TLP **7** (1&2): 93–121, 2007.
doi:10.1017/S1471068406002833 Printed in the United Kingdom

Integration of declarative and constraint programming

PETRA HOFSTEDT and PETER PEPPER
Technische Universität Berlin, Sekr. FR 5-13,
Franklinstr. 28/29, D-10587 Berlin, Germany
(*e-mail:* {ph,pepper}@cs.tu-berlin.de)

submitted 18 December 2003; revised 20 February 2005, 25 November 2005; accepted 13 January 2006

Abstract

Combining a set of existing constraint solvers into an integrated system of cooperating solvers is a useful and economic principle to solve hybrid constraint problems. In this paper we show that this approach can also be used to integrate different language paradigms into a unified framework. Furthermore, we study the syntactic, semantic and operational impacts of this idea for the amalgamation of declarative and constraint programming.

KEYWORDS: declarative languages, constraints, cooperative constraint solving, language integration, multiparadigm constraint programming languages

1 Introduction

Declarative programming languages base on the idea that programs should be as close as possible to the problem specification and domain. In particular, the semantics of the computation does not depend on the concepts of time and state. Programs of these languages usually consist of directly formulated mathematical objects, i.e. of predicates and functions in logic and functional (logic) languages resp. which are used to describe properties of problems and required solutions.

Our point of view is to *consider declarative programming as constraint programming:* Syntactically this is evident. Logic languages are based on predicates, and goals (on these predicates) are constraints. For functional languages the underlying equality relations can be regarded as constraints as well. But this point of view also applies to the (operational) semantics: the evaluation of expressions in logic and functional languages consists of their stepwise transformation to a normal form, while particular knowledge is collected (in the form of substitutions). This corresponds to a stepwise propagation of constraints.

This kind of consideration opens an interesting potential: In (Hofstedt 2000b) a framework for cooperating constraint solvers has been introduced which allows the integration of arbitrary solvers and the handling of hybrid constraints. Considering declarative programming as constraint programming and looking at the language evaluation mechanisms as constraint solvers we are able to integrate these solvers into this framework. Within the framework it is then possible to extend the declarative languages by constraint systems and, thus, to build constraint languages customised for a given set of requirements for a comfortable modelling and solving of many problems. The paper elaborates this approach.

We start with a recapitulation of necessary concepts and elements of the syntax and semantics of declarative languages in Section 2. Section 3 reintroduces the framework for cooperating solvers of Hofstedt (2000b). In Section 4 we examine the integration of a functional logic language and consider the approach w.r.t. a logic language. We conclude our paper with a discussion of the gains and perspectives of the approach in Section 5 and compare it with related work.

2 Declarative programming languages

Declarative languages are roughly distinguished into functional, logic, and constraint programming languages.

All declarative languages base on the concepts of signatures and terms.

Definition 2.1. A *signature* $\Sigma = (S, F, R)$ consists of a set S of sorts, a set F of S-sorted function symbols and a set R of S-sorted predicate symbols. R contains in particular equality symbols for every sort $s \in S$ and the predicate symbols *true* and *false*.

By $\mathscr{T}(F, X)$ we denote the usual set of *free F-terms*, short: *terms*, over the set X of S-sorted variables. Variable-free terms are called *ground terms*. Expressions $r(t_1, \dots, t_n)$ with terms t_i and a predicate symbol $r \in R$ as outermost symbol are called *predicate terms*. By $\mathscr{T}(\Sigma, X)$ we denote the set of terms and predicate terms.

Given a Σ-*structure* $\mathscr{D}$ we obtain the usual notions of the value of a ground term t in $\mathscr{D}$ and of the validity of a predicate term p in $\mathscr{D}$, denoted as $\mathscr{D} \vDash p$. $\lhd$

In connection with the evaluation of programs the well-known notions of substitutions and unifiers play a central role. We briefly recall their main aspects.

Definition 2.2. By $t[t']$ we denote a term t with a distinguished subterm t'. (This can be formally defined using either positions or contexts.)

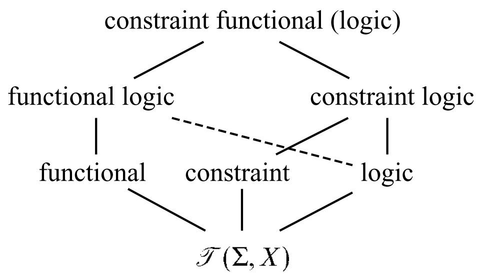
A *substitution* σ is a sort-preserving association $\{x_1 = t_1, \dots, x_m = t_m\}$ from variables $x_i \in X$ to terms $t_i \in \mathscr{T}(F, X)$. (Since it fits more nicely into our overall framework and therefore simplifies some of the later presentations, we write substitutions here as special equations.) The application of a substitution σ to a term or predicate term e is denoted as $\sigma(e)$.

A *unifier* of two terms or predicate terms t and t' is a substitution σ which makes them equal: $\sigma(t) = \sigma(t')$. The most general unifier is denoted as $mgu(t, t')$.

The *composition* of some substitutions σ and ϕ is defined by $(\sigma \circ \phi)(x) = \sigma(\phi(x))$ for all $x \in X$. A substitution σ is *idempotent*, if $\sigma \circ \sigma = \sigma$ holds.

The *parallel composition* $\uparrow$ of idempotent substitutions is defined as in (Palamidessi 1990), i.e. $(\sigma \uparrow \phi) = mgu(\sigma, \phi)$. (Since we consider substitutions as special equations, *mgu* is defined for them.) $\lhd$

Starting from the foundation $\mathcal{T}(\Sigma, X)$ we build a hierarchy of language paradigms.



In the following we briefly sketch the syntax and the underlying ideas of this hierarchy. In the subsequent sections we will then address the much more important issues of the semantic and operational integration of the different paradigms.

Functional programming. The realm of functional programming languages (examples are HASKELL (Hudak *et al.* 2000) and OPAL (Didrich *et al.* 1994)) is inhabited by a plethora of syntactic variations. For our conceptual treatment we can constrain this to a minimal core that is as close as possible to the other kinds of languages that we are treating here. In the following, we distinguish two (disjoint) subsets of the set F of function symbols of the signature Σ: the set $\Delta \subseteq F$ of *constructors* of the underlying data types and the set $\Phi \subseteq F$ of *defined functions*.

DEFINITION 2.3. A *functional program* P over Σ is given by a finite set of rules of the form (called *pattern-based definitions*)

$$f(t_1, \dots, t_n) \to t$$

where $f \in \Phi$ is a defined function and the parameter terms $t_i \in \mathcal{T}(\Delta, X)$ are *linear constructor terms*, i.e. are built up from constructors and variables such that every variable occurs only once. The right-hand side $t \in \mathcal{T}(F, X')$ is an arbitrary F-term, restricted to those variables $X' \subseteq X$ that actually occur on the left-hand side. $\lhd$

The *evaluation* of a functional program P reduces a given *ground term* e using the rules of P until a normal form is obtained (Field and Harrison 1988). Each reduction step picks some function call $f(e'_1, \dots, e'_n)$, that is, a subterm of $e[f(e'_1, \dots, e'_n)]$, which can be unified with the left-hand side of some rule $f(t_1, \dots, t_n) \to t$. The resulting unifier $\sigma = mgu(f(t_1, \dots, t_n), f(e'_1, \dots, e'_n))$ is then applied to the right-hand side t to derive the new term $e[\sigma(t)]$.

Since there may be different applicable rules for a chosen subterm caused by overlapping left-hand sides, the rule selection strategy – e.g. first-fit or best-fit – of the language ensures a deterministic rule choice.

Moreover, there are different *reduction strategies* for picking a redex in each step, for example leftmost-innermost, leftmost-outermost, lazy, etc. These strategies lead to quite different semantics as has already been studied extensively in (Manna 1974). These differences are mainly reflected in the model-theoretic interpretation of the equality $t_1 = t_2$ of the functional domain. For the following illustrating examples an intuitive understanding of equality will do. A thorough elaboration will be given in Sect. 4.

Example 2.1. The following functional program provides rules for the addition of natural numbers which are represented by the constructors 0 and s.

```
add(0,X)     → X                (1)
add(s(X),Y) → s(add(X,Y))       (2)
```

This leads e. g. to the following evaluation sequence, where the reduced subterms are underlined:

$\underline{\texttt{add(s(0),s(s(0)))}} \rightsquigarrow_{(2)} \texttt{s(}\underline{\texttt{add(0,s(s(0)))}}\texttt{)} \rightsquigarrow_{(1)} \texttt{s(s(s(0)))}$ ◁

Since we will use it later on, we present a second example here. It comes from the realm of resistors and their composition.

Example 2.2. For describing the composition of resistors we have three constructor functions: a simple resistor, sequential composition, and parallel composition. The laws of physics entail the program rules:

$$\texttt{rc}(\texttt{simple}(\texttt{X})) \rightarrow \texttt{X}$$
$$\texttt{rc}(\texttt{seq}(\texttt{R1},\texttt{R2})) \rightarrow \texttt{rc}(\texttt{R1}) + \texttt{rc}(\texttt{R2})$$
$$\texttt{rc}(\texttt{par}(\texttt{R1},\texttt{R2})) \rightarrow 1/(1/\texttt{rc}(\texttt{R1}) + 1/\texttt{rc}(\texttt{R2}))$$

This allows e. g. the following evaluation:

$\underline{\texttt{rc}(\texttt{par}(\texttt{simple}(300),\texttt{simple}(600)))} \rightsquigarrow 1/(1/\underline{\texttt{rc}(\texttt{simple}(300))} + 1/\underline{\texttt{rc}(\texttt{simple}(600))})$
$\rightsquigarrow 1/(1/300 + 1/600) \rightsquigarrow 200$ ◁

Functional logic programming. Functional logic programming was originally developed as an extension of functional languages by concepts of logic languages, cf. (Reddy 1985; Loogen 1995). However, they can as well be considered as embedding functional concepts into a logic language, which is indicated by the dashed connection in the above diagram. Typical representatives are BABEL (Moreno-Navarro and Rodríguez-Artalejo 1992) and CURRY (Hanus *et al.* 2003).

Syntactically, a functional logic program looks like a functional program. The difference lies in the evaluation mechanism. Whereas functional programs only allow the reduction of ground terms to normal forms, functional logic programs also allow the solving of equations using residuation (Aït-Kaci and Nasr 1989) and narrowing (Hanus 1994).

A *narrowing step* is a transition $e[l'] \rightsquigarrow_{l \rightarrow r,\sigma} \sigma(e[r])$, where $l \rightarrow r$ is a rule from the program P and l' is a non-variable term (i. e. $l' \notin X$) such that $\sigma = mgu(l', l)$.

Example 2.3. We again use the above addition example. In order to solve the equation $\texttt{add}(\texttt{s}(\texttt{A}),\texttt{B}) = \texttt{s}(\texttt{s}(0))$, we apply narrowing. The chosen subterm is underlined.

$\underline{\texttt{add(s(A),B)}} = \texttt{s(s(0))} \rightsquigarrow_{(2),\{\texttt{X1=A,Y1=B}\}} \texttt{s(}\underline{\texttt{add(A,B)}}\texttt{)} = \texttt{s(s(0))}$
$\rightsquigarrow_{(1),\{\texttt{A=0,X2=B}\}} \texttt{s(B)} = \texttt{s(s(0))}$

Thus, a solution of the initial equation is given by the substitution $\sigma(\texttt{A}) = 0$ and $\sigma(\texttt{B}) = \texttt{s}(0)$ which is computed by unification of s(B) and s(s(0)) after the last narrowing step. ◁

We need to ensure confluence of the rewrite system which is essential for completeness. Thus, the rules of functional logic programs usually must satisfy particular conditions, e. g. linearity of the left-hand sides, no free variables in the right-hand sides and (weak) nonambiguity for lazy languages, cf. (Hanus 1994).

An overview of functional logic programming languages is given in (Hanus 1994), where the following assessment is made: “In comparison with pure functional languages, functional logic languages have more expressive power due to the availability of features like function inversion, partial data structures, and logic variables. In comparison with pure logic languages, functional logic languages have a more efficient operational behaviour since functions allow more deterministic evaluations than predicates.”

It should not come as a surprise that the aforementioned semantic intricacies of functional languages carry over to functional logic programming, leading to notions such as innermost narrowing and the like, cf. (Hanus 1994). But there are more severe problems, which are often ignored in the literature. Consider the example from (Hanus 1994).

EXAMPLE 2.4. Consider the following program

```
f(X) → a
g(a) → a
```

For the equation $\mathtt{f(g(X))} = \mathtt{a}$ innermost narrowing yields the substitution $\{\mathtt{X} = \mathtt{a}\}$ as the only solution, whereas outermost narrowing provides the identity substitution $\{\}$ as solution. However, this depends on the language semantics. In most functional languages the function g would be considered undefined for all arguments but a (since there are no patterns for the other cases). And in a call-by-value semantics this would entail that $\{\mathtt{X} = \mathtt{a}\}$ is the only solution. This illustrates that call-by-name or call-by-need semantics is incompatible with innermost strategies (Manna 1974). These observations will play a role in our later considerations in Sect. 4. $\lhd$

Logic programming. By contrast to functional programming, logic programming is based on predicate terms.

DEFINITION 2.4. A *logic program* P is a set of rules of the form

$$q_0(t_{0,1}, \dots, t_{0,m}) \;\; \texttt{:-} \;\; q_1(t_{1,1}, \;\dots,\; t_{1,n}), \dots, q_k(t_{k,1}, \dots, t_{k,r}), \qquad k \geqslant 0.$$

where the $q_i(\dots)$ are predicate terms and they are called *atoms*. The borderline case $k = 0$, which has no conditions, is called a *fact*. $\lhd$

The *evaluation* of a logic program is based on *resolution*. One starts with a *goal* $(R_1 \wedge \cdots \wedge R_l)$, which is a conjunction of atoms, and adds its negation $(\neg R_1 \vee \ldots \vee \neg R_l)$ to the set of rules. Then one has to find a *refutation*, that is, a sequence of resolution steps which ends with the empty clause $\Box$.

A *resolution step* on a *goal* $G = (R'_1 \wedge \ldots \wedge R'_m)$ and a program P takes one subgoal R'_i and a new variant $r = \; (Q : - Q_1, \dots, Q_n.)$, $n \geqslant 0$, of a rule of P such that R'_i and Q can be unified with most general unifier σ. The result of the resolution step $G \rightsquigarrow_{\sigma,r} G'$ is the new goal $G' = (\sigma(R'_1) \wedge \ldots \wedge \sigma(R'_{i-1}) \wedge \sigma(Q_1) \wedge \ldots \wedge \sigma(Q_n) \wedge \sigma(R'_{i+1}) \wedge \ldots \wedge \sigma(R'_m))$.

If a refutation can be computed, i.e. $P \vDash \exists(R_1 \wedge \ldots \wedge R_l)$ holds, then the computation yields an *answer substitution* ϕ (as composition of the unifiers computed in the resolution steps) such that $P \vDash \forall\, \phi(R_1 \wedge \ldots \wedge R_l)$ holds. For a detailed description of logic programming and its well-known representative PROLOG see for example (Nilsson and Małuszyński 1995).

Constraint programming. Here problems are specified by means of *constraints*, that is, first order formulas which express conditions or restrictions describing properties of objects and relations between them. Constraints come from constraint systems:

DEFINITION 2.5. Let $\Sigma = (S, F, R)$ be a signature such that R contains at least a predicate symbol $=^s$ for every sort $s \in S$. Let X be a set of Σ-variables. Let $\mathscr{D}$ be a Σ-structure with equality, i.e. for every predicate symbol $=^s$ there is an according predicate which is an equivalence relation and fulfils the following requirements:

For all $f \in F$, $r \in R$ and all terms $t_i, t_i' \in \mathscr{T}(F, X)$ of appropriate sorts s_i:
If for all i: $\mathscr{D} \vDash \forall(t_i =^{s_i} t_i')$, then

- $\mathscr{D} \vDash \forall(f(t_1, \ldots, t_n) =^s f(t_1', \ldots, t_n'))$, when $f(t_1, \ldots, t_n)$ and $f(t_1', \ldots, t_n')$ are both defined, or both terms are undefined,
- $\mathscr{D} \vDash \forall(r(t_1, \ldots, t_m) \leftrightarrow r(t_1', \ldots, t_m'))$, when $r(t_1, \ldots, t_n)$ and $r(t_1', \ldots, t_n')$ are both defined, or both terms are undefined.

A basic *constraint* is of the form $r(t_1, \ldots, t_m)$, where $r \in R$ and $t_i \in \mathscr{T}(F, X)$. The set of basic constraints over Σ is denoted by $\mathscr{C}onstraint$. It contains the two distinct constraints *true* and *false* with $\mathscr{D} \vDash true$ and $\mathscr{D} \nvDash false$.

A *constraint system* is a 4-tuple $\zeta = (\Sigma, \mathscr{D}, X, \mathscr{C}ons)$, where $\{true, false\} \subseteq \mathscr{C}ons \subseteq \mathscr{C}onstraint$. $\lhd$

EXAMPLE 2.5. A typical example are constraint systems for linear arithmetic with constraints that are equalities and inequalities, e.g. $\mathtt{X} + 2 * \mathtt{Y} = 7$ and $\mathtt{X} \leqslant 3.5$.

In this realm the signature Σ usually contains the function symbols $+, -, *, /$ and the relation symbols $=, \geqslant, \leqslant$. $\lhd$

The *evaluation* of constraints is handled by *constraint solvers*. These are sophisticated algorithms for particular application domains, for example the simplex algorithm for linear arithmetic. Constraint solvers can not only check the satisfiability of constraints but can also compute entailed constraints, projections and even solutions. A *solution* of a constraint is a valuation which satisfies it.

In order to allow more convenient programming, constraint systems can be embedded into an appropriate language, which provides concepts like recursion, encapsulation and abstraction. This leads to constraint logic and constraint functional (logic) programming.

Constraint logic programming. Logic programs are extended by constraints such that the right-hand side of a rule may not only contain atoms but also constraints from an arbitrary constraint system. Consequently, the evaluation mechanism of logic languages, viz. resolution, has to be extended by mechanisms for collecting

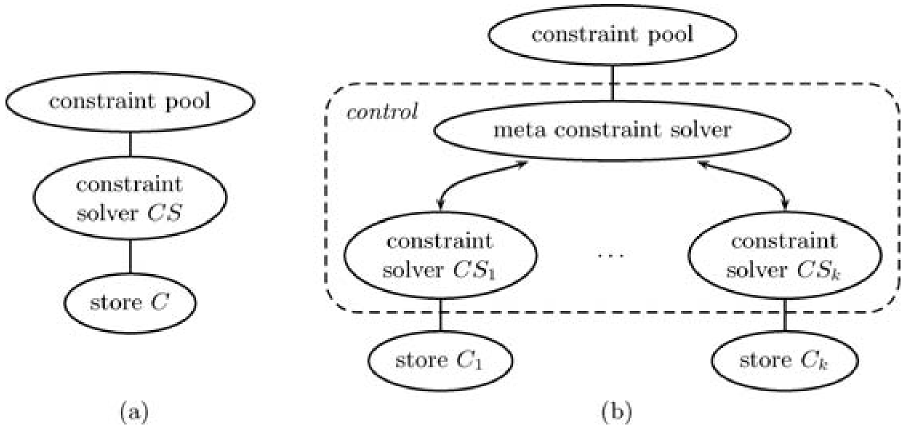


Fig. 1. General architecture for (cooperating) constraint solvers.

constraints and checking their satisfiability using appropriate constraint solvers (Jaffar *et al.* 1998).

The first and also the most typical language which has been extended by constraints, was the logic programming language PROLOG ; the initial motivation was to overcome the limitations of the expressive power of the language when reasoning about arithmetic. A typical and comfortable constraint logic programming system is ECL^{i}PS^e (Cheadle *et al.* 2003).

Constraint functional (logic) programming. Functional (logic) languages can be extended further by *guarding* the rules with sets of constraints.

DEFINITION 2.6. A *constraint functional (logic) program P* over Σ is given by a finite set of rules of the form

$f(t_1, \ldots, t_n) \to t$ **where** G

where – as in functional logic programs – $f \in \Phi$, $t_i \in \mathscr{T}(\Delta, X)$ and $t \in \mathscr{T}(F, X)$. But now we have in addition a set G of (basic) constraints over Σ and X. $\lhd$

For example, TOY($\mathscr{FD}$) (Fernández *et al.* 2003) and TOY($\mathscr{R}$) (Hortalá-González *et al.* 1997) allow finite domain constraints and real arithmetic constraints, resp.

An example of a constraint functional logic program and its evaluation will be considered informally in Section 3.1. The evaluation mechanism is discussed in detail and formally in Section 4.

3 Cooperating constraint solvers

The basic architecture of a simple constraint solving system is a *solver algorithm CS* associated to a *constraint store C* and a *constraint pool*; both are sets of (basic) constraints (see Figure 1 (a)).

- Initially the constraint store of a solver is empty, more precisely: it contains only the constraint *true*; the constraint pool contains the constraints to solve.
- By the so-called *constraint propagation* the solver adds constraints from the pool to its store while ensuring that the constraints in the store remain

https://doi.org/10.1017/S1471068406002833 Published online by Cambridge University Press

satisfiable. In this way the set of possible valuations for the variables of the constraint store is successively narrowed.
- If the solver detects an inconsistency, the corresponding constraint is rejected.

When *constraints from different realms* shall be used together, one has two possibilities. Either one programs a new solver that is capable of handling all kinds of constraints. Or one takes several existing solvers, one for each realm, and coordinates them by some mediating program. The former approach usually generates more efficient solvers, but the amount of implementation work becomes prohibitive, when more and more kinds of constraints shall be integrated. Therefore we focus on the second approach, which is more flexible and more economic.

In (Hofstedt 2000a) a framework for cooperating constraint solvers has been introduced and formally described, including cooperation strategies for the solvers. In (Hofstedt 2001) termination and confluence, as well as soundness and completeness restrictions are examined. An implementation of our system META-S is described in (Frank *et al.* 2003a); (Frank *et al.* 2003b).

Figure 1 (b) shows the architecture of our system for cooperating solvers. In the following let L with $\mu, \nu \in L$ denote the set of indices of constraint systems.

- The *stores* C_ν of the individual constraint solvers CS_ν hold the constraints which have been propagated so far. Initially they are all empty.
- The *constraint pool* is again the set of constraints that still need to be considered. Initially it contains the whole constraint problem to be solved.
- The *meta solver* coordinates the work of the individual solvers. It distributes the constraints from the pool to the appropriate solvers, which put them into their stores by *constraint propagation* and use them for their local computation (see the function $tell_\nu$ below). Conversely, constraints in the local stores may be *projected* to the pool in order to make them available as new information to other solvers (see the functions $proj_{\nu \to \mu}$ below).

The process of sequent propagations and projections ends, when no more information exchange takes place. Then the contents of the stores and of the pool together represent the result: it may indicate, whether the initial constraint conjunction was unsatisfiable or not; moreover, restrictions of the solution space are provided by means of projections of the stores. The restrictions may even provide a full solution of the problem.

Details of the cooperation and communication of the involved solvers are determined by the *cooperation strategy* of the solvers. A cooperation strategy may influence the solution process with regard to different criteria. The solver cooperation system META-S which implements our ideas provides a flexible strategy definition framework. One can prescribe particular search strategies, one can formulate choice heuristics for constraints with respect to their gestalt and their domains, one can specify the order of propagation and projection and so forth. For this system it has been shown in (Frank *et al.* 2003a) that appropriate strategies for solver cooperation can yield comfortable performance improvements for various kinds of constraint problems.

At the heart of our approach are the requirements for the *interfaces* by which the solvers are integrated into the system. There are essentially two kinds of operations that constitute this interface (cf. Sections 3.2 and 3.3):

- For every solver CS_ν there is a function $tell_\nu$ for propagating constraints from the pool to the store C_ν.
- For every pair of solvers CS_ν, CS_μ there is a function $proj_{\nu\to\mu}$ for providing information from the store C_ν to the solver CS_μ (via the constraint pool). Note that this entails the translation into the other solver's signature.

3.1 An example

To give a better intuition about the working of our approach, we present a small example. It is taken from the realm of constraint functional logic programming and illustrates the interaction of a functional logic language with a finite domain solver and a solver for interval arithmetic.

Example 3.1. The following program[1] describes resistors from a certain set $\{300\,\Omega, \dots, 3000\,\Omega\}$, as well as the formulas for the sequential and parallel composition of resistors. The formulation is a mixture of functional logic programming and constraint programming. The first constraint uses the membership test $\in_{\mathscr{FD}}$ from a constraint system for finite domains, and the other two constraints use the equality $=_{\mathscr{A}}$ from a constraint system over rational arithmetic. The equality $=_{\mathscr{FL}}$ comes from the functional logic resolution mechanism.

$$\begin{array}{ll}
\texttt{rc(simple(X))} & \to \texttt{X}\ \textbf{where}\ \texttt{X} \in_{\mathscr{FD}} \{300, 600, 900, 1200, \dots, 2700, 3000\} \\
\texttt{rc(seq(R1,R2))} & \to \texttt{Z}\ \textbf{where}\ \texttt{X} + \texttt{Y} =_{\mathscr{A}} \texttt{Z},\quad \texttt{X} =_{\mathscr{FL}} \texttt{rc(R1)},\quad \texttt{Y} =_{\mathscr{FL}} \texttt{rc(R2)} \\
\texttt{rc(par(R1,R2))} & \to \texttt{Z}\ \textbf{where}\ 1/\texttt{X} + 1/\texttt{Y} =_{\mathscr{A}} 1/\texttt{Z},\quad \texttt{X} =_{\mathscr{FL}} \texttt{rc(R1)},\quad \texttt{Y} =_{\mathscr{FL}} \texttt{rc(R2)}
\end{array}$$

Note that the various subterms (including equations) in this program are all homogeneous, that is, they are in $\mathscr{T}(\Sigma, X)$ for the signature of one of the underlying solvers. This is possible in general: by introducing auxiliary variables we can always turn hybrid terms into homogeneous terms; this is called flattening of terms and constraints, resp. Again, this relies on an appropriate semantic definition of the functional logic equality $=_{\mathscr{FL}}$ (which will be discussed in Sect. 4).

In the following we sketch the way, in which our approach may handle the above program. For the time being, the treatment is on a more intuitive basis. The formalisation of the various steps will be given subsequently. We use the following notation to illustrate the snapshots from the evaluation, where the three stores $C_{\mathscr{FL}}$, $C_{\mathscr{FD}}$ and $C_{\mathscr{A}}$ belong to the functional logic language solver, the finite domain solver, and the arithmetic solver, resp.

ConstraintPool		
store of $CS_{\mathscr{FL}}$	*store of* $CS_{\mathscr{FD}}$	*store of* $CS_{\mathscr{A}}$

[1] We use so-called extra variables in the rules, i. e. variables which occur in the body but not in the head. We discuss the issue of completeness in presence of extra variables briefly in Sect. 4.1.

(1) Suppose we want to compose two resistors in parallel and need a combined resistance of 200 Ω. The question is, which resistors do we have to pick from our set. This question is formalizable in our program as the equation $\mathtt{rc}(\mathtt{par}(\mathtt{RA}, \mathtt{RB})) =_{\mathcal{FL}} 200$. This leads to the following initial configuration.

$\mathtt{rc}(\mathtt{par}(\mathtt{RA}, \mathtt{RB})) =_{\mathcal{FL}} 200$		
true	true	true

(2) We apply the third rule (using narrowing) and reach the following system state.

$\mathtt{Z} =_{\mathcal{FL}} 200, \quad 1/\mathtt{X} + 1/\mathtt{Y} =_{\mathcal{A}} 1/\mathtt{Z}, \quad \mathtt{X} =_{\mathcal{FL}} \mathtt{rc}(\mathtt{RA}), \quad \mathtt{Y} =_{\mathcal{FL}} \mathtt{rc}(\mathtt{RB})$		
true	true	true

(3) When there are several constraints in the pool, the particular cooperation strategy of the system decides, which constraint to choose next for propagation, here e. g. the goal $1/\mathtt{X} + 1/\mathtt{Y} =_{\mathcal{A}} 1/\mathtt{Z}$. We propagate it (using the function $tell_{\mathcal{A}}$) to the arithmetic solver $CS_{\mathcal{A}}$. This is followed by a propagation of $\mathtt{Z} =_{\mathcal{FL}} 200$ to the store of $CS_{\mathcal{FL}}$. (In the pertinent stores we can omit the index of the equality symbol.)

$\mathtt{X} =_{\mathcal{FL}} \mathtt{rc}(\mathtt{RA}), \quad \mathtt{Y} =_{\mathcal{FL}} \mathtt{rc}(\mathtt{RB})$		
$\mathtt{Z} = 200$	true	$1/\mathtt{X} + 1/\mathtt{Y} = 1/\mathtt{Z}$

(4) Next, the system chooses the goal $\mathtt{X} =_{\mathcal{FL}} \mathtt{rc}(\mathtt{RA})$ for a narrowing step based on the rules of our program (using the function $tell_{\mathcal{FL}}$). This leads to a disjunction of three possibilities:

$$\begin{array}{l} \ \mathtt{RA} =_{\mathcal{FL}} \mathtt{simple}(\mathtt{X}) \wedge \mathtt{X} \in_{\mathcal{FD}} \{300, 600, \ldots, 3000\} \\ \vee\ \mathtt{RA} =_{\mathcal{FL}} \mathtt{seq}(\mathtt{R1}, \mathtt{R2}) \wedge (\mathtt{X}_1 + \mathtt{Y}_1 =_{\mathcal{A}} \mathtt{X}) \wedge \mathtt{X}_1 =_{\mathcal{FL}} \mathtt{rc}(\mathtt{R1}) \wedge \mathtt{Y}_1 =_{\mathcal{FL}} \mathtt{rc}(\mathtt{R2}) \\ \vee\ \mathtt{RA} =_{\mathcal{FL}} \mathtt{par}(\mathtt{R1}, \mathtt{R2}) \wedge (1/\mathtt{X}_2 + 1/\mathtt{Y}_2 =_{\mathcal{A}} 1/\mathtt{X}) \wedge \mathtt{X}_2 =_{\mathcal{FL}} \mathtt{rc}(\mathtt{R1}) \wedge \mathtt{Y}_2 =_{\mathcal{FL}} \mathtt{rc}(\mathtt{R2}) \end{array}$$

Due to the disjunction, we have to form three instances of our configuration, each representing one of the choices. For lack of space we only present the derivation of the first alternative here.

$\mathtt{RA} =_{\mathcal{FL}} \mathtt{simple}(\mathtt{X}), \quad \mathtt{X} \in_{\mathcal{FD}} \{300, \ldots, 3000\}, \quad \mathtt{Y} =_{\mathcal{FL}} \mathtt{rc}(\mathtt{RB})$		
$\mathtt{Z} = 200$	true	$1/\mathtt{X} + 1/\mathtt{Y} = 1/\mathtt{Z}$

Note that the store $C_{\mathcal{FL}}$ did not change in this step. The application of a program rule causes a replacement of the chosen constraint by new ones according to the right-hand side of the rule. In contrast, an enhancement of the store happens at the propagation of constraints which do not contain defined functions any more, as we will see in Step (6).

(5) Now we apply the same process to the second goal $\mathtt{Y} =_{\mathcal{FL}} \mathtt{rc}(\mathtt{RB})$, again pursuing only the first variant.

$\mathtt{RA} =_{\mathscr{FL}} \mathtt{simple(X)}, \mathtt{X} \in_{\mathscr{FD}} \{300,\dots\}, \mathtt{RB} =_{\mathscr{FL}} \mathtt{simple(Y)}, \mathtt{Y} \in_{\mathscr{FD}} \{300,\dots,3000\}$		
$\mathtt{Z} = 200$	`true`	$1/\mathtt{X} + 1/\mathtt{Y} = 1/\mathtt{Z}$

(6) Next we propagate all the constraints of the pool to their associated stores.

`true`		
$\mathtt{RA} = \mathtt{simple(X)}$ $\mathtt{RB} = \mathtt{simple(Y)}$ $\mathtt{Z} = 200$	$\mathtt{X} \in \{300,\dots,3000\}$ $\mathtt{Y} \in \{300,\dots,3000\}$	$1/\mathtt{X} + 1/\mathtt{Y} = 1/\mathtt{Z}$

(7) At this point a system without solver cooperation would terminate the computation and not draw any further conclusions.

But our system enables solver *cooperation*. For the continuation of our example we assume that the cooperation strategy of the system now forces the finite domain solver to project its store by generating bounds for the variables using the function $proj_{\mathscr{FD}\to\mathscr{A}}$.[2] This is followed by a projection of $C_{\mathscr{FL}}$ for the variable Z common to $C_{\mathscr{FL}}$ and $C_{\mathscr{A}}$.

$(\mathtt{Z} =_{\mathscr{A}} 200)$, $(300 \leqslant_{\mathscr{A}} \mathtt{X})$, $(\mathtt{X} \leqslant_{\mathscr{A}} 3000)$, $(300 \leqslant_{\mathscr{A}} \mathtt{Y})$, $(\mathtt{Y} \leqslant_{\mathscr{A}} 3000)$		
$\mathtt{RA} = \mathtt{simple(X)}$ $\mathtt{RB} = \mathtt{simple(Y)}$ $\mathtt{Z} = 200$	$\mathtt{X} \in \{300,\dots,3000\}$ $\mathtt{Y} \in \{300,\dots,3000\}$	$1/\mathtt{X} + 1/\mathtt{Y} = 1/\mathtt{Z}$

(8) The new constraints in the pool are now amenable to treatment by the arithmetic solver. Therefore the meta solver propagates them (using $tell_{\mathscr{A}}$) to this solver. Using its computational capabilities, the arithmetic solver can derive more accurate bounds:

`true`		
$\mathtt{RA} = \mathtt{simple(X)}$ $\mathtt{RB} = \mathtt{simple(Y)}$ $\mathtt{Z} = 200$	$\mathtt{X} \in \{300,\dots,3000\}$ $\mathtt{Y} \in \{300,\dots,3000\}$	$1/\mathtt{X} + 1/\mathtt{Y} = 1/200, \mathtt{Z} = 200$ $(300 \leqslant \mathtt{X})$, $(\mathtt{X} \leqslant 600)$ $(300 \leqslant \mathtt{Y})$, $(\mathtt{Y} \leqslant 600)$

(9) In the following steps the arithmetic solver's improved bounds can be projected back to the pool, from where they are propagated to the finite domain solver, which then narrows down its choices.

`true`		
$\mathtt{RA} = \mathtt{simple(X)}$ $\mathtt{RB} = \mathtt{simple(Y)}$ $\mathtt{Z} = 200$	$\mathtt{X} \in \{300, 600\}$ $\mathtt{Y} \in \{300, 600\}$	$1/\mathtt{X} + 1/\mathtt{Y} = 1/200, \mathtt{Z} = 200$ $(300 \leqslant \mathtt{X})$, $(\mathtt{X} \leqslant 600)$ $(300 \leqslant \mathtt{Y})$, $(\mathtt{Y} \leqslant 600)$

[2] Actually, our implementation META-S (Frank *et al.* 2003a) distinguishes between weak projection generating only constraint conjunctions and strong projection which is allowed to project disjunctions as well (first proposed in (Hofstedt 2001)). Using different kinds of projections we are able to realise a variant of the Andorra principle (Costa *et al.* 1991; Warren 1988) which proved to be very advantageous w. r. t. efficiency. The generation of bounds in our example represents a weak projection.

https://doi.org/10.1017/S1471068406002833 Published online by Cambridge University Press

(10) At this point we need strong projection (cf. footnote (2)) by which the finite domain solver puts a disjunction of equations

$$(\mathtt{X} =_{\mathscr{A}} 300 \wedge \mathtt{Y} =_{\mathscr{A}} 300) \;\vee\; \ldots \;\vee\; (\mathtt{X} =_{\mathscr{A}} 600 \wedge \mathtt{Y} =_{\mathscr{A}} 600)$$

into the pool. Each of these four conjunctions can again be propagated to the arithmetic solver. Two of them will lead to solutions, but the other two propagations lead to inconsistencies in the arithmetic solver and are therefore discarded. One successful final configuration is:

true		
RA = simple(300) RB = simple(600) Z = 200	X = 300 Y = 600	Z = 200 X = 300 Y = 600

The solution $\{\mathtt{RA} = \mathtt{simple}(300), \mathtt{RB} = \mathtt{simple}(600)\}$ can be extracted from the constraint store $C_{\mathscr{FL}}$ of the solver $CS_{\mathscr{FL}}$. $\triangleleft$

This small example already demonstrates the important role of cooperation strategies for the efficiency of the computation. For example, the ability to control the order of weak and strong projections allows a considerable restriction of the set of variable assignments before an explicit search for solutions is initiated for the remaining alternatives. Suppose that we had applied the strong projection at an earlier point. This would have caused a search across all 100 alternatives of resistor combinations, i. e. $(\mathtt{X} =_{\mathscr{A}} 300 \wedge \mathtt{Y} =_{\mathscr{A}} 300), \ldots, (\mathtt{X} =_{\mathscr{A}} 3000 \wedge \mathtt{Y} =_{\mathscr{A}} 3000)$. As a matter of fact, it is in general a good strategy to delay the introduction of disjunctions as long as possible for the sake of efficiency. This is suggested by experiences with the KIDS system (Westfold and Smith 2001).

Based on the intuitive insights provided by this example we will now look more deeply into the precise definitions of the propagation function $tell_\nu$ and the projection function $proj_{\nu\to\mu}$.

3.2 Constraint propagation ($tell_\nu$)

As can be seen e. g. in Step (3) of the above example, the function $tell_\nu$, $\nu \in L$, takes a constraint $c \in \mathscr{C}ons_\nu$ (i. e. a basic constraint of the constraint system of solver CS_ν) from the pool and adds it to the constraint store C_ν, which leads to a new store C'_ν. There may also be a remaining part c'' of c, which is put back into the pool (but this happens rarely in practice).

Example 3.2. Suppose we have the constraint $\sqrt{\mathtt{X}} = \mathtt{Y}$ in the pool. This may be used to put the constraint $\mathtt{X} = \mathtt{Y}^2$ into the store of some solver, while keeping the constraint $\mathtt{Y} \geqslant 0$ in the pool. $\triangleleft$

Figure 2 shows the requirements for the function $tell_\nu$.[3] The function returns three values. The first one is a Boolean indicator of the success or failure. The second one

[3] In (Hofstedt 2000a) two forms of successful propagation are distinguished, which is necessary for general solvers to ensure termination of the system. For the “language solvers” considered in this paper this is simplified.

https://doi.org/10.1017/S1471068406002833 Published online by Cambridge University Press

$tell_\nu : \mathscr{C}ons_\nu \times Store_\nu \longrightarrow \{true, false\} \times Store_\nu \times \mathscr{DC}ons_\nu$ with

1. if $tell_\nu(c, C_\nu) = (true, C'_\nu, c'')$, then

   (a) $\mathscr{D}_\nu \vDash \forall((C_\nu \wedge c) \longleftrightarrow (C'_\nu \wedge c''))$,
   (b) $\mathscr{D}_\nu \vDash \forall(C'_\nu \longrightarrow C_\nu)$,
   (c) $\mathscr{D}_\nu \vDash \exists C'_\nu$,

2. if $tell_\nu(c, C_\nu) = (false, C'_\nu, c'')$, then

   $C'_\nu = C_\nu$, $c'' = false$, $\mathscr{D}_\nu \vDash \neg\exists(C_\nu \wedge c)$.

Fig. 2. Interface function $tell_\nu$, $\nu \in L$, (requirements).

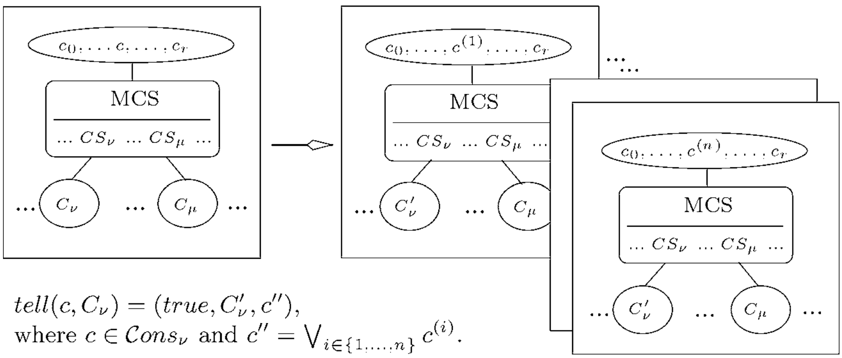


Fig. 3. Application of the interface function *tell*.

is the modified store. And the third one is the remaining constraint $c'' \in \mathscr{DC}ons_\nu$, which is put back into the pool. By $\mathscr{DC}ons_\nu$ we denote the set of disjunctions of constraint conjunctions.

When the solver successfully propagates a constraint c to a store C_ν (Case 1), then it must be ensured that the (overall) knowledge of the store and the constraint is neither lost nor increased (a). It is only possible to add constraints to a store, but not to delete them. Thus, the new constraint store C'_ν must imply the old one (b). Of course, the new store C'_ν has to be satisfiable in the domain $\mathscr{D}_\nu$ of CS_ν as it is a constraint store (c). In Example 3.1, e. g. in Steps (3) and (4), $tell_{\mathscr{A}}$ and $tell_{\mathscr{FL}}$ have been applied according to this definition.

This first case also covers the situation that a solver is not able to handle a certain constraint c, i. e. if the solver is incomplete. In this case the store C_ν does not change and $c = c''$ remains in the pool.[4]

Figure 3 visualises the state change of the system when a solver performs a successful constraint propagation. The left side shows the system before the propagation, the right side afterwards. When we propagate c to C_ν by $tell_\nu(c, C_\nu)$, c is deleted from the pool and propagated to the store C_ν. The resulting new store

[4] Again, to ensure termination of the overall system, this particular case must be detected by the overall machinery and the treatment of the constraint must be suspended. We omit this technical detail in favour of readability.

https://doi.org/10.1017/S1471068406002833 Published online by Cambridge University Press

$proj_{\nu\to\mu}: \wp(X_\nu \cap X_\mu) \times Store_\nu \to \mathcal{DCC}ons_\mu$, where $\mathcal{V}ars(proj_{\nu\to\mu}(X, C_\nu)) = Y \subseteq X$,
must be *sound*, i.e. for every valuation σ_ν for the variables of Y must hold:
If $(\mathcal{D}_\nu, \sigma_\nu) \vDash \exists_{-Y} C_\nu$, then $(\mathcal{D}_\mu, \sigma_\nu) \vDash proj_{\nu\to\mu}(X, C_\nu)$, where
$\exists_{-Y} C_\nu$ denotes the existential closure of formula C_ν except for the variables of Y.

Fig. 4. Interface function $proj_{\nu\to\mu}$, $\mu \neq \nu$, $\mu, \nu \in L$, (requirements).

C'_ν and the remaining constraint c'' may in general be disjunctions of constraint conjunctions, e.g. $c'' = \bigvee_{i\in\{1,\dots,n\}} c^{(i)}$. Since store and pool are sets of basic constraints, this causes a splitting of the system as shown in Figure 3.

If $tell_\nu(c, C_\nu)$ fails (Case 2) because c and C_ν are contradictory, then $false$ is added to the constraint pool and C_ν does not change (not shown in Figure 3).

Example 3.3. The interface function $tell_{\mathscr{A}}$ of our rational arithmetic solver $CS_{\mathscr{A}}$ could work as follows (see e.g. Step (10) in Example 3.1):

Given the store $C = (1/\mathtt{X} + 1/\mathtt{Y} =_{\mathscr{A}} 1/200)$ the propagation $tell_{\mathscr{A}}((\mathtt{X} =_{\mathscr{A}} 300), C) = (true, C', true)$ yields – after some computation by the solver $CS_{\mathscr{A}}$ – a new simplified store $C' = (\mathtt{X} =_{\mathscr{A}} 300 \wedge \mathtt{Y} =_{\mathscr{A}} 600)$. On the other hand, $tell_{\mathscr{A}}((\mathtt{Y} >_{\mathscr{A}} 600), C') = (false, C', false)$ represents a failing propagation. $\lhd$

For each concrete solver CS_ν the user must provide a suitable function $tell_\nu$. However, constraint propagation is mainly based on the satisfiability test which is the main operation of a constraint solver. Moreover, the requirements in Figure 2 are chosen in such a way that they allow an easy integration of many existing solvers into the cooperating system, taking particular properties of solvers (like their incompleteness (cf. Case 1) or an existing entailment test) into consideration.

Examples for such concrete propagation functions will be shown in Sect. 4 for the special cases of functional logic and logic languages, when they are viewed as solvers.

3.3 Projection of constraint stores ($proj_{\nu\to\mu}$)

Projection is used for information exchange between constraint solvers CS_ν and CS_μ, $\nu \neq \mu$, $\mu, \nu \in L$, via the constraint pool.

Figure 4 shows the requirements for the function $proj_{\nu\to\mu}$. The function $proj_{\nu\to\mu}$ takes a set of common variables of the two solvers and the local store C_ν and returns (a disjunction of conjunctions of) constraints to the pool. $proj_{\nu\to\mu}(X, C_\nu) = c$ describes a projection of a store C_ν w.r.t. common variables (i.e. $X \subseteq X_\nu \cap X_\mu$) to provide constraints c of another solver CS_μ. It provides knowledge implied by the store C_ν. Projection does not change the stores but only extends the pool by the projected constraints.

To ensure that finally no solution is lost, a projection c must provide every satisfying valuation of the current store C_ν. That is, $proj_{\nu\to\mu}$ must be defined in such a way that every solution σ_ν of C_ν is a solution of its projection c in $\mathscr{D}_\mu$ (soundness).

Example 3.4. Consider the arithmetic solver $CS_{\mathscr{A}}$ and the finite domain solver $CS_{\mathscr{F}\mathscr{D}}$. Let $C_{\mathscr{F}\mathscr{D}} = (\mathtt{X} \in_{\mathscr{F}\mathscr{D}} \{300, \ldots, 3000\} \wedge \mathtt{Y} \in_{\mathscr{F}\mathscr{D}} \{300, \ldots, 3000\})$ hold. Define $proj_{\mathscr{F}\mathscr{D} \rightarrow \mathscr{A}}$ (as a weak projection, cf. footnote (2)) such that

$proj_{\mathscr{F}\mathscr{D} \rightarrow \mathscr{A}}(\{\mathtt{X}\}, C_{\mathscr{F}\mathscr{D}}) = ((\mathtt{X} \geqslant_{\mathscr{A}} 300) \wedge (\mathtt{X} \leqslant_{\mathscr{A}} 3000))$.

This corresponds to Step (7) of Example 3.1. ◁

As in the case of $tell_{\nu}$ the function $proj_{\nu \rightarrow \mu}$ has to be concretely programmed for every pair of given solvers. For many pairs of solvers it is possible to automatically provide simple projection functions generating equality constraints which at least express variable bindings. Often it is also sufficient to provide actual projection functions only for particular pairs of solvers and to reduce superfluous communication.

Examples of projection functions will also be shown in Section 4 in connection with constraint functional logic and constraint logic languages.

3.4 Semantics

In (Hofstedt 2001) we define reduction systems describing solver collaborations based on the interface functions $tell_{\nu}$ and $proj_{\nu \rightarrow \mu}$. The reduction systems work on so-called configurations which consist of representations of the current constraint pool and the associated stores, which together represent the state of the system.

(Hofstedt 2001) gives a detailed discussion of the termination, confluence, soundness and completeness of the reduction systems.

Based on the signatures and Σ-structures of the incorporated constraint systems we build combined constraint systems by their disjoint unions. Clearly, for those sorts and function and predicate symbols that are common to different systems the corresponding carrier sets, functions and predicates must be identical. This applies in particular to the semantics of the equality symbol (on shared sorts).

It is shown that – using our method – no solutions are lost. If the system fails then the given constraint problem is unsatisfiable. Furthermore, we discuss the situation of suspended constraints remaining in the pool due to the incompleteness of solvers. If all incorporated solvers are complete and the reduction relation is terminating, then for a satisfiable constraint problem the method can be guaranteed to be successful.

The soundness and completeness results do not depend on the particular cooperation strategy of the solvers.

The (obvious) fact that the completeness of the overall system depends on the completeness of the individual solvers will play a role in connection with functional logic languages.

4 Combination of declarative and constraint programming

As a particular application, our system of cooperating solvers allows the integration of different host languages by treating them as constraint solvers. As a matter of fact, it even makes it possible to work quite flexibly with different evaluation strategies, that is, different operational semantics. A good point in case are functional (logic) languages, which have very different semantics depending on the chosen evaluation mechanisms. As mentioned in Sect. 2, this has already been elaborated in the textbook (Manna 1974) for functional languages; for the realm of functional logic programming the pertinent issues are described in (Hanus 1994; Hanus 1995).

The feasibility of this idea is based on two main observations: First, the evaluation of expressions in declarative languages consists of their stepwise transformation to a normal form while particular knowledge (in the form of substitutions) is collected. Second, this way of proceeding is similar to a stepwise propagation of constraints to a store, which is simplified in doing so.

In the following, we consider the integration of a functional logic language and of a logic language, resp., into the system of cooperating solvers. Syntactically, we extend the languages by constraints, but their evaluation mechanisms are nearly unchanged: they are only extended by a mechanism for collecting constraints of the other constraint solvers.

A four-step process. The integration of a declarative language into our system of cooperating solvers requires four activities.

1. The inherent constraints of the language have to be identified.
2. Conversely, the constraints from the other domains have to be integrated into the syntax of the language.
3. The language evaluation mechanism (e.g. reduction or resolution) has to be extended by gathering constraints from the other domains.
4. Finally one needs to carefully define the interface functions $tell_{\cdot}$ and $proj_{\cdot\to\cdot}$ of the new language solver.

4.1 A functional logic language as constraint solver

The introduction of constraints into the rules of a functional logic language yields constraint functional logic programming. We follow our four step process. Recall that the basic syntactic construct is

$$f(t_1, \ldots, t_n) \to t \ \textbf{where}\ G$$

where G is a set of constraints, including constraints from other domains. Moreover, all constraints in G are homogeneous, that is, are built from the signature of one solver, including equalities $t_1 =_{\mathscr{FL}} t_2$ of the functional logic language.

(1) Identifying language constraints. Functional logic languages are based on equalities between terms. In order to state precisely what this means, we have to look more closely into the semantic models of these languages, cf. e.g. (Manna 1974; Winskel 1993; Broy *et al.* 1987). It has to be ensured that the *operational semantics* induced by our cooperating solvers is compatible with the *mathematical semantics* of the language under consideration.

Recall that the semantic value of a term t in the model $\mathscr{D}$ is denoted by $[\![t]\!]_{\mathscr{D}}$; when $\mathscr{D}$ is obvious from the context, we omit it and just write $[\![t]\!]$. We are dealing with partial functions; according to the traditional convention we write "$\perp$" (pronounced "bottom") to express partiality: $[\![t]\!] = \perp$ means: t has no proper semantic value. We note in passing that there are two kinds of partiality, both of which can be subsumed under the element $\perp$. One is nontermination of evaluations, the other is the lack of matching rules. For example, if there is no rule for $\mathtt{X}/0$ – independent of the responsible solver – then we have $[\![t/0]\!] = \bot$. The following discussion is exemplified with nontermination but it applies to both cases.

Example 4.1. Consider the following superficial example over the natural numbers with constructors 0 and $\mathtt{s}$. (To ease reading we write $\mathtt{s}$ without parentheses.)

$$\begin{array}{ll} \mathtt{f}(0,\mathtt{Y}) & \to 0 \\ \mathtt{f}(\mathtt{s}\,\mathtt{X},\mathtt{Y}) & \to \mathtt{f}(\mathtt{X},\mathtt{f}(\mathtt{s}\,\mathtt{X},\mathtt{Y})) \end{array}$$

We consider a functional logic solver $CS_{\mathscr{FL}}$ working in cooperation with others and we use again the notation from Example 3.1 to illustrate the snapshots from the evaluation. The formalisation of the interface functions for $CS_{\mathscr{FL}}$ will be given and explained in more detail subsequently.

Start from the constraint $\mathtt{Z} =_{\mathscr{FL}} \mathtt{f}(0,\mathtt{f}(\mathtt{s}\,0,\mathtt{s}\,0))$ in the pool.

$\mathtt{Z} =_{\mathscr{FL}} \mathtt{f}(0,\mathtt{f}(\mathtt{s}\,0,\mathtt{s}\,0))$		
true	true	true

In a *call-by-name semantics* we first reduce the outermost $\mathtt{f}$ using the first rule of the program. This immediately yields the result: $[\![\mathtt{Z}]\!] =_{\mathscr{FL}} 0$. Semantically this means $[\![\mathtt{Z}]\!] = [\![\mathtt{f}(0,\mathtt{f}(\mathtt{s}\,0,\mathtt{s}\,0))]\!]_{cbn} = [\![0]\!]$.

In a *call-by-value semantics* we first reduce the innermost $\mathtt{f}$ using the second rule: $\mathtt{Z} =_{\mathscr{FL}} \mathtt{f}(0,\mathtt{f}(0,\mathtt{f}(\mathtt{s}\,0,\mathtt{s}\,0)))$. As one can immediately see, this reduction process will never terminate, that is, there will always be an equation $\mathtt{Z} =_{\mathscr{FL}} t_i$ in the pool (with longer and longer right-hand sides t_i). Semantically this means $[\![\mathtt{Z}]\!] = [\![\mathtt{f}(0,\mathtt{f}(\mathtt{s}\,0,\mathtt{s}\,0))]\!]_{cbv} = \bot$. $\lhd$

What does this mean for the introduction of variables? Recall that we need auxiliary variables for the homogeneity of the constraints for different solvers, i. e. we need to flatten constraints and terms. To see the pertinent problems, we modify the above example by introducing several auxiliary variables. Moreover, we add a further function g.

$$\begin{array}{ll} \mathtt{f}(0,\mathtt{Y}) & \to 0 \\ \mathtt{f}(\mathtt{s}\,\mathtt{X},\mathtt{Y}) & \to \mathtt{f}(\mathtt{X},\mathtt{W}) \textbf{ where } \mathtt{W} =_{\mathscr{FL}} \mathtt{f}(\mathtt{s}\,\mathtt{X},\mathtt{Y}) \\ \mathtt{g}(\mathtt{X}) & \to 0 \textbf{ where } \mathtt{f}(\mathtt{s}\,\mathtt{X},\mathtt{s}\,\mathtt{X}) =_{\mathscr{FL}} 0 \end{array}$$

Let us first consider the *operational semantics*. We start from

$\mathtt{Z} =_{\mathscr{FL}} \mathrm{g}(0)$		
true	true	true

After the first two steps this leads to the following situation:

$\mathtt{Z} =_{\mathscr{FL}} 0,\ \mathtt{f}(0,\mathtt{W}) =_{\mathscr{FL}} 0,\ \mathtt{W} =_{\mathscr{FL}} \mathtt{f}(\mathtt{s}\,0,\ \mathtt{s}\,0)$		
true	true	true

In the next steps we may propagate $\mathtt{Z} =_{\mathscr{FL}} 0$ and evaluate the two calls of $\mathtt{f}$:

$\mathtt{W} =_{\mathscr{FL}} \mathtt{f}(0, \mathtt{W}'),\ \mathtt{W}' =_{\mathscr{FL}} \mathtt{f}(\mathtt{s}\,0, \mathtt{s}\,0)$		
$\mathtt{Z} = 0$	true	true

This may continue into the following configuration:

$\mathtt{W}^{(i)} =_{\mathscr{FL}} \mathtt{f}(\mathtt{s}\,0, \mathtt{s}\,0)$		
$\mathtt{Z} = 0,\ \mathtt{W} = 0,\ \mathtt{W}' = 0,\ \ldots$	true	true

This process will obviously continue forever, creating more and more auxiliary variables $\mathtt{W}^{(j)}$.

Let us compare this w. r. t. the original (not flattened) definition of the function $\mathtt{f}$ in Example 4.1 and the two kinds of *mathematical semantics* that we consider here.

- Under *call-by-value* semantics, the result is $[\![\mathrm{Z}]\!] = [\![\mathrm{g}(0)]\!] = \perp$.
- Under *call-by-name* semantics the result is $[\![\mathrm{Z}]\!] = [\![\mathrm{g}(0)]\!] = [\![0]\!]$.

Note that the auxiliary variables $\mathtt{W}$, $\mathtt{W}'$, … do not appear at all in the solution space of the mathematical semantics, since they are only internal artefacts of the computation process.

Coming back to the operational semantics, we can only look up the value of $\mathtt{Z}$ in the store, when the computation is finished, that is, when the pool is empty. In the above example this will never happen; therefore we will never be able to extract the value $\mathtt{Z} = 0$ from the store. This means that $\mathtt{Z}$ has no value, i.e. $[\![\mathtt{Z}]\!] = \perp$. In other words: Without further precautions, the introduction of variables is only compatible with *call-by-value* semantics.

How could we implement *call-by-name* semantics and still allow the introduction of variables? A simple solution consists of the introduction of a dependency relation between variables. In our example, $\mathtt{W}'$ would depend on $\mathtt{W}$, because it occurs in the term on the right-hand side of $\mathtt{W}$. As soon as the pertinent rule would reduce $\mathtt{W} =_{\mathscr{FL}} \mathtt{f}(0, \mathtt{W}')$ to $\mathtt{W} =_{\mathscr{FL}} 0$, the dependent variable $\mathtt{W}'$ would be eliminated from the pool. Moreover, the order of evaluation would have to follow the dependencies.

As can be seen from this sketch, our method of handling hybrid constraints by flattening is more naturally related to *call-by-value* semantics, but with a little effort other semantic principles such as *call-by-name* can be accommodated as well.

We note in passing that these considerations do not only apply to functional logic solvers, but to all kinds of solvers. Consider for example an arithmetic solver with the rule $0 * \mathtt{X} = 0$. If we multiply the term $\mathrm{g}(0)$ from the above example by 0 we obtain from $\mathtt{Z} = 0 * \mathrm{g}(0)$ the two constraints $\mathtt{Z} = 0 * \mathtt{X}_1$, $\mathtt{X}_1 =_{\mathscr{FL}} \mathrm{g}(0)$, which leads to $\mathtt{Z} = 0$, $\mathtt{X}_1 =_{\mathscr{FL}} \mathrm{g}(0)$. Again, the differences between *call-by-name* and *call-by-value* can be captured by introducing a dependency of the variable $\mathtt{X}_1$ on $\mathtt{Z}$.

Remark on strict equality. From a semantic point of view it is unsatisfactory that we express undefinedness only operationally by the fact that the pool does not become empty. In a denotational setting, one would prefer to represent this situation also as $\perp$. This brings strict equality into the game.

In discussions about mathematical semantics there are usually two kinds of equality: Under *strong equality* we have e. g. $(\bot =_{\mathscr{D}} \bot) = true$ and $(\bot =_{\mathscr{D}} 3) = false$. And *strict equality* has the property $[\![t_1 = t_2]\!] =_{\mathscr{D}} \bot$ if $[\![t_1]\!] =_{\mathscr{D}} \bot$ or $[\![t_2]\!] =_{\mathscr{D}} \bot$. Strong equality obeys the laws of classical two-valued logic, but it is in general not decidable. Strict equality is a "normal" operator in the language but the semantics leads to all problems of three-valued logic. Neither equality is "better" than the other, they just serve different purposes.

Call-by-value semantics evidently conforms nicely to strict equality. In our above example we have $[\![\mathtt{Z}]\!] = [\![\mathtt{g}(0)]\!] = [\![\mathtt{f}(\mathtt{s}\,0, \mathtt{s}\,0)]\!] = \cdots = \bot$. Therefore every pool contains some equality of the kind $[\![\mathtt{W}^{(i)} = \mathtt{f}(\mathtt{s}\,0, \mathtt{s}\,0)]\!]$ and thus $[\![\mathtt{W}^{(i)} = \bot]\!]$, which – under strict equality – is $\bot$. Finally, we must consider conjunctions as strict such that the whole configuration is $\bot$. This exactly reflects the fact that our computation does not terminate.[5] The overall system remains semantically consistent, since a "wrong" equation such as $\mathtt{Z} = 0$ is in conjunction with $\bot$ and therefore does no harm. As a matter of fact, the overall configuration just moves in each step from one representation of the value $\bot$ to another representation of the value $\bot$.

The same considerations apply to *call-by-name* semantics. Here we have the situation that e. g. $[\![\mathtt{f}(0, \bot)]\!] = 0$. Therefore an equality like $\mathtt{f}(0, \mathtt{W}) = 0$, where both terms are different from $\bot$, has the same truth value under both kinds of equality. However, if we encounter a function that is nonterminating even under *call-by-name* semantics, then we need again the strict equality in order to represent the overall nontermination by $\bot$.

The fact that the operational semantics requires additional means such as dependencies among variables is independent of the kind of equality. If one considers the above example, the computation is consistent, since it simply keeps adding valid equalities of the kind $\mathtt{W}^{(i)} = 0$ to the store. All we need is a mechanism to stop it from doing this forever.

(2) Extending the language by constraints of other domains. The syntax of the language also contains the constraints from the other domains (occurring in the part ... **where** G of the rules). Therefore they are also part of the language constraints.

To get a clean separation of concerns we also have to flatten hybrid terms from different domains. Suppose, for example, that we had given the third rule of our program in the hybrid form

$$\mathtt{rc}(\mathtt{par}(\mathtt{R1}, \mathtt{R2})) \to \mathtt{Z}\ \textbf{where}\ 1/\mathtt{rc}(\mathtt{R1}) + 1/\mathtt{rc}(\mathtt{R2}) =_{\mathscr{A}} 1/\mathtt{Z}$$

Then we would need to put this into the separated form

$$\mathtt{rc}(\mathtt{par}(\mathtt{R1}, \mathtt{R2})) \to \mathtt{Z}\ \textbf{where}\ 1/\mathtt{X} + 1/\mathtt{Y} =_{\mathscr{A}} 1/\mathtt{Z}, \quad \mathtt{X} =_{\mathscr{FL}} \mathtt{rc}(\mathtt{R1}), \quad \mathtt{Y} =_{\mathscr{FL}} \mathtt{rc}(\mathtt{R2})$$

(3) Extending the language evaluation mechanism by gathering constraints. Functional logic languages are based on *narrowing*. Therefore we consider this evaluation mechanism as constraint solver $CS_{\mathscr{FL}}$. But in addition to performing the narrowing

[5] The formal disjunctions that reflect the various branches of the evaluation need a different semantic treatment in order to accommodate the inherent nondeterminism.

https://doi.org/10.1017/S1471068406002833 Published online by Cambridge University Press

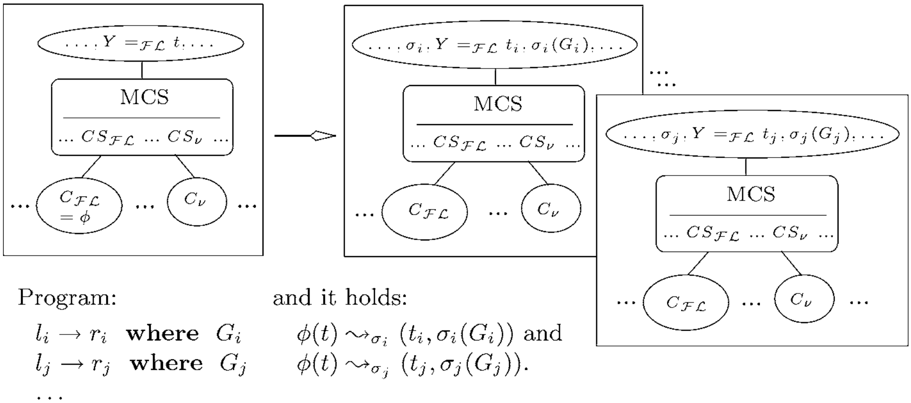


Fig. 5. Application of the interface function $tell_{\mathcal{FL}}$ (see also Figure 6).

steps, the solver must also collect constraints from other domains (occurring in the part ... **where** G).

The basic concept of a *narrowing step with constraints* can be described as follows: Let e be an equation with a distinguished non-variable subterm t, that is, $e[t]$ with $t \notin X$. Let ($l \rightarrow r$ **where** G) be a rule from the program such that $\sigma = mgu(t, l)$ unifies the subterm t with the left-hand side l of the rule. Then the narrowing step yields the new equation $\sigma(e[r])$ together with the rewritten constraint $\sigma(G)$, we write: $e[t] \rightsquigarrow_\sigma (\sigma(e[r]), \sigma(G))$.

The extensive discussion of the previous pages has shown the various possibilities for choosing evaluation strategies and their impacts on the semantics.

(4) Defining the interface functions $tell_{\mathcal{FL}}$ *and* $proj_{\mathcal{FL}\rightarrow\nu}$ *of the particular language solver*. This final step has to integrate the effects of narrowing and constraint collection with the other solvers.

Propagation. Figure 5 illustrates how the interface function $tell_{\mathcal{FL}}$ is used to simulate a narrowing step with constraints, and Figure 6 gives the formal definition of the pertinent requirements. To ease the presentation we suppose that all functional logic equalities are in the form $Y =_{\mathcal{FL}} t$ with variable Y and term t. This can always be achieved with the help of auxiliary variables.

Like all solvers, $CS_{\mathcal{FL}}$ propagates constraints to its store $C_{\mathcal{FL}}$ thereby checking the satisfiability of $C_{\mathcal{FL}}$ in conjunction with the new constraints. Therefore the function $tell_{\mathcal{FL}}$ incorporates the principle of narrowing.

In the light of the preceding discussion on flattening we assume that all constraints which get into the pool (either as part of the initial problem to solve or as results of propagations or projections) are decomposed with the help of auxiliary variables such that all subterms of the form $f(t_1, \ldots, t_n)$, where f is a defined function, are extracted. The definition of $tell_{\mathcal{FL}}$ therefore only needs to consider narrowing steps on outermost terms. The distinction between *call-by-value* and *call-by-name* must therefore be based on the aforementioned dependency relation among variables.

tell$_{\mathcal{FL}}$: Let P be a functional logic program with constraints, let $C_{\mathcal{FL}} = \phi$ be the current constraint store of $CS_{\mathcal{FL}}$. (Recall that the constraint store $C_{\mathcal{FL}}$ is nothing but a substitution ϕ from variables to constructor terms, which is written in the form of equations and thus can be treated like constraints.) Let $c = (Y =_{\mathcal{FL}} t)$ be the constraint to be propagated. Let $\hat{t} = \phi(t)$.

Finally, we use the following notion: A rule $p = (l_p \to r_p$ **where** $G_p)$ *applies to* $\hat{t}$, if for $\hat{t} \notin X$ there is a unifier $\sigma_p = mgu(\hat{t}, l_p)$.

We need to distinguish the following cases

1. Let $\hat{t}$ contain defined functions; that is, $\hat{t}$ is of the form $f(\ldots)$ with f being the only defined function (due to the maximal flattening). If the set $P_c \subseteq P$ of applicable rules is nonempty, then
   $tell_{\mathcal{FL}}(c, C_{\mathcal{FL}}) = (true,\ C_{\mathcal{FL}},\ \bigvee_{p \in P_c}(\sigma_p \wedge (Y =_{\mathcal{FL}} \sigma_p(r_p)) \wedge \sigma_p(G_p)))$.
2. Let $\hat{t}$ be a constructor term, i. e. $\hat{t} \in \mathcal{T}(\Delta, X_{\mathcal{FL}})$.

(a) If $(\{Y = t\} \uparrow C_{\mathcal{FL}}) \neq \emptyset$, then
$tell_{\mathcal{FL}}(c, C_{\mathcal{FL}}) = (true,\ \{Y = t\} \uparrow C_{\mathcal{FL}}\ , true)$.

(b) If $(\{Y = t\} \uparrow C_{\mathcal{FL}}) = \emptyset$, then
$tell_{\mathcal{FL}}(c, C_{\mathcal{FL}}) = (false, C_{\mathcal{FL}}, false)$.

Fig. 6. Interface function *tell*$_{\mathcal{FL}}$.

We have to distinguish two kinds of constraints $Y =_{\mathcal{FL}} t$ (see Figure 6):

(1) When the term t still contains defined functions, a narrowing step is applied as part of *tell*$_{\mathcal{FL}}$. This is only reflected by a change of the constraint pool. The store does not change in this case. Note that due to the flattening t contains exactly one defined function f, and this function is the outermost symbol in t. Moreover, since the substitution ϕ defined in Figure 6 only contains constructor terms, $\hat{t} = \phi(t)$ retains this property.

(2) When the term t is a constructor term then the constraint $Y =_{\mathcal{FL}} t$ is added to the store if possible. Thereby, the satisfiability test is realised by the parallel composition $\uparrow$ of substitutions.

This integration of the narrowing into the propagation sometimes leads to earlier bindings of variables and thus to a faster recognition of unsuccessful computations. The price to be paid is that the solver $CS_{\mathcal{FL}}$ is amalgamated into the function *tell*$_{\mathcal{FL}}$ – and with it all its problems.

Example 4.2. To elucidate the interface definition we use our running example from Sect. 3.1. In Step (2) of this example we applied the only matching rule

$$\texttt{rc(par(R1,R2))} \to \texttt{Z}\ \textbf{where}\ 1/\texttt{X} + 1/\texttt{Y} =_{\mathcal{A}} 1/\texttt{Z}, \quad \texttt{X} =_{\mathcal{FL}} \texttt{rc(R1)}, \quad \texttt{Y} =_{\mathcal{FL}} \texttt{rc(R2)}$$

to the initial configuration. We split the equation in the pool into two equations using an auxiliary variable such that the pool reads:

$$\texttt{Z}_1 =_{\mathcal{FL}} \texttt{rc(par(RA,RB))}, \quad \texttt{Z}_1 =_{\mathcal{FL}} 200$$

We pick the first term and apply *tell*$_{\mathcal{FL}}$ as described in Figure 6. Note that we have the special situation, where the store $C_{\mathcal{FL}} = \phi$ is still empty such that $\hat{t} = \phi(\texttt{rc(par(RA,RB))}) = \texttt{rc(par(RA,RB))}$.

https://doi.org/10.1017/S1471068406002833 Published online by Cambridge University Press

$proj_{\mathcal{FL}\to\nu}$: The projection of a store $C_{\mathcal{FL}} = \phi$ w.r.t. a constraint system ζ_ν and a set of variables $X \subseteq X_{\mathcal{FL}} \cap X_\nu$ makes the substitutions for $\mathtt{x} \in X$ explicit: $proj_{\mathcal{FL}\to\nu}(X,\phi) = \begin{cases} \phi\vert_X & \text{if } \phi \neq \emptyset \\ true & \text{otherwise.} \end{cases}$

Fig. 7. Interface function $proj_{\mathscr{F}\mathscr{L}\to\nu}, \nu \in L$.

The narrowing step for the term with (a new instance of) the rule unifies the subterm $\mathtt{rc(par(RA,RB))}$ – which happens to be the full term – with the left-hand side $\mathtt{rc(par(R1,R2))}$ of the rule. Since this matching exists, we have Case 1 of Figure 6. The resulting most general unifier is the substitution $\sigma = \{\mathtt{R1} = \mathtt{RA}, \mathtt{R2} = \mathtt{RB}\}$. Applied to the rule this substitution yields the instantiated right-hand side $\mathtt{Z}$ and constraints $1/\mathtt{X} + 1/\mathtt{Y} =_{\mathscr{A}} 1/\mathtt{Z}$, $\mathtt{X} =_{\mathscr{F}\mathscr{L}} \mathtt{rc(RA)}, \mathtt{Y} =_{\mathscr{F}\mathscr{L}} \mathtt{rc(RB)}$. This is put back into the pool:

$$\mathtt{R1} =_{\mathscr{F}\mathscr{L}} \mathtt{RA},\ \mathtt{R2} =_{\mathscr{F}\mathscr{L}} \mathtt{RB},\ \mathtt{Z}_1 =_{\mathscr{F}\mathscr{L}} \mathtt{Z},\ 1/\mathtt{X} + 1/\mathtt{Y} =_{\mathscr{A}} 1/\mathtt{Z},\ \mathtt{X} =_{\mathscr{F}\mathscr{L}} \mathtt{rc(RA)},$$
$$\mathtt{Y} =_{\mathscr{F}\mathscr{L}} \mathtt{rc(RB)},\ \mathtt{Z}_1 =_{\mathscr{F}\mathscr{L}} 200$$

If there is more than one applicable rule, we get a number of newly built constraint pools.

Applying $tell_{\mathscr{F}\mathscr{L}}$ to the two constraints $\mathtt{R1} =_{\mathscr{F}\mathscr{L}} \mathtt{RA}$ and $\mathtt{R2} =_{\mathscr{F}\mathscr{L}} \mathtt{RB}$ leads to Case 2.(a) of Figure 6. As a result the store $C_{\mathscr{F}\mathscr{L}}$ then contains these constraints (a substitution). The remaining constraints in the pool are as follows:

$$\mathtt{Z}_1 =_{\mathscr{F}\mathscr{L}} \mathtt{Z},\ \mathtt{Z}_1 =_{\mathscr{F}\mathscr{L}} 200,\ 1/\mathtt{X} + 1/\mathtt{Y} =_{\mathscr{A}} 1/\mathtt{Z},\ \mathtt{X} =_{\mathscr{F}\mathscr{L}} \mathtt{rc(RA)},\ \mathtt{Y} =_{\mathscr{F}\mathscr{L}} \mathtt{rc(RB)}.$$

Note, that Example 3.1 displays simplified forms for brevity. ◁

Projection. Since the constraint store $C_{\mathscr{F}\mathscr{L}}$ only contains substitutions, the projection is trivial. It generates equality constraints representing them. For example, in Step (7) the constraint $\mathtt{Z} =_{\mathscr{A}} 200$ is transferred from $C_{\mathscr{F}\mathscr{L}}$ to the pool using $proj_{\mathscr{F}\mathscr{L}\to\mathscr{A}}$.

The specification of the function $proj_{\mathscr{F}\mathscr{L}\to\nu}$ is given in Figure 7. (Note, that the equalities in the result $\phi|_X$ are indexed ν to express the fact that they now belong to the domain of the solver CS_ν.)

Completeness and other aspects. Since the interface functions $tell_{\mathscr{F}\mathscr{L}}$ and $proj_{\mathscr{F}\mathscr{L}\to\nu}$ of our functional logic language solver fulfil the requirements given in Sect. 3, the soundness and completeness results of the cooperation framework hold for the integration of a functional logic language – however, only relative to the completeness of the narrowing strategy encoded within $tell_{\mathscr{F}\mathscr{L}}$.

It is well known that narrowing is in principle a complete and sound method for theories presented by confluent and terminating rewrite systems. But in connection with functional logic languages and in particular in the presence of extra variables the issue of completeness becomes more intricate. This problem has been investigated by several authors, among others (Middeldorp and Hamoen 1994; Suzuki *et al.* 1995; Hanus 1995). These papers discuss slightly different syntactic criteria, which guarantee that certain narrowing strategies (e.g. lazy or needed narrowing) are sound and complete.

Evidently, these criteria carry over to the narrowing strategy that we embed into our function $tell_{\mathscr{F}\mathscr{L}}$. For lack of space we cannot go into details here. Suffice it to say that, for example, our program rule

$$\texttt{rc(par(R1,R2))} \rightarrow \texttt{Z} \textbf{ where } 1/\texttt{X} + 1/\texttt{Y} =_{\mathscr{A}} 1/\texttt{Z}, \quad \texttt{X} =_{\mathscr{FL}} \texttt{rc(R1)}, \quad \texttt{Y} =_{\mathscr{FL}} \texttt{rc(R2)}$$

contains the three extra variables `X`, `Y` and `Z` and is in the class 3-CTRS of (Middeldorp and Hamoen 1994). But the syntactic form also meets the requirements of being *orthogonal*, *properly oriented* and *right-stable* (Suzuki *et al.* 1995), which guarantees that the program is level-confluent. (These conditions essentially state that the variables are in a nice left-to-right order.)

As a matter of fact, the rule is also *constructor-based* and *functional* in the sense of (Hanus 1995), since the variables `R1` and `R2` in the guarding constraints occur in the left-hand side and the variable `Z` of the right-hand side is defined by the guarding constraints. This also entails the completeness of lazy and needed narrowing. But Hanus imposes a further requirement: the equality $=_{\mathscr{FL}}$ has to be strict (which allows to consider it as an equality $(Y \equiv t) =_{\mathscr{FL}} true$). Since we only put substitutions with constructor terms into the store $C_{\mathscr{FL}}$, this works in our approach as well. (This is similar to the language CURRY (Hanus *et al.* 2003).)

However, there is one further issue! The above quoted requirements refer to the functional logic equalities in the guard G. But our rules have a more general form:

$$f(t_1,\ldots,t_n) \rightarrow r \textbf{ where } c_1,\ldots,c_m, u_1 =_{\mathscr{FL}} v_1,\ldots,u_k =_{\mathscr{FL}} v_k$$

Only the equalities $u_i =_{\mathscr{FL}} v_i$ (with u_i being a constructor term, possibly only a variable) belong to the functional logic solver. The c_j are constraints from other domains.

This raises the question: How do these constraints $c_1,\ldots,c_m$ from the other domains fit into the picture? Let us consider our standard example

$$\texttt{rc(par(R1,R2))} \rightarrow \texttt{Z} \textbf{ where } 1/\texttt{X} + 1/\texttt{Y} =_{\mathscr{A}} 1/\texttt{Z}, \quad \texttt{X} =_{\mathscr{FL}} \texttt{rc(R1)}, \quad \texttt{Y} =_{\mathscr{FL}} \texttt{rc(R2)}$$

Without the first constraint, the variable `Z` would not occur in the guard and thus the rule would not even be 3-CTRS (and thus no completeness results could be given at all). On the other hand, all constraints c_i do not contain any of the defined functions and therefore do not participate in any kind of narrowing strategy. So, from the point of view of narrowing, a variable like `Z` can be treated like a constant.

If one of the other solvers finds a solution, e. g. $\texttt{Z} = 200$ in our example, then the corresponding substitution is put into the store $C_{\mathscr{FL}}$. This way the associated value is guaranteed to be a ground constructor term and thus does not interfere with the narrowing strategies.

This illustrates again the – evident – fact that the completeness of the overall system, and thus also of its narrowing part, depends on the completeness of all participating solvers.

4.2 A logic language as constraint solver

The integration of a logic language into the system of cooperating solvers yields a constraint logic programming language. Since this is actually simpler than the case of functional logic languages, we only sketch it here briefly.

tell$_{\mathscr{L}}$: Let P be a constraint logic program, let $C_{\mathscr{L}} = \phi$ be the current store of $CS_{\mathscr{L}}$.

1. Let $R = p(t_1, \ldots, t_m)$ be the constraint (goal) which is to be propagated. Let $\hat{R} = \phi(R)$. We use the following notion: A rule $p = (Q_p \;\texttt{:-}\; rhs_p)$ *applies to* $\hat{R}$, if there is a unifier $\sigma_p = mgu(\hat{R}, Q_p)$.

(a) If the set $P_R \subseteq P$ of applicable rules is nonempty, then
$tell_{\mathscr{L}}(R, C_{\mathscr{L}}) = (true,\ C_{\mathscr{L}},\ \bigvee_{p \in P_R}(\sigma_p \wedge \sigma_p(rhs_p)))$.

(b) If there is no applicable rule in P, then
$tell_{\mathscr{L}}(R, C_{\mathscr{L}}) = (false, C_{\mathscr{L}}, false)$.

2. Let $c = (Y =_{\mathscr{L}} t)$ be the constraint which is to be propagated.

(a) If $(\{Y = t\} \uparrow C_{\mathscr{L}}) \neq \emptyset$, then
$tell_{\mathscr{L}}(c, C_{\mathscr{L}}) = (true,\ \{Y = t\} \uparrow C_{\mathscr{L}}\ , true)$.

(b) If $(\{Y = t\} \uparrow C_{\mathscr{L}}) = \emptyset$, then
$tell_{\mathscr{L}}(c, C_{\mathscr{L}}) = (false, C_{\mathscr{L}}, false)$.

Fig. 8. Interface function *tell*$_{\mathscr{L}}$.

(1 & 2) Identifying language constraints and extending the language by constraints of other domains. It is widely accepted that logic programming can be interpreted as constraint programming over the Herbrand universe. The appropriate constraint solving mechanism $CS_{\mathscr{L}}$ is resolution.

The goals according to a given constraint logic program P are the natural constraints of a logic language solver. Furthermore, the set $\mathscr{C}ons_{\mathscr{L}}$ of constraints of this solver must contain equality constraints $Y =_{\mathscr{L}} t$ between variables and terms to represent substitutions.[6]

We extend the syntax of the language by constraints of other constraint systems which yields the typical CLP syntax, cf. e.g. (Jaffar and Lassez 1987). Thus, the set $\mathscr{C}ons_{\mathscr{L}}$ must furthermore include all constraints of the incorporated solver(s).

(3 & 4) Extending the language evaluation mechanism by gathering constraints and defining the interface functions of the particular language solver. For the integration of $CS_{\mathscr{L}}$ into the system the interface functions *tell*$_{\mathscr{L}}$ and *proj*$_{\mathscr{L}\to v}$ must be defined. Step (3), i.e. gathering constraints during resolution, is realised by the extension of the resolution step from atoms to the whole body including the constraints of other domains.

Propagation. The propagation function *tell*$_{\mathscr{L}}$ emulates resolution steps (including gathering constraints). Its formal definition is given in Figure 8. Case 1.(a) represents a resolution step on a goal R as a successful propagation, where for every applicable rule we get a newly created constraint pool and, thus, a new instantiation of the architecture. If there is no applicable rule for a goal, i.e. Case 1.(b), the propagation fails (in contrast to the functional logic solver considered before, undefinedness of a predicate is regarded as failure here).

Similar to the definition of *tell*$_{\mathscr{F}\mathscr{L}}$ the remaining cases describe the propagation of equality constraints by parallel composition of substitutions (Case 2).

[6] Not all CLP systems support equalities on the syntactical level. Rather they only generate them internally in the solver. In our system, they are explicitly visible.

Projection. As before, the projection function provides constraints representing the substitution from the store which has been computed during resolution. The definition of the projection function $proj_{\mathscr{L}\to v}$ is the same as for the functional logic language solver given in Figure 7, where the index $\mathscr{F}\mathscr{L}$ is replaced by $\mathscr{L}$ to denote the origin of the projection.

Since the interface functions $tell_{\mathscr{L}}$ and $proj_{\mathscr{L}\to v}$ fulfil the requirements given in Sect. 3, the soundness and completeness results of the cooperation framework hold for the integration of a logic language.

5 Conclusion and related work

This paper describes a general approach for the integration of declarative languages and constraint systems. This essentially means to treat their evaluation mechanisms together with programs as constraint solvers. This is done in *four general steps*:

1. Identifying language constraints,
2. Extending the language by constraints of other domains,
3. Extending the language evaluation mechanism by gathering constraints,
4. Defining the interface functions of the particular language solver.

The most important aspect of our approach is that the overall system for cooperating solvers allows the handling of hybrid constraints over different domains.

Gains and perspectives. Our general framework for cooperating solvers provides mechanisms for the definition of cooperation strategies. Similar to CLP(X) (Jaffar and Lassez 1987) and CFLP(X) (López-Fraguas 1992), which are covered by our approach, the framework can thus be instantiated by three parameters: a strategy definition $\mathscr{S}$, a set $\mathscr{X}$ of constraint systems and a host language $\mathscr{Y}$. In this way, our approach enables the *building of constraint languages customised for a given set of requirements for a comfortable modelling and the solution of many problems.*

For example, if one needs a convenient language to express search problems with special constraints over finite and/or further domains, the user chooses a logic language and according constraint systems. For problems which allow or require a more deterministic modelling one may decide to build a constraint language based on a functional or functional logic language. In a similar way one can imagine to combine a database language with constraints (Kanellakis *et al.* 1995) or a particular special-purpose language or system, such as an expert system, a geographical information system, or a planning system. Of course, the user is responsible for a sound specification of the language constraints and the definition of the interface functions.

The meta-solver system META-S (Frank *et al.* 2003b; Frank *et al.* 2003a) implements our ideas. Even though META-S provides programming constructs for its integration into other applications, which may even be imperative languages, the true integration of imperative languages according to our approach is an open question and a topic of future research. The main reason is that declarative languages abstract from real-world issues such as time and state while imperative languages are time-dependent which complicates their integration also for our approach.

The choice of an appropriate cooperation strategy plays an important role for the efficiency of the cooperating system (Frank *et al.* 2003a).

The definition of solver cooperation strategies is also very interesting in the case of language solvers. This will allow for example to switch from depth-first search to breadth-first search or an evaluation mechanism based on the Andorra principle (Costa *et al.* 1991; Warren 1988) using a logic language without reimplementing the evaluation mechanisms. The system META-S already offers predefined strategy patterns for these search strategies which can be refined by problem-dependent knowledge or user knowledge about the program.

If the user is able to define cooperation strategies for the solvers and the language(s) or to refine them on the base of predefined strategy patterns (as given in our implementation), she/he can also employ problem-dependent knowledge and user knowledge, e. g. about the termination of particular predicates, to guide the computation.

First results (Frank *et al.* 2004) on the integration of language solvers into the meta-solver framework META-S according to the described approach confirm our theoretical considerations.

Finally, the approach opens a further interesting perspective: A simple approach for the combination of different languages consists of the definition of an explicit interface and providing language constructs for initialising subcomputations in the particular languages. In this way one reaches a loose coupling and interaction of programs written in different languages.

Our cooperation framework bases on a similar idea: It provides a meta mechanism which takes care of the strategy of the cooperating solvers and provides the constraint pool for maintaining and managing common data and constraints. Besides this the main concept is the uniform solver interface which allows to integrate declarative languages as solvers as shown above. Using this interface it is not only possible to integrate constraint solvers and language evaluation mechanisms but also to integrate language evaluation mechanisms among each other by appropriate interface definitions. This finally yields a language interaction according to the above sketched interface model.

Related Work. (López-Fraguas 1992) considered a general scheme CFLP(X) for constraint functional logic programming. The scheme is based on lazy functional logic languages and allows beside conditions constraints in the guards of the rules. Using our cooperation approach, we achieve a covering of the CFLP(X) and CLP(X) (Jaffar and Lassez 1987) approaches. In (Hortalá-González *et al.* 1997) and (Fernández *et al.* 2003), extending CFLP(X), functional logic programming is integrated with real arithmetic and finite domain constraints respectively. The lazy functional logic languages TOY($\mathcal{R}$) and TOY($\mathcal{FD}$) are the respective implementations. (Lux 2001) integrates linear constraints over real numbers into the functional logic language CURRY in a similar way.

OPEN CFLP by (Kobayashi *et al.* 2003) combines a functional logic host language with collaborating equational solvers which may be distributed in an open environment. It provides the user with a declarative strategy definition for the cooperating solvers basing upon a set of basic operators. However, the strategy language of META-S gives finer control over the individual collaboration steps because of its well considered solver interface on the one hand and its structural pattern-matching and constraint rewriting facilities which provide a finer and more intuitive control for strategy definition on the other hand.

While our approach pursues the idea to integrate languages into a system of cooperating solvers the approaches (López-Fraguas 1992; Hortalá-González *et al.* 1997; Fernández *et al.* 2003; Kobayashi *et al.* 2003; Lux 2001) come from the opposite point of view and extend the functional logic program evaluation by constraint evaluation.

In contrast to the other approaches our framework allows the integration of several constraint systems. The user can integrate desired domains and solvers which satisfy the interface requirements as discussed in Sect. 3. The opportunity to integrate different host languages and constraint systems also distinguishes our approach from other existing systems of cooperating solvers (for example (Hong 1994; Monfroy 1996; Rueher 1995)) that usually have one fixed host language (a logic language).

Furthermore both, the above mentioned languages and the cooperative systems, mainly have a fixed order of evaluation of constraints and functional expressions resp. In contrast, an integration according to our ideas using the META-S system allows the user to either define its own strategies or to refine existing strategy patterns in a simple way. The usefulness of different cooperation strategies has been proven for usual solvers (e. g. on arithmetic, see (Frank *et al.* 2003b)). As well, first results for the integration of a logic language into META-S confirm their usefulness for language solvers (Frank *et al.* 2004).

OZ (Müller *et al.* 1995) supports (constraint) logic, functional and object-oriented programming styles within one (as well fixed) language. The computation in OZ is based on the concept of computation spaces (Schulte 2002) which consist of a constraint store containing only basic constraints and propagators (for more complex constraints) manipulating them. Similar to our framework, computation spaces can be used to describe solver cooperations and search strategies. However, this relies on all solvers sharing the same store format and hence is not satisfying for the main goal of our approach, i. e. the cooperation of black box solvers independent of their implementation.

Acknowledgments

We want to thank our colleague Stephan Frank for many valuable comments and his assistance in preparing the manuscript. Our particular thanks goes to the reviewers who provided detailed and insightful comments that helped to improve the paper considerably.

References

AÏT-KACI, H. AND NASR, R. 1989. Integrating Logic and Functional Programming. *Lisp and Symbolic Computation 2,* 1, 51–89.

Broy, M., Wirsing, M., and Pepper, P. 1987. On the Algebraic Definition of Programming Languages. *ACM Transactions on Programming Languages and Systems 9,* 1 (January), 54–99.

Cheadle, A. M., Harvey, W., Sadler, A. J., Schimpf, J., Shen, K., and Wallace, M. G. 2003. ECL^{i}PS^e . An Introduction. Tech. Rep. IC-PARC-03-1, Centre for Planning and Resource Control, Imperial College London.

Costa, V. S., Warren, D. H., and Yang, R. 1991. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming – PPOPP*. SIGPLAN Notices, vol. 26 (7). ACM Press, 83–93.

Didrich, K., Fett, A., Gerke, C., Grieskamp, W., and Pepper, P. 1994. OPAL: Design and Implementation of an Algebraic Programming Language. In *International Conference on Programming Languages and System Architectures*, J. Gutknecht, Ed. LNCS, vol. 782. Springer, 228–244.

Fernández, A., Hortalá-Gonzáles, T., and Sáenz-Pérez, F. 2003. Solving Combinatorial Problems with a Constraint Functional Logic Language. In *Practical Aspects of Declarative Languages, 5th International Symposium – PADL*, V. Dahl and P. Wadler, Eds. LNCS, vol. 2562. Springer, 320–338.

Field, A. and Harrison, P. 1988. *Functional Programming*. Addison-Wesley.

Frank, S., Hofstedt, P., and Mai, P. R. 2003a. A Flexible Meta-solver Framework for Constraint Solver Collaboration. In *Advances in Artificial Intelligence, 26th German Conference on Artificial Intelligence – KI 2003*, A. Günter, R. Kruse, and B. Neumann, Eds. LNCS, vol. 2821. Springer, Kiel, Germany, 520–534.

Frank, S., Hofstedt, P., and Mai, P. R. 2003b. Meta-S: A Strategy-oriented Meta-Solver Framework. In *16th International Florida Artificial Intelligence Research Society Conference – FLAIRS*, I. Russell and S. M. Haller, Eds. The AAAI Press, St. Augustine, Florida, 177–181.

Frank, S., Hofstedt, P., and Reckmann, D. 2004. Strategies for the Efficient Solution of Hybrid Constraint Logic Programs. In *Third Workshop on Multiparadigm Constraint Programming Languages – MultiCPL*, S. Muñoz-Hernández, J. M. Gómez-Perez, and P. Hofstedt, Eds. Saint-Malo, France, 103–117.

Hanus, M. 1994. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming 19/20*, 583–628.

Hanus, M. 1995. On Extra Variables in (Equational) Logic Programming. In *Twelfth International Conference on Logic Programming*. MIT Press, 665–679.

Hanus, M., Antoy, S., Kuchen, H., López-Fraguas, F. J., Lux, W., Moreno-Navarro, J. J., and Steiner, F. 2003. Curry: An Integrated Functional Logic Language. Report. Version 0.8 of April 15, 2003.

Hofstedt, P. 2000a. Better Communication for Tighter Cooperation. In *First International Conference on Computational Logic – CL*. LNCS, vol. 1861. Springer, 342–358.

Hofstedt, P. 2000b. Cooperating Constraint Solvers. In *Sixth International Conference on Principles and Practice of Constraint Programming – CP*. LNCS, vol. 1894. Springer, 520–524.

Hofstedt, P. 2001. Cooperation and Coordination of Constraint Solvers. Ph.D. thesis, Dresden University of Technology.

Hong, H. 1994. Confluency of Cooperative Constraint Solvers. Tech. Rep. 94–08, Research Institute for Symbolic Computation, Linz, Austria.

Hortalá-González, T., López-Fraguas, F., Sánchez-Hernández, J., and Ullán-Hernández, E. 1997. Declarative Programming with real Constraints. Tech. Rep. SIP 5997, Universidad Complutense de Madrid.

Hudak, P., Peterson, J., and Fasel, J. H. 2000. A Gentle Introduction to Haskell 98. Tech. rep., Yale University, Department of Computer Science. `http://www.haskell.org/tutorial/`, visited 2005-11-18.

Jaffar, J. and Lassez, J.-L. 1987. Constraint Logic Programming. In *14th ACM Symposium on Principles of Programming Languages – POPL*. ACM Press, 111–119.

Jaffar, J., Maher, M., Marriott, K., and Stuckey, P. 1998. The Semantics of Constraint Logic Programs. *Journal of Logic Programming 37,* 1–3, 1–46.

Kanellakis, P., Kuper, G., and Revesz, P. 1995. Constraint Query Languages. *Journal of Computer and System Sciences 51,* 1, 26–52.

Kobayashi, N., Marin, M., and Ida, T. 2003. Collaborative Constraint Functional Logic Programming System in an Open Environment. *IEICE Transactions on Information and Systems E86-D,* 1, 63–70.

Loogen, R. 1995. *Integration funktionaler und logischer Programmiersprachen*. Oldenbourg.

López-Fraguas, F.-J. 1992. A General Scheme for Constraint Functional Logic Programming. In *Algebraic and Logic Programming – ALP*, H. Kirchner and G. Levi, Eds. LNCS, vol. 632. Springer, 213–227.

Lux, W. 2001. Adding Linear Constraints over Real Numbers to Curry. In *Functional and Logic Programming, 5th International Symposium – FLOPS*, H. Kuchen and K. Ueda, Eds. LNCS, vol. 2024. Springer, 185–200.

Manna, Z. 1974. *Mathematical Theory of Computation*. McGraw-Hill.

Middeldorp, A. and Hamoen, E. 1994. Completeness Results for Basic Narrowing. *Applicable Algebra in Engineering, Communication and Computing* 5, 213–253.

Monfroy, E. 1996. Solver Collaboration for Constraint Logic Programming. Ph.D. thesis, Centre de Recherche en Informatique de Nancy. INRIA.

Moreno-Navarro, J. and Rodríguez-Artalejo, M. 1992. Logic Programming with Functions and Predicates: The Language BABEL. *Journal for Logic Programming 12,* 3&4, 191–223.

Müller, M., Müller, T., and Van Roy, P. 1995. Multiparadigm Programming in Oz. In *Workshop on Visions for the Future of Logic Programming*, D. Smith, O. Ridoux, and P. V. Roy, Eds. Portland, Oregon.

Nilsson, U. and Małuszyński, J. 1995. *Logic, Programming and Prolog*. John Wiley & Sons Ltd.

Palamidessi, C. 1990. Algebraic Properties of Idempotent Substitutions. In *Automata, Languages and Programming – ICALP*, M. Paterson, Ed. LNCS, vol. 443. Springer, 386–399.

Reddy, U. 1985. Narrowing as the Operational Semantics of Functional Languages. In *IEEE Symposium on Logic Programming*. 138–151.

Rueher, M. 1995. An Architecture for Cooperating Constraint Solvers on Reals. In *Constraint Programming: Basics and Trends*, A. Podelski, Ed. LNCS, vol. 910. Springer, 231–250.

Schulte, C. 2002. *Programming Constraint Services*. LNCS, vol. 2302. Springer.

Suzuki, T., Middeldorp, A., and Ida, T. 1995. Level-Confluence of Conditional Rewrite Systems with Extra Variables in Right-Hand Sides. In *6th International Conference on Rewriting Techniques and Applications*. LNCS, vol. 914. Springer, 179–193.

Warren, D. H. 1988. The Andorra Principle. Presented at the Gigalips Workshop, Swedish Institute of Computer Science (SICS), Stockholm, Sweden.

Westfold, S. J. and Smith, D. R. 2001. Synthesis of Efficient Constraint Satisfaction Programs. In *Knowledge Engineering Reviews. Special Issue on AI and OR*. Kestrel Institute Technical Report KES.U.01.7.

Winskel, G. 1993. *Formal Semantics of Programming Languages*. MIT Press.