizes a Haskell-like syntax. It has some built-in data types including `Nat` and `Bool` as well as a data type `Constraint` with the single constructor `Success`. It is used for instance in the context of conditional rules where guard expressions are checked for satisfiability (cf. Curry). Further features of Brooks are the off-side rule, infix operators, and currying along with higher-order functions. The example program in Fig. 3 can't show every detail but may give a general impression.

```

data Seq = Nil | Nat : Seq          -- Sequences of natural numbers

(==:=) :: Seq -> Seq -> Constraint -- Equality constraint for sequences
Nil    ==:= Nil                    = Success
(x : xs) ==:= (y : ys) | x ==:= y = xs ==:= ys

(++ ) :: Seq -> Seq -> Seq         -- Concatenation of sequences
Nil    ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)

last :: Seq -> Nat                -- Get "logically" the last element
last xs | (ys ++ (x : Nil)) ==:= xs = x where ys :: Seq
                                           x :: Nat

```

Fig. 3. Brooks code to (inefficiently) extract the last element from a sequence using a search-oriented technique. The idea is that x is the last element of xs if there exists a sequence ys so that x appended to ys is equal to xs . Note that the equality constraint (`==:=`) for `Nat` is predefined.

The implementation of Brooks is based on an abstract machine which supports evaluation by innermost, lazy, and needed narrowing.

Innermost narrowing is the simplest of the three strategies and often more efficient than lazy narrowing but is incomplete even if the corresponding term rewriting system is confluent and terminating. It does not allow for infinite data structures and impedes certain elegant programming techniques. Therefore modern implementations of functional-logic programming languages favour *lazy narrowing* [MNRA92] or *needed narrowing* [HAK⁺02]. Needed narrowing is optimal wrt. the length of derivations and the number of computed solutions but in its original form can only be applied to a restricted class of programs, called inductively sequential systems [AEH00].

The choice between different evaluation strategies is in principle made via a compile time switch except that currently needed narrowing gets a special treatment. It deviates from the other strategies in that the pattern unification process doesn't work rule-oriented but is guided by *definitional trees* [Ant92]. A definitional tree is a particular hierarchical structure to describe the order in which arguments are to be inspected.

Brooks programs are translated according to the desired evaluation strategy into code for a unified abstract machine BEBAM (**B**rooks **E**xtended **B**ABEL **A**bstract **M**achine, [Met02]) which can be regarded as an extension of a narrowing machine originally designed for innermost and lazy narrowing [Loo95]. It provides additional support for needed narrowing.

The extension of Brooks by constraints and its integration into the solver cooperation framework using the ideas described in Sect. 4 is a topic of future research.

6 Turtle

A further work [Gra03] investigates the combination of constraint-based and imperative programming which is called constraint imperative programming (CIP). CIP tries to join the advantages of both paradigms and, thus, to provide them to the user, i.e. the more effective development and increased program correctness of declarative programs on the one hand and the efficiency and the wide acceptance of imperative programming languages in industry and academia on the other hand.

Grabmüller [Gra03] introduces the constraint imperative programming language TURTLE. It was developed by extending an imperative base language by concepts for constraint programming. The imperative subset of TURTLE consists of all programming constructs necessary for imperative, structured programming including imperative control structures, variables, functions and a rich type system allowing the definition of algebraic recursive data types by the user in a functional style. The language moreover integrates further language constructs known from functional programming languages, like higher-order functions and polymorphic modules.

This base language was then enriched with four concepts necessary for extending it to a constraint imperative language, i.e. by constrainable variables, constraint statements, user-defined constraints and constraint solvers.

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 annotating the type of the variable with an exclamation mark in the variable definition. In contrast to constrainable variables, normal, imperative variables are as usual set to values by assignment statements and cannot be determined by constraints. If such a variable appears in a constraint, its value is taken as a constant when the constraint is solved.

Constraint statements allow the programmer to place constraints on constrainable variables using a require statement. It normally consists of a constraint conjunction and a body which is a sequence of statements. The constraint conjunction is enforced as long as the body executes, i.e. the built-in constraint solver tries to satisfy the constraints by adding them to its store. When the solver is able to solve the conjunction of the store together with the newly added constraint, each constrainable variable gets assigned a value such that the values of all constrainable variables form a solution of the store. When the body of the

constraint statement is left, all constraints added for that statement are removed from the store.

```
var x: int := 2;
var y: !int := var 0;
require y>=x and y=1 : weak in
  io.put(!y); // prints "2"
end;
```

Fig. 4. A TURTLE constraint statement.

In Fig. 4, first a normal variable x and a constrainable variable y are declared. The variable x is initialized with 2, the value 0 is assigned to y . The constraint statement requires the constraints $y \geq x$ and $y = 1$ to hold during the execution of the following print statement.

TURTLE allows the use of constraint hierarchies. Strengths can be assigned to individual constraints which represent their importance. When no such annotation is given, the strength defaults to the strongest strength, called mandatory. 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. Mandatory constraints always get satisfied by the solvers and if that is not possible, an exception is raised.

In Fig. 4 the first constraint has the strongest strength such that its satisfaction by the solver is stronger desired than that of the second one. This results in a assignment of the value 2 to the constrainable variable y .

Since y is a constrainable variable, the behaviour of the expression $y \geq x$ 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 fulfil 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.

User-defined constraints abstract over constraints in the same way as functions abstract over statements. The purpose of a user-defined constraint is to collect constraints which might be needed in several constraint statements. A typical example for this is the `all_different` constraint (see Fig. 5) which requires all variables of a sequence (the argument of the constraint) to be mutually different. A user-defined constraint can be applied in a `require` statement like base constraints.

The last component necessary for constraint imperative programming is a *constraint solver* which 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 the store and for determining values for the variables on success. The system currently integrates two

```

constraint all_different(l:list of !int)
  while (tl l <> null) do
    var ll: list of !int := tl l;
    while (ll <> null) do
      require hd l <> hd ll;
      ll := tl ll;
    end;
    l := tl l;
  end;
end;

require all_different ([a,b,c,d,e]) in
  ...

```

Fig. 5. The user-defined constraint `all_different` and its usage in a constraint statement.

experimental solvers: a solver for acyclic linear real arithmetic constraints based on the Indigo algorithm [BAFB96] and a finite domain solver over integers.

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.

7 Conclusion

This paper describes the current work on multiparadigm programming languages of our group at Berlin University of Technology.

Multiparadigm programming languages combine different programming paradigms, such as functional, logic, constraint or imperative ones. The idea of a multiparadigm language is to increase expressiveness and problem-solving power such that the programmer can use a wide range of styles and language features from different paradigms.

Starting with a general framework for the combination of constraint domains and the cooperation of constraint solvers we worked out how to extend this “multi-domain” combination to a general method for building multiparadigm constraint languages. Simultaneously with this more abstract approach we investigated particular combined paradigms and developed the functional logic programming language Brooks and the constraint imperative language TURTLE.

Future research work concerns the further development of the existing theories and implemented languages but as well aims at a common model for language paradigm integration.

References

- [AEH00] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.
- [Ant92] S. Antoy. Definitional Trees. In *3rd Int’l Conf. on Algebraic and Logic Programming (ALP’92)*, volume 632 of *LNCS*, pages 143–157. Springer-Verlag, 1992.
- [BAFB96] A. Borning, R. Anderson, and B. Freeman-Benson. The Indigo Algorithm. Technical Report 96-05-01, Dept. of Computer Science and Engineering, University of Washington, 1996.
- [FHM03] St. Frank, P. Hofstedt, and P.R. Mai. Meta-S: A strategy-oriented Meta-Solver Framework. In I. Russel and S. Haller, editors, *16th Int’l FLAIRS Conference*. AAAI, 2003.
- [FM02] St. Frank and P.R. Mai. Strategies for Cooperating Constraint Solvers. Diploma thesis, Berlin University of Technology, 2002. In english.
- [Gra03] M. Grabmüller. Constraint Imperative Programming. Diploma thesis, Berlin University of Technology, 2003. In english.
- [HAK⁺02] M. Hanus, S. Antoy, H. Kuchen, F.J. López-Fraguas, W. Lux, J.J. Moreno-Navarro, and F. Steiner. Curry – An Integrated Functional Logic Language. Language Report, Version 0.7.2, 2002.
- [HM03] P. Hofstedt and A. Metzner. Multiple Evaluation Strategies for the Multiparadigm Programming Language Brooks. In Germán Vidal, editor, *12th International Workshop on Functional and (Constraint) Logic Programming, WFLP’03*, 2003.
- [Hof00] P. Hofstedt. Better Communication for Tighter Cooperation. In *First International Conference on Computational Logic – CL*, volume 1861 of *LNCS*. Springer-Verlag, 2000.
- [Hof01] P. Hofstedt. *Cooperation and Coordination of Constraint Solvers*. PhD thesis, Dresden University of Technology, 2001.
- [Hof02] P. Hofstedt. A general Approach for Building Constraint Languages. In *AI 2002: Advances in Artificial Intelligence*, volume 2557 of *LNCS*. Springer-Verlag, 2002.
- [JMMS98] J. Jaffar, M.J. Maher, K. Marriott, and P. Stuckey. The Semantics of Constraint Logic Programs. *Journal of Logic Programming*, 37:1–46, 1998.
- [Loo95] R. Loogen. *Integration funktionaler und logischer Programmiersprachen*. R. Oldenbourg Verlag, 1995.
- [Met02] A. Metzner. Eine abstrakte Maschine für die (schrittweise) Abarbeitung funktional-logischer Programme. Diploma thesis, Berlin University of Technology, 2002. In german.
- [MNRA92] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.