 Electronic Notes in Theoretical Computer Science 177 (2007) 235–252 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

A Fully Sound Goal Solving Calculus for the Cooperation of Solvers in the *CFLP* Scheme

S. Est´evez Mart´ın[a](#_bookmark0)*,*[1](#_bookmark2) A. J. Fern´andez[b](#_bookmark1)*,*[2](#_bookmark2)

M.T. Hortala´ Gonz´alez[a](#_bookmark0)*,*[1](#_bookmark2) M. Rodr´ıguez Artalejo[a](#_bookmark0)*,*[1](#_bookmark2)

R. del Vado V´ırseda[a](#_bookmark0)*,*[1](#_bookmark2)

a *Departamento de Sistemas Informa´ticos y Computacio´n Universidad Complutense de Madrid*

b *Departamento de Lenguajes y Ciencias de la Computacio´n Universidad de M´alaga*

Abstract

The *CFLP* scheme for Constraint Functional Logic Programming has instances *CFLP* (D) corresponding to different constraint domains D. In this paper, we propose an amalgamated sum construction for building coordination domains C, suitable to represent the cooperation among several constraint domains D1*,. ..,* D*n* via a mediatorial domain M. Moreover, we present a cooperative goal solving calculus for *CFLP* (C), based on lazy narrowing, invocation of solvers for the different domains D*i* involved in the coordination domain C, and projection operations for converting D*i* constraints into D*j* constraints with the aid of mediatorial

constraints (so-called bridges) supplied by M. Under natural correctness assumptions for the projection operations, the cooperative goal solving calculus can be proved fully sound w.r.t. the declarative semantics of *CFLP* (C). As a relevant concrete instance of our proposal, we consider the cooperation between Herbrand, real arithmetic and finite domain constraints.

*Keywords:* Cooperative Goal Solving, Constraints, Functional-Logic Programming, Lazy Narrowing.

# Introduction

The scheme *CFLP* for Constraint Functional Logic Programming, recently pro- posed in [[11](#_bookmark30)], continues a long history of attempts to combine the expressive power of functional and logic programming with the improvements in performance pro- vided by domain specific constraint solvers. As the well-known *CLP* scheme [[9](#_bookmark28)], *CFLP* has many possible instances *CFLP* (D) corresponding to different specific constraint domains D given as parameters. In spite of the generality of the approach,

1 Author partially supported by projects TIN2005-09207-C03-03 and S-0505/TIC0407.

2 Author partially supported by projects TIN2004-7943-C04-01 and TIN2005-08818-C04-01.

1571-0661 © 2007 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2007.01.026

the use of one fixed domain D is an important limitation, since many practical pro- blems involve more than one domain.

A solution to this practical problem in the *CLP* context can be found in the concept of solver cooperation [[5](#_bookmark24)], an issue that is raising an increasing interest in the cons- traint community. In general, solver cooperation aims at overcoming two problems: a lack of declarativity of the solutions (i.e., the interaction among solvers makes it easier to express compound problems) and a poor performance of the systems (i.e., the communication among solvers can improve the efficiency of the solving process). This paper presents a proposal for coordinated programming in the *CFLP* scheme as described in [[11](#_bookmark30)]. We introduce *coordination domains* as amalgamated sums of the various domains to be coordinated, along with a *mediatorial domain* which supplies special communication constraints, called *bridges*, used to impose equivalences among values of different base types. Building upon previous works [[2](#_bookmark20),[10](#_bookmark29),[15](#_bookmark34)], we also describe a coordinated goal solving calculus which combines lazy narrowing with the invocation of the cooperating solvers and two kinds of communi- cation operations, namely the creation of bridges and the projection of constraints between different constraint stores. Projection operations are guided by existing bridges. Using the declarative semantics of *CFLP* , we have proved a semantic re- sult called *full soundness*, ensuring soundness and local completeness of the goal

solving calculus.

In order to place our proposal for solver cooperation in context, we briefly dis- cuss main differences and similarities with a limited selection of related proposals existing in the literature. E. Monfroy [[14](#_bookmark33)] proposed the system BALI (Binding Archi- tecture for Solver Integration) that facilitates the specification of solver cooperation as well as integration of heterogeneous solvers via a number of cooperations primi- tives. Monfroy’s approach assumes that all the solvers work over a common store, while our present proposal requires communication among different stores. Also, Mircea Marin [[12](#_bookmark31)] developed a *CFLP* scheme that combines Monfroy’s approach to solver cooperation with a higher-order lazy narrowing calculus somewhat simi- lar to [[10](#_bookmark29),[15](#_bookmark34)] and the goal solving calculus presented in this paper. In contrast to our proposal, Marin’s approach allows for higher-order unification, which leads both to greater expressivity and to less efficient implementations. Moreover, the instance of *CFLP* implemented by Marin and others [[13](#_bookmark32)] combines four solvers over a constraint domain for algebraic symbolic computation, while the instance we are currently implementing deals with the cooperation among Herbrand, finite domain and real arithmetic constraints. Recently, P. Hofstedt [[7](#_bookmark26),[8](#_bookmark27)] proposed a general approach for the combination of various constraint systems and declarative languages into an integrated system of cooperating solvers. In Hofstedt’s proposal, the goal solving procedure of a declarative language is viewed also as a solver, and cooperation of solvers is achieved by two mechanisms: constraint propagation, that submits a constraint belonging to some domain D to its constraint store, say *S*D; and projection of constraint stores, that consults the contents of a given store *S*D and deduces constraints for another domain. Projection, as used in this paper,

differs from Hofstedt’s projection in the creation and use of bridges; while Hofs- tedt’s propagation corresponds to our goal solving rules for placing constraints in stores and invoking constraint solvers. Hofstedt also proposes the construction of combined computation domains, similar to our coordination domains. The lack of bridges in Hofstedt’s approach corresponds to the lack of mediatorial domains within her combined domains. In different places along the paper we will include comparisons to Hofstedt’s approach; see especially Table 5 in Section [5](#_bookmark18).

The structure of the paper is as follows: Section [2](#_bookmark3) introduces the basic notions of constraint domains and solvers underlying the *CFLP* scheme. Section [3](#_bookmark7) describes the constructions needed for coordination in our setting, namely coordination do- mains, bridges and projections. Programs, goals, the lazy narrowing calculus for cooperative goal solving (with a typical example), and the full soundness result are described in Section [4](#_bookmark9). Section [5](#_bookmark18) summarizes conclusions and future work.

# Constraint Domains and Solvers in the *CFLP* Scheme

In this section, we recall the essentials of the *CFLP* (D) scheme [[11](#_bookmark30)], which serves as a logical and semantic framework for lazy Constraint Functional Logic Programming (briefly *CFLP* ) over a parametrically given constraint domain D. The proper choice of D for modeling the coordination of several constraint domains will be discussed in Section [3](#_bookmark7). As a main novelty w.r.t. [[11](#_bookmark30)], the current presentation of *CFLP* (D) includes now an explicit treatment of a Milner-like polymorphic type system in the line of previous work in Functional Logic Programming [[4](#_bookmark23)].

* 1. *Signatures and Constraint Domains*

We assume a *universal signature* Σ = ⟨*T C, DC, DF* ⟩, where *TC* =

*TCn*,

*DC* =

*n*∈N *DCn* and *DF* =

*n*∈N

*n*∈N *DFn* are families of countably infinite and

mutually disjoint sets of *type constructor*, *data constructor* and *deﬁned function*

symbols, respectively. We also assume a countable set T V*ar* of *type variables*.

*Types τ* ∈ *T ype*Σ have the syntax *τ* ::= *α* | *C τ*1 *... τn* | (*τ*1*,... , τn*) | *τ* → *τ* ', where *α* ∈ T V*ar* and *C* ∈ *TCn*. By convention, *C τn* abbreviates *C τ*1 *... τn*, “→” associates to the right, *τn* → *τ* abbreviates *τ*1 → ··· → *τn* → *τ* , and the set of type variables occurring in *τ* is written T V*ar*(*τ* ). A type *τ* is called *monomorphic* iff T V*ar*(*τ* ) = ∅, and *polymorphic* otherwise. Types *C τn*, (*τ*1*,... , τn*) and *τ* → *τ* ' are used to represent constructed values, tuples and functions, respectively. A type without any occurrence of “→” is called a *datatype*. Each *n*-ary *c* ∈ *DCn* comes with a principal type declaration *c* :: *τn* → *C αk*, where *n, k* ≥ 0, *α*1*,... , αk* are pairwise different, *τi* are datatypes, and T V*ar*(*τi*) ⊆ {*α*1,. . . , *αk*} for all 1 ≤ *i* ≤ *n*. Also, each *n*-ary *f* ∈ *DFn* comes with a principal type declaration *f* :: *τn* → *τ* , where *τi*, *τ* are arbitrary types. For the sake of semantic considerations, we assume a special data constructor (⊥ :: *α*) ∈ *DC*0, intended to represent an *undeﬁned data value* that belongs to every type. [3](#_bookmark4)

3 In concrete programming languages such as T OY [[1](#_bookmark21)] and *Curry* [[6](#_bookmark25)], data constructors and their principal types are introduced by datatype declarations, the principal types of defined functions can be either declared or inferred, and ⊥ does not textually occur in programs.

Intuitively, a constraint domain provides specific data elements, along with cer- tain primitive functions operating upon them. Following this idea, and extending the formal approach of [[11](#_bookmark30)] with a type system, we consider *domain speciﬁc sig- natures* Γ=⟨*BT, PF* ⟩ disjoint from Σ, where *BT* is a family of *base types* (such as *int* for integer numbers or *real* for real numbers) and *PF* is a family of *primi-* *tive function* symbols, each one with an associated principal type declaration *p* :: *τ*1→*.. .*→*τn*→*τ* (shortly, *p* :: *τn*→*τ* ), where *τ*1, *.. .*, *τn* and *τ* are datatypes. The number *n* is called *arity* of *p*, and the set of *n*-ary symbols in *PF* is noted as *PFn*. A *constraint domain* over a specific signature Γ (in short, Γ-*domain*) is a struc- ture D=⟨{U D}*d*∈*BT ,* {*p*D}*p*∈*PF* ⟩, where each *d* ∈ *BT* is interpreted as a non-empty

*d*

set UD of *base elements* of type *d*, as e.g., Z=UD or R=UD

; and interpretations

*d int real*

*p*D of primitive function symbols behave as explained in Subsection [2.3](#_bookmark5) below.

* 1. *Extended Types, Expressions, Patterns and Substitutions over a Domain* D

Given a Γ-domain D, extended types *τ* ∈ *T ype*Σ*,*Γ over Γ have the syntax *τ* ::= *α* | *d* | *C τ*1 *... τn* | *τ* → *τ* ' | (*τ*1*,... , τn*), where *d* ∈ *BT*Γ. Obviously, *T ype*Σ ⊆ *T ype*Σ*,*Γ. Given a countable infinite set V*ar* of *data variables* disjoint from T V*ar*,Σ and Γ,

*expressions e* ∈ *Exp*D *over* D have the syntax *e* ::= *X* | *u* | *h* | (*e e*1), where *X* ∈ V*ar*,

*u* ∈ UD =*def* UD, and *h* ∈ *DC*Σ ∪ *DF*Σ ∪ *P F*Γ. Note that (*e e*1)- not to be

*d*∈*BT*Γ

*d*

confused with the pair (*e, e*1) - stands for the *application* operation which applies the function denoted by *e* to the argument denoted by *e*1. Following usual conventions, we assume that application associates to the left, and we abbreviate (*e e*1 *... en*) as (*e en*). Expressions without repeated variable occurrences are called *linear*, variable- free expressions are called *ground* and expressions without any occurrence of ⊥ are called *total*. *Patterns* over D are special expressions *t* ∈ *Pat*D whose syntax is defined as *t*::=*X* | *u* | (*c tm*) | (*f tm*) | (*p tm*), where *X*∈V*ar*, *u*∈UD, *c*∈*DCn* with

Σ

*m*≤*n*, *f* ∈*DFn* with *m<n*, and *p*∈*PFn* with *m<n*. The set of all ground patterns

Σ Γ

over D is noted *GP at*D. The following classification of expressions is also useful:

(*X em*), with *X*∈V*ar* and *m*≥0, is called a *flexible expression*, while *u*∈UD and (*h em*) with *h*∈*DC*Σ∪*DF*Σ∪*P F*Γ are called *rigid expressions*. Moreover, a rigid expression

(*h em*) is called *active* iff *h* ∈ *DFn* ∪ *PFn* and *m* ≥ *n*, and *passive* otherwise.

Σ Γ

We also consider *substitutions σ*, *θ* ∈ *Subs*D over D as mappings from variables

to patterns, and by convention, we write *ε* for the identity substitution, *eσ* instead of *σ*(*e*) for any *e* ∈ *Exp*D, and *σθ* for the composition of *σ* and *θ*. A substitution *σ* such that *σσ* = *σ* is called *idempotent*. The *domain* V*dom*(*σ*) ⊆ V*ar* and *variable range* V*ran*(*σ*) ⊆ V*ar* of *σ* are defined as usual. For any set of variables *χ* ⊆ V*ar* we define the *restriction σ* TX as the substitution *σ*' such that V*dom*(*σ*') = *χ* and *σ*'(*X*)= *σ*(*X*) for all *X* ∈ *χ*. Given *χ* ⊆ V*ar*, we write *σ* =X *θ* to indicate that *σ* TX

= *θ* TX , and we abbreviate *σ* =V\X *θ* as *σ* =\X *θ*. *Type substitutions* mapping type

variables to types can be defined analogously. *Monomorphic instances τ* ' of a given type *τ* can be obtained by applying type substitutions to *τ* .

Finally, we define the *information ordering* ±D as the least partial ordering over *Exp*D such that ⊥ ±D *e* for all *e* ∈ *Exp*D and (*e e*1) ±D (*e*' *e*' ) whenever *e* ±D *e*' and *e*1 ±D *e*' . The information ordering is useful for semantic considerations.

1

1

* 1. *Interpreting Primitive Function Symbols*

Assume a specific signature Γ = ⟨*BT, PF* ⟩ and a Γ-domain D. We define the *carrier set D*D of D as the set *GP at*D of all the ground patterns over D. For each *p*∈*PFn* whose declared principal type in Γ is *p*::*τ n*→*τ* , the *interpretation* of *p* must be a set of tuples *p*D⊆*Dn*+1. By convention, we write *p*D*tn*→*t* to indicate (*tn, t*)∈*p*D. Moreover, *p*D is required to satisfy three conditions:

D

* + 1. Polarity: For all *tn, t*'*n, t, t*'∈*D*D, if *p*D *tn*→*t*,*tn*±D*t*'*n* and *t*±D*t*' then *p*D*t*'*n*→*t*'

(i.e., monotonicity w.r.t. arguments and antimonotonicity w.r.t. result).

* + 1. Radicality: For all *t**n, t*∈*D*D, if *p*D*tn*→*t* then *t* = ⊥ or else there is some total

*t*'∈*D*D such that *t*'±D*t*, and *p*D*tn*→*t*'.

* + 1. Well-Typedness: For all monomorphic *τ* '*n,τ* '∈*T ype*Σ*,*Γ and all *tn, t*∈*D*D, if

*τ* '*n*→*τ* ' is a monomorphic instance of *τn*→*τ* , D ▶*MT tn*::*τ* '*n* and *p*D*tn*→*t* then

D▶*MT t*::*τ* ' (where D▶*MT tn*::*τ* '*n* abbreviates D▶*MT t*1::*τ* ' *,... ,*D▶*MT tn*::*τ* ' ).

1 *n*

*Type judgements* of the form D▶*MT t*::*τ* ' as used in item (iii) above mean that *τ* ' is a monomorphic instance of *e*’s principal type, and can be derived by well-known type inference rules, see e.g. [[4](#_bookmark23)].

* 1. *Constraint Solutions and Constraint Solvers*

*Constraints* over a given Γ-domain D are logical statements built from atomic cons- traints by means of logical conjunction ∧ and existential quantification ∃. *Atomic constraints* can have the form ♦ (standing for truth), ◆ (standing for falsity), or *p en*→!*t* with *p*∈*PFn*, *en*∈*Exp*D and *t*∈*Pat*D total. *Atomic primitive constraints* have the form ♦, ◆ or *p tn*→!*t* with *tn*∈*Pat*D. In the sequel, the set of all primitive constraints (resp. atomic primitive constraints) over D is noted *PCon*(D), (resp. *AP Con*(D)). Three concrete constraint domains considered in this paper are:

Γ

* The *Herbrand domain* H, with no specific base type, which supports syntactic equality and disequality constraints *seq e*1*e*2→!*t* (abbreviated as *e*1==*e*2 resp. *e*1*/*=*e*2 when *t* is *true* resp. *false*) over elements of any type. See [[11](#_bookmark30)] for details.
* FD, with specific base type *int*, which supports *ﬁnite domain* constraints over

UFD=Z and the primitive functions described in [[3](#_bookmark22)] and summarized in Table 1.

*int*

* R, with specific base type *real*, which supports *real arithmetic* constraints over

R

U

*real*

=R and the primitive functions described in[[11](#_bookmark30)] and summarized in Table 2.

Ground substitutions *η* over D are called *valuations*. The set of all valuations over D is denoted *V al*D. For any *π* ∈ *PCon*(D), *Sol*D(*π*)= {*η* ∈ *V al*D|*η* satisfies *π*}

can be defined in a natural way; see [[11](#_bookmark30)] for details. Moreover, the set of solutions of

Π ⊆ *PCon*(D) is defined as *Sol*D(Π) =

*π*∈Π

*Sol*D(*π*). Therefore, sets of constraints

are interpreted as conjunctions. A variable *X* ∈ *var*(Π) such that *η*(*X*) /= ⊥ for all

*η* ∈ *Sol*D(Π) is said to be *demanded* by Π. In practical constraint domains, the set of variables demanded by Π is expected to be decidable.

For any constraint domain D we postulate a *constraint solver* given as a function

*solve*D such that for any finite Π ⊆ *AP Con*(D), *solve*D(Π) returns a finite disjunc-

*j*=1

tion *k* ∃*Y j.* (Π*j* *σj*) fulfilling the following correctness conditions:

* For all 1≤*j*≤*k*: *Y j* are new variables, Π*j* ⊆ *AP Con*(D) finite, *σj* idempotent substitution such that V*dom*(*σj* )⊆*var*(Π), V*ran*(*σj*)⊆*Y j*, and *solve*D(Π*j*)=Π*j* *ε*.
* *Sol*D(Π) = *k Sol*D(∃*Y j.* (Π*j* *σj*)) (where is interpreted as conjunction).

*j*=1

Π is called a *solved form* iff *solve*D(Π) = Π *ε*. In the sequel, we will use the following notations:

* Π ▶▶*solve*D ∃*Y* '*.* (Π' *σ*') to indicate that ∃*Y* '*.* (Π' *σ*') is ∃*Y j.* (Π*j* *σj*) for some 1 ≤ *j* ≤ *k* (successful solving step).
* Π ▶▶*solve*D ◆ to indicate that *k* = 0 (failing solving step; in this case, *Sol*D(Π)=∅). A solving step Π ▶▶*solve*D ∃*Y* '*.* (Π' *σ*') is called *admissible* w.r.t. a set of variables *U* iff the two following conditions hold:
* *Uσ*' is a set of pairwise variable-disjoint linear patterns.
* Either *U* ∩ *var*(Π')= ∅ or else some variable in *U* is demanded by Π'. This notion will be used in the goal solving calculus presented in Section 4.

# Coordination of Domains in the *CFLP* Scheme

In this section, we describe the construction of the coordination domain C built from various domains D*i*, intended to cooperate, and a mediatorial domain M, which supplies special communication constraints called *bridges*. Instances *CFLP* (C), where C is a coordination domain, provide a declarative semantic framework for cooperative *CFLP* programming and goal solving.

* 1. *Mediatorial and Coordination Domains*

Assume a Γ-domain D and a Γ'-domain D' with specific signatures Γ=⟨*BT, PF* ⟩

'

and Γ'=⟨*BT* '*,PF* '⟩. D and D' are called *joinable* iff *PF* ∩*PF* '=∅ and UD=UD for

*d d*

all *d*∈*BT* ∩*BT* '. The *amalgamated sum* D ⊕ D' of two joinable domains D and

D' is a new domain with specific signature Γ''=⟨*BT* ''*,PF* ''⟩ where *BT* ''=*BT* ∪*BT* ',

*PF* ''=*PF* ∪*PF* ', and is constructed as follows:

* For all *d*∈*BT* , UD'' =UD, and for all *d*∈*BT* ', UD'' =UD'.

*d d* D'' *d d* D

* For all *p*∈*PF* and all *tn, t*∈*D*D'' : *p tn*→*t* ⇔*def* either *tn*∈*D*D, *t*∈*D*D and *p*

or else *t*=⊥.

*tn*→*t*,

* For all *p*∈*PF* '

D'

and all *tn, t*∈*D*D'' : *p*

D''

*tn*→*t* ⇔*def* either *tn*∈*D*D' , *t*∈*D*D' and

*p tn*→*t*, or else *t*=⊥.

The amalgamated sum of *n* pairwise joinable domains can be defined analogously.

Assume *n* pairwise joinable domains D*i* with specific signatures Γ*i* = ⟨*BTi,PFi*⟩

(1 ≤ *i* ≤ *n*) and another domain M with specific signature Γ0 = ⟨*BT*0*,PF*0⟩. M is called a *mediatorial domain* for D1*,... ,* D*n* iff

* *BT*0 ⊆ *n BTi*, and for all 1≤*i*≤*n*: *P F*0 ∩ *P Fi* = ∅.

*i*=1

* For each *p* ∈ *P F*0 there exists 1≤*i, j*≤*n*, *di* ∈ *BTi* and *dj* ∈ *BTj* such that *p* is an

*equivalence* primitive *equivdi,dj* :: *di* → *dj* → *bool* and there is an injective partial

mapping *injd ,d* :: UD*i* −→ UD*j* such that, for all *t*1*, t*2*,t* ∈ *D*M: *equiv*M

*t*1 *t*2 → *t*

*i j di dj di,dj*

⇔*def t*1 ∈ *dom*(*injdi,dj* ), *t*2 = *injdi,dj* (*t*1) and *true* ±M *t*, or else *t*1 ∈ *dom*(*injdi,dj* ),

*t*2 ∈ UD*j* , *t*2 /= *injd ,d* (*t*1) and *false* ±M *t*, or else *t* = ⊥.

*dj i j*

We note that, for fixed *i, j*, 1≤*i, j*≤*n*:

* + If *d* ∈ *BTi* ∩ *BTj*, an equivalence primitive *equivd,d* :: *d* → *d* → *bool* can be defined

if wished, whose interpretation *equiv*M is based on the identity function *injd,d* =

*d,d*

*id* : UD*i*

*d*

→ UD*j*

(UD*i*

= UD*j*

due to the joinability requirements). The primitive

*equivd,d* may be useful for communication purposes in case that D*i* and D*j* have different primitives involving the common base type *d*.

*d*

*d*

*d*

* + There can be none, one, or more than one possibilities of choosing base types *di*

∈ *BTi*, *dj* ∈ *BTj* such that an equivalence primitive *equivdi,dj* :: *di* → *dj* → *bool* is available in M. An equivalence primitive is called *redundant* iff there is some other equivalence primitive whose interpretation is based on the same partial injection or its inverse. We assume that no redundant equivalence primitives are available in

M. If *equivdi,dj* is available in M for some *di* ∈ *BTi*, *dj* ∈ *BTj*, we say that D*i* and

D*j* are *comparable*.

Assume now *n* given pairwise joinable domains D1*, ... ,* D*n* with specific sig- natures Γ1*, ... ,* Γ*n* and a mediatorial domain M for D1*,... ,* D*n*. Then, the *n* +1 domains M*,* D1*,... ,* D*n* are pairwise joinable, and the amalgamated sum C = M⊕ D1 ⊕ *...* ⊕ D*n* can be built. In the sequel, we assume that the Herbrand domain H is taken as one of the D*i*, and thus C = M⊕ H⊕ D1 ⊕ *...* ⊕ D*n*. Such a C is called a *coordination domain*, because *CFLP* (C) supports coordinated *CFLP* program- ming, using *bridge constraints* of the form *e*1#==*di,dj e*2 =*def equivdi,dj e*1*e*2→!*true* for communication between D*i* and D*j* (this will work for all the equivalence primi- tives available in the mediatorial domain M).

The instance *CRW L*(C) of the *Constraint ReWriting Logic CRW L* presented in

[[11](#_bookmark30)] provides a declarative semantics for *CFLP* (C) programming, whose usefulness for correctness results will be seen in Subsection 4.4.

* 1. *Bridges and Projections for Cooperative Goal Solving*

The cooperative goal solving calculus for *CFLP* (C) described in Section 4 below, stores bridge constraints in a special store *M* and uses them for enabling coope- ration between different solvers. More precisely, bridge constraints of the form *e*1#==*di,dj e*2 can be used either for *binding* or *projection* purposes. Binding simply instantiates a variable occurring at one end of a bridge whenever the other end of the bridge becomes a primitive value. Projection is a more complex operation which in- fers constraints to be placed in D*j*’s store from the constraints available in D*i*’s store and the relevant bridges available in *M* . This enables each solver to take advan- tage of the computations performed by other solvers. For every pair *i, j* such that D*i* and D*j* are comparable, we postulate a projection function *projections*D*i*→D*j*

such that for any *π* ∈ *AP Con*(D*i*) and any finite set *M* of bridge constraints,

*projections*D*i*→D*j* (*π, M* ) returns a finite disjunction *l* ∃*Y k.* Π' fulfilling the follo-

*k*=1

*k*

wing *safety conditions*:

* + 1. For all 1 ≤ *k* ≤ *l*: *Y k* are new variables, and Π' ⊆ *AP Con*(D*j*) is finite.

*l * ' *k* '

* + 1. *Sol*C(*π* ∧ *M* ) ⊆

as conjunctions).

*k*=1 *Sol*C (∃*Y k.* (*π* ∧ Π*k* ∧ *M* )) (where *M* and Π*k* are interpreted

In the sequel, we use the notation (*π, M* ) ▶▶ D*i*→D*j* ∃*Y* '*.* Π' to indicate that

*projections*

*k*

∃*Y* '*.* Π' is ∃*Y k.* Π'

for some 1 ≤ *k* ≤ *l* (successful projection step). Our projections

are inspired by those of [[7](#_bookmark26),[8](#_bookmark27)], but our proposal of bridge constraints is a novelty. [4](#_bookmark11) Following the terminology of [[8](#_bookmark27)], we say that a projection returning *k* alternatives is *strong* if *k >* 1 and *weak* otherwise.

In order to maximize the opportunities for projection, we postulate for each pair *i, j* such that D*i* and D*j* are comparable a function *bridges*D*i*→D*j* such that for any *π* ∈ *AP Con*(D*i*) and any finite set *M* of bridge constraints, *bridges*D*i*→D*j* (*π, M* ) returns a finite set *M* ' of new bridge constraints involving new variables *V* , so that the following *safety condition* holds: *Sol*C(*π* ∧ *M* ) ⊆ *Sol*C(∃*V.* (*π* ∧ *M* ∧ *M* ')) (where *M* and *M* ' are interpreted as conjunctions).

|  |  |  |
| --- | --- | --- |
| *π* ∈ *APCon*(FD) | *bridges*FD→R(*π, M* ) | *projections*FD→R(*π, M* ) |
| domain[*X*1*,...,Xn*] *a b* | {*Xi*#==*RXi* | 1≤i≤n, *Xi* has no bridge in *M* , *RXi* new} | {*a* ≤ *RXi*, *RXi* ≤ *b* | 1≤i≤n, (*Xi*#==  *RXi*) ∈ *M* } |
| belongs *X* [*a*1*,..., an*] | {*X*#==*RX* | *X* has no bridge in *M* ,  *RX* new} | {*min*(*a*1*,..,an*)≤*RX, RX*≤*max*(*a*1*, ., an*)  | 1≤i≤n, (*X*#==*RX*) ∈ *M* } |
| *t*1#*<t*2 (analogously #*<*=*,*#*>,*#=*>,*#=) | {*Xi*#==*RXi* | 1≤i≤2, *ti* is a varia- ble *Xi* with no bridge in *M* , *RXi* new} | {*t*R *< t*R | For 1≤i≤2: either *ti* is an  1 2  integer constant *n* and *t*R is *n*, or else  *i*  *ti* is a variable *Xi*, (*Xi*#==*RXi*) ∈ *M* ,  and *t*R is *RXi*}  *i* |
| *t*1#+*t*2 →!*t*3 (analo- gously #−*,* #∗) | {*Xi*#==*RXi*|1≤i≤3,*ti* is a variable  *Xi* with no bridge in *M* ,*RXi* new} | {*t*R + *t*R →! *t*R | For 1≤i≤3: *t*R is  1 2 3 *i*  determined as in the previous case} |

Table 1

Bridge Constraints and Projections from FD to R

As a concrete example, Table 1 and Table 2 show a partial description of the functions *bridges* and *projections* between the comparable domains FD and R, where bridges constraints written as *u*#==*v* are based on an equivalence primitive *equiv* :: *int* → *real* → *bool*. The tables do not show all possible cases due to lack of space. Some cases omitted here can be found in [[2](#_bookmark20)].

# Coordinated *CFLP* Programming

In this section, we discuss the syntax of *CFLP* (C)-programs and admissible goals for programs, in order to set the basis for coordinated programming in the *CFLP* scheme using lazy narrowing with cooperation of constraint solvers.

* 1. *Structure of Program Rules and Goals*

*CFLP* (C)-*programs* are sets of constrained rewriting rules that define the beha- vior of possibly higher-order and/or non-deterministic lazy functions over C, called *program rules*. More precisely, a program rule for a defined function symbol *f* ∈

4 Projections in [[8](#_bookmark27)] depend on the set of variables common to the stores of D*i* and D*j*. In our *CFLP*

framework, well-typing usually prevents the occurrence of one and the same variable in two different stores.

|  |  |  |
| --- | --- | --- |
| *π* ∈ *APCon*(R) | *bridges*R→FD(*π, M* ) | *projections*R→FD(*π, M* ) |
| *RX<*= *RY* | ∅ (no bridges are created) | {*X*#*<*=*Y* |(*X*#==*RX*),(*Y* #==*RY* )∈*M* } |
| *RX<*= *a* | ∅ (no bridges are created) | {*X*#*<*=[*a*♩ | *a*∈R, (*X*#==*RX*)∈*M* } |
| *t*1 + *t*2 →! *t*3 (ana- logously for −*,* ∗) | {*X*#==*RX* | *t*3 is a variable *RX* with no bridge in *M* , *X* new, for 1≤i≤2, *ti* is either an integer cons- tant or a variable *RXi* with bridge (*Xi*#==*RXi*) ∈ *M* } | {*t*FD #+ *t*FD →! *t*FD | For 1≤i≤3: *t*FD  1 2 3 *i*  is determined as in the previous case} |

Table 2

Bridge Constraints and Projections from R to FD

*DFn* with principal type *τn* → *τ* has the form *f tn* = *r* ⇐ *C*, where *f* ∈ *DFn*,

Σ Σ

*tn* is a linear sequence of patterns, *r* is an expression and *C* is a finite conjunction

*δ*1*,... , δm* of atomic constraints *δi* for each 1 ≤ *i* ≤ *m*, possibly including occurrences of defined function symbols. Program rules are required to be well-typed. [5](#_bookmark12)

As an example for the rest of the paper, we consider the following program fragment adapted from [[8](#_bookmark27)] and written in T OY syntax [[1](#_bookmark21)]. Function rc computes the capacity of circuits built from a set of resistors with given capacities by means of sequential and parallel composition. The program rules involve typical constraints over the domains FD and R, as well as cooperation via communication bridges *X* #== *C* with *X* :: *int* and *C* :: *real*.

data resistor = res real | seq resistor resistor | par resistor resistor type capacity :: real

rc :: resistor -> capacity

rc (res C) = C <== X #== C, belongs X [300,600,900,...,2700,3000], labeling [] [X]

rc (seq R1 R2) = rc R1 + rc R2

rc (par R1 R2) = 1/((1/rc R1) + (1/rc R2))

In the sequel, we consider *CFLP* (C)-*goals* in the general form *G* ≡ ∃*U. P* *C*

*M* *H* *S*1 *Sn*, in order to represent a generic state of the computation

with cooperation of solvers over the coordination domain C = M ⊕ H ⊕ D1 ⊕ *...*

⊕ D*n*. The symbol is interpreted as conjunction and,

* *U* is a finite set of so-called *existential variables*, intended to represent local varia- bles in the computation.
* *P* is a set of so-called *productions* of the form *e*1 → *t*1*,* *, em* → *tm*, where *ei* ∈

*Exp*D and *ti* ∈ *Pat*D for all 1 ≤ *i* ≤ *m*. [6](#_bookmark13) The set of *produced variables* of *G* is defined as the set *pvar*(*P* ) of variables occurring in *t*1 *tm*.

* *C* is a finite set of constraints to be solved, possibly including active occurrences of defined functions symbols.
* *M* is the so-called *mediatorial store* including bridge constraints of one of the four

5 The notion of well-typed *CFLP* program can be formalized by an easy extension of [[4](#_bookmark23)].

6 A production *ei* → *ti* can be viewed as a *suspension*. It is solved by evaluating *ei* by lazy narrowing and unifying the result with *ti*.

following forms: *X*#==*d ,d X*' or *u*#==*d ,d X*' or *X*#==*d ,d u*' or *u*#==*d ,d u*',

*i j i j i j i j*

where 1 ≤ *i, j* ≤ *n* are such that D*i* and D*j* are comparable, *X, X*' are variables,

*u* ∈ UD*i* and *u*' ∈ UD*j* .

*di dj*

* + 1. *H* is the so-called *Herbrand store*, including a finite set Π of atomic primitive H-constraints and an answer substitution *θ* with variable bindings. We use the notation (Π *θ*) to represent the store *H*.
    2. *Si* (1 ≤ *i* ≤ *n*) is a D*i constraint store* associated to the domain D*i*, including a finite set Π*i* ⊆ *PCon*(D*i*) of atomic primitive D*i*-constraints and an answer substitution *θi* with variable bindings. We use the notation (Π*i* *θi*) to represent the structure of the store *Si*.

We work with *admissible* goals *G* satisfying the *goal invariants* given in [[10](#_bookmark29),[15](#_bookmark34)]. We also write □ to denote an *inconsistent goal*. Moreover, we say that a variable *X* is a *demanded variable* in a goal *G* iff *X* is demanded by some of the constraint stores occurring in *G* in the sense explained in Subsection [2.4](#_bookmark6). For example, *X* is demanded by the FD constraint *X* #=*>* 3, but not demanded by the H constraint *suc X /*= *zero*, where *suc* and *zero* are constructor symbols.

Two special kinds of admissible goals are useful. *Initial goals*, consisting just of a finite conjunction *C* of constraints and without any existential variables; and *solved goals* (also called *solved forms*), consisting of a conjunction of constraint stores in solved form (*H*, *M* and *Si*, for each 1 ≤ *i* ≤ *n*) and empty *P* and *C* parts, possibly with existential variables.

In the sequel, we use the following notations in order to indicate the transforma- tion of a goal by applying a substitution *σ* and also adding *σ* to the corresponding store *H* or *Si* (1 ≤ *i* ≤ *n*):

* (*P* *C* *M* *H* *S*1 *...* *Sn*)@*H σ*=*def* (*Pσ* *Cσ* *Mσ* *H* T *σ* *S*1*σ* *Snσ*),

where *H* T *σ* ≡ (Π *θ*) T *σ* =*def* Π*σ* *θσ*.

* (*P* *C* *M* *H* *S*1 *.. .* *Si* *.. .* *Sn*)@*Si σ*=*def* (*Pσ* *Cσ* *Mσ* *Hσ* *S*1*σ* *Si* T

*σ* *.. .* *Snσ*), where *Si* T *σ* ≡ (Π*i* *θi*) T *σ* =*def* Π*iσ* *θiσ*.

* 1. *A Lazy Narrowing Calculus for Cooperative Goal Solving*

The *Cooperative Constrained Lazy Narrowing Calculus CCLNC*(C) presented in this section generalizes [[2](#_bookmark20)] to cooperative goal solving in *CFLP* (C) for any coordi- nation domain C and has been proved fully sound w.r.t. *CRW L*(C) semantics, as shown in Subsection [4.4](#_bookmark16). Moreover, projections (as understood in this paper and [[8](#_bookmark27)]) can operate over the constraints included in the constraint stores of the current goal, while the propagations used in [[2](#_bookmark20)] can only operate over constraints in the *C* part of the current goal, that are not yet placed in any particular store. Due to this difference, projections are computationally more powerful and more difficult to implement than propagations.

As in the case of related calculi, *CCLNC*(C) is based on goal transformation rules intended to transform a given initial goal into solved form. The presentation below distinguishes two kinds of goal transformation rules: rules for constrained lazy narrowing with *sharing*, relying on the productions (these rules are easily adapted from [[10](#_bookmark29),[15](#_bookmark34)]; see Table 3), and new rules for cooperative constraint solving, relying

DC Decomposition

∃*U. h em*→*h tm, P* *C* *M* *H* *S*1 *.. .* *Sn* ▶▶DC ∃*U. em* → *tm, P* *C* *M* *H* *S*1 *Sn*

CF Conflict Failure ∃*U. e* → *t, P* *C* *M* *H* *S*1 *Sn* ▶▶CF □

if *e* is rigid and passive, *t* ∈*/* V*ar*, *e* and *t* have conflicting roots.

SP Simple Production

∃*U. s* → *t, P* *C* *M* *H* *S*1 *.. .* *Sn* ▶▶SP ∃*U* '*.* (*P* *C* *M* *H* *S*1 *Sn*)@*H σ*

if *s* ≡ *X* ∈ V*ar*, *t* ∈*/* V*ar*, *σ* = {*X* '→ *t*} or *s* ∈ *P at*D, *t* ≡ *X* ∈ V*ar*, *σ* = {*X* '→ *s*}; *U* ' ≡ *U* \ {*X*}.

IM Imitation

∃*X, U. h em*→*X, P* *C* *M* *H* *S*1 *.. .* *Sn* ▶▶IM ∃*Xm, U.*(*em* → *Xm, P* *C* *M* *H* *S*1 *Sn*)*σ*

if *h em* ∈*/ P at*D is passive, *X* is a demanded variable, *σ* = {*X* '→ *h Xm*}, and *Xm* are new variables.

EL Elimination ∃*X, U.e* → *X, P* *C* *M* *H* *S*1 *Sn*▶▶EL∃*U.P* *C* *M* *H* *S*1 *Sn*

if *X* does not occur in the rest of the goal.

DF Defined Function

∃*U. f enak* → *t, P* *C* *M* *H* *S*1 *Sn* ▶▶DF

∃*X, Y, U. en* → *tn, r* → *X, X ak* → *t, P* *C*'*, C* *M* *H* *S*1 *Sn*

if *f* ∈ *DFn* (*k* ≥ 0)*, t* ∈*/* V*ar* or *t* is a demanded variable and *R* : *f tn* = *r* ⇐ *C*' is a fresh variant of a rule in P, with *Y* = *var*(*R*) and *X* are new variables (if *k* = 0 we can omit *X*).

Σ

PC Place Constraint

∃*U. p en*→*t, P* *C* *M* *H* *S*1 *.. .* *Sn* ▶▶PC ∃*U.P* *p en*→!*t, C* *M* *H* *S*1 *Sn*

if *p* ∈ *PFn, t* ∈*/* V*ar* or *t* is a demanded variable.

Γ

FC Flatten Constraint

∃*U. P* *p en* →! *t, C* *M* *H* *S*1 *Sn* ▶▶FC

∃*V m, U. am* → *Vm, P* *p tn* →! *t, C* *M* *H* *S*1 *Sn*

if some *ei* ∈*/ P at*D, *am* are those *ei* which are not patterns, *V m* are new variables, *p tn* is obtained from

*p en* by replacing each *ei* which is not a pattern by *Vi*.

SC Submit Constraints

∃*U.P* *p tn* →! *t, C* *M* *H* *S*1 *.. .* *Si* *.. .Sn* ▶▶SC ∃*U.P* *C* *M* ' *H*' *S*1 *.. .* *S*' *Sn*

*i*

If SB cannot be used to set new bridges, and one of the following cases applies:

* If *p tn* →! *t* is a bridge *t*1 #== *t*2 then *M* ' ≡ (*t*1 #== *t*2 ∧ *M* ), *H*' ≡ *H*, and *S*' ≡ *Si*.

*i*

* If *p tn* →! *t* is *seq t*1 *t*2 →! *t* then *M* ' ≡ *M* , *H*' ≡ (*seq t*1 *t*2 →! *t* ∧ Π *θ*), and *S*' ≡ *Si*.

*i*

* If *p tn* →! *t* is a primitive constraint *π* ∈ *PCon*(D*i*) then *M* ' ≡ *M* , *H*' ≡ *H*, and *S*' ≡ (*π* ∧ Π*i* *θi*).

*i*

Table 3

Rules for Constrained Lazy Narrowing

on bridges and projections. The following two rules describe the creation of new bridge constraints stored in *M* with the aim of enabling projections, and the actual projection of constraints via bridges between any pair of constraint stores *Si* and *Sj* (1 ≤ *i, j* ≤ *n*) corresponding to comparable domains D*i* and D*j*.

SB Set Bridges

∃*U. P* *π, C* *M* *H* *S* 1 *Sn* ▶▶SB

∃*V, U. P* *π, C* *M* '*, M* *H* *S*1 *Sn*

If *π* ∈ *AP Con*(D*i*), *M* ' ≡ *bridges*D*i*→D*j* (*π, M* ) /= ∅, and *V* = *var*(*M* ')\ *var*(*M* ) are the new variables occurring in the new bridge constraints.

PR Projection

∃*U. P* *C* *M* *H* *S*1 *...* *Si* *...* *Sj* *Sn* ▶▶PR

∃*Y* '*, U. P* *C* *M* *H* *S*1 *...* *Si* *...* *S*' *Sn*

*j*

Where *Si* ≡ (*π* ∧ Π*i* *θ**i*) is the D*i*-store, *Sj* ≡ (Π*j* *θj*) is the D*j*-store, and

*j*

(*π, M* ) ▶▶

*projections*

D*i*→D*j* ∃*Y* '*.* Π', with *S*' ≡ (Π' ∧ Π*j* *θj*).

MS *M* -Solver

* ∃*U.P* *C* *X* #==

*d ,d*

*i j*

*u , M*

'

*H* *S* *.. .*

1

*S* ▶▶

*n*

MS

1

∃*U .*(*P* *C* *M* *H* *S* *.. .*

'

1

*S* )@ *σ*

*n*

*S*

*i*

If *X*∈*/pvar*(*P* ), *u*'∈U D*j* , *σ* = {*X*'→*u*} with *u*∈U D*i* such that *equiv*M *u u*'→*true* and *U* ' = *U* \ {*X*}.

*dj*

*di*

*di,dj*

* ∃*U.P* *C* *u* #==*d*

*i j*

*,d*

*X, M* *H* *S* *.. .*

1

*S* ▶▶

*n*

MS

2

∃*U .*(*P* *C* *M* *H* *S* *.. .*

'

1

*S* )@ *σ*

*n*

*S*

*j*

If *X*∈*/pvar*(*P* ), *u*∈U*di* , *σ* = {*X*'→*u* } with *u* ∈U*dj* such that *equivdi,dj u u* →*true* and *U* = *U* \ {*X*}.

D*i*

'

'

D*j*

M '

'

* ∃*U.P* *C* *u* #==

*d ,d*

*i j*

*u , M*

'

*H* *S* *.. .*

1

*S* ▶▶

*n* MS

3

∃*U. P* *C* *M* *H* *S* *.. .*

1

*Sn*

If *u*∈U*di* , *u* ∈U*dj* , and *equivdi,dj u u* →*true* .

D*i*

'

D*j*

M '

* ∃*U. P* *C* *u* #==

*d ,d*

*i j*

*u , M*

'

*H* *S* *.. .* *S* ▶▶ □

1

*n*

MS

4

If *u*∈U*di* , *u* ∈U*dj* , and *equivdi,dj u u* →*false* .

HS *H*-Solver ∃*U.P* *C* *M* *H* *S*1 *.. .* *Sn* ▶▶HS ∃*Y* '*, U.*(*P* *C* *M* (Π' *σ*) *S*1 *Sn*)@*H σ*'

D*i*

'

D*j*

M '

If *H* = (Π *σ*), and the H-solving step Π ▶▶

∃*Y .* (Π *σ* ) is admissible w.r.t. *pvar*(*P* ).

'

' '

*solve*

H

S*i*S *Si*-Solver

∃*U.P* *C* *M* *H* *S*1 *...* *Si* *.. .* *Sn* ▶▶S S ∃*Y* '*, U.*(*P* *C* *M* *H* *S*1 *...* (Π' *σi*) *Sn*)@*S σ*'

i

*i*

*i i*

If *Si* = (Π*i* *σi*), and the D*i*-solving step Π*i* ▶▶ D ∃*Y* '*.* (Π' *σ*') is admissible w.r.t. *pvar*(*P* ).

*solve i*

*i i*

SF Solving Failure ∃*U. P* *C* *M* *H* *S*1 *.. .* *Si* *Sn* ▶▶SF □

If *K*▶▶*solve*D ◆, where D is the constraint domain H or D*i* (1≤*i*≤*n*) and *K* is the set of constraints included in D's store.

Table 4

Rules for Constraint Solving

The four rules in Table 4 describe the process of *constraint solving* by means of the application of a constraint solver over the corresponding stores (*M* , *H* or *Si*). Note that the constraint solving rules impose certain technical conditions to the variable bindings produced by solvers. These conditions are needed for ensuring the admissibility of goals (see [[10](#_bookmark29),[15](#_bookmark34)] for more details).

* 1. *An Example of Cooperative Goal Solving*

In order to illustrate the behavior of *CCLNC*(C), let us discuss a goal solving example inspired by [[8](#_bookmark27)] and involving cooperation among the domains H, FD and

R. We compute all the solved forms from the constraint rc (par RA RB) == 200 and the program rules given in Subsection [4.1](#_bookmark10). At each goal transformation step, we underline which subgoal is selected. For the sake of readability, we omit explicit quantification of existential variables. See Section [5](#_bookmark18) for a comparison between the computations below and those sketched in [[8](#_bookmark27)].

 rc(par RA RB)==200  ▶▶*FC* rc(par RA RB)→C  C==200  ▶▶*PC*

rc(par RA RB)→C C==200  ▶▶*HS* rc(par RA RB)→200 ▶▶*DFrc.*3 *,DC,SP* 2 1/(1/rc(RA)+1/rc(RB))→200  ▶▶*PC* 1/(1/rc(RA)+1/rc(RB))→!200  ▶▶*FC*

1/rc(RA)+1/rc(RB)→C1  1/C1→!200  ▶▶*PC,SC*

 1/rc(RA)+1/rc(RB)→!C1  1/C1→!200 ▶▶*FC,P C*2 *,SC*

 1/rc(RA)→!C2, 1/rc(RB)→!C3  C2+C3→!C1, 1/C1→!200 ▶▶*FC*2 *,SC*2 rc(RA)→C4,rc(RB)→C51/C4→!C2,1/C5→!C3,C2+C3→!C1,1/C1→!200▶▶*RS* rc(RA)→C4, rc(RB)→C5  (1/C4)+(1/C5)==1/200 ▶▶*DFrc.*1

RA→res C6, C6→C4, rc(RB)→C5  X6#==C6,

belongs X6 [300,600,900,1200,2700,3000], labeling [] [X6] (1/C4)+(1/C5)==1/200▶▶2 *SP*

rc(RB)→C5  X6#==C6,belongs X6 [300,..,3000], labeling [] [X6] RA'→res C6 

(1/C6)+(1/C5)==1/200▶▶2 *SC*

rc(RB)→C5  X6#==C6  RA'→res C6  belongs X6 [300,..,3000], labeling [] [X6] 

(1/C6)+(1/C5)==1/200▶▶∗

*DF* 1

*rc.*

 X7#==C7,X6#==C6  RB'→res C7,RA'→res C6  belongs X7 [300,..,3000], labeling [] [X7],

belongs X6 [300,..,3000], labeling [] [X6]  (1/C6)+(1/C7)==1/200▶▶2

*PR*FD→R

X7#==C7,X6#==C6 RB'→res C7,RA'→res C6  belongs X7 [300,..,3000], labeling [] [X7], belongs X6 [300,..,3000], labeling [] [X6]  300≤C7,C7≤3000,300≤C6,C6≤3000 ,

(1/C6)+(1/C7)==1/200▶▶*RS*

X7#==C7,X6#==C6  RB'→res C7,RA'→res C6  belongs X7 [300,..,3000], labeling [] [X7], belongs X6 [300,..,3000], labeling [] [X6]  300≤C7,C7≤600,300≤C6,C6≤600,

(1/C6)+(1/C7)==1/200▶▶4

*PR*R→FD

 X7#==C7,X6#==C6  RB'→res C7, RA '→res C6  300#≤X7, X7#≤600, 300#≤X6,X6#≤600,

belongs X7 [300,..,3000], labeling [] [X7], belongs X6 [300,..,3000], labeling [] [X6]

 300≤C7, C7≤600, 300≤C6, C6≤600, (1/C6)+(1/C7)==1/200 ▶▶*FS*

At this point there are four possible continuations of the computation:

G1 ≡  300#==C7,300#==C6  RB'→res C7,RA'→res C6  X7'→300,X6'→300  300≤C7,C7≤600, 300≤C6,C6≤ 600,(1/C6)+(1/C7)==1/200 ▶▶2 *MS * RB'→res 300,RA'→res 300  300≤300,

300≤600,300≤300,300≤600,(1/300)+(1/300)==1/200 ▶▶*RS* □

G2 ≡  300#==C7,600#==C6  RB'→res C7,RA'→res C6  X7'→300,X6'→600 300≤C7,C7≤600,300≤C6,

C6≤ 600,(1/C6)+(1/C7)==1/200 ▶▶2 *MS * RB'→res 300,RA'→res 600  300≤300,300≤600, 300≤600,600≤600,(1/600)+(1/300)==1/200 ▶▶*RS * RB'→res 300,RA'→res 600

G3 ≡  600#==C7,300#==C6  RB'→res C7,RA'→res C6  X7'→600,X6'→300  300≤C7,C7≤600,300≤C6, C6≤600,(1/C6)+(1/C7)==1/200 ▶▶*MS*2 *,RS * RB'→res 600, RA'→res 300

G4 ≡  600#==C7,600#==C6  RB'→res C7,RA'→res C6  X7'→600,X6'→600  300≤C7,C7≤600,300≤C6, C6≤600,(1/C6)+(1/C7)==1/200 ▶▶*MS*2 *,RS* □

* 1. *Full Soundness of the Cooperative Goal Solving Calculus*

This section presents the main semantic result of the paper, namely full soundness of the cooperative goal solving calculus w.r.t. the declarative semantics of *CFLP* (C), formalized by means of the constraint rewriting logic *CRW L*(C). We define the notion of *solution* for an admissible goal *G* ≡ ∃*U. P* *C* *M* *H* *S*1 *...*

*Sn* and a given *CFLP* (C)-program P as a valuation *μ* ∈ *V al*(C) such that there

exists some other valuation *μ*' =\*U μ* fulfilling the following two conditions: *μ*' is a solution of (*P* *C*) (which means, by definition, P ▶*CRW L*(C) (*P* *C*)*μ*' [7](#_bookmark19) ) and *μ*'

∈ *Sol*C (*M* *H* *S*1 *Sn*) (which can be proved equivalent to *μ*' ∈ *Sol*M(*M* )

∩ *Sol*H(*H*) ∩ *Sol*D1 (*S*1) ∩ *...* ∩ *Sol*D*n* (*Sn*)). We write *Sol*P (*G*) for the set of all solutions for *G*. It is easy to check that *Sol*P (*S*) = *Sol*C (*S*) for any solved goal *S*.

The following theorem proves that the goal transformation rules preserve the solutions of admissible goals and fail only in case of inconsistent goals. The proof (given in Appendix [A](#_bookmark35)) essentially relies on the correctness conditions for solvers and the safety conditions for bridges and projections, as required in Subsections [2.4](#_bookmark6) and [3.2](#_bookmark8).

Theorem 4.1 (*Full Soundness) Assume an admissible goal G not in solved form for a given CFLP* (C)*-program* P*. For any CCLNC*(C)*-rule* RL *applicable to G,*

*there exist l admissible goals Gk such that G* ▶▶RL*,*P *Gk for each* 1 ≤ *k* ≤ *l and*

*Sol*P (*G*) = *l Sol*P (*Gk*)*. Moreover, the transformation steps fail only in case of*

*k*=1

*inconsistent goals (i.e., if G* ▶▶RL*,*P □ *then Sol*P (*G*) = ∅*).*

The soundness of the calculus follows easily from Theorem [4.1](#_bookmark17). It ensures that the solved forms obtained as computed answers for an initial goal using the rules of the cooperative goal solving calculus are indeed semantically valid answers of *G*.

Corollary 4.2 (*Soundness) Let G an initial admissible goal and* P *a CFLP* (C)*-*

*program such that G* ▶▶∗ *S, where S is a solved goal. Then, Sol*C(*S*) ⊆ *Sol*P (*G*)*.*

P

Proof. As an obvious consequence of Theorem [4.1](#_bookmark17), one gets *Sol*P (*G*') ⊆ *Sol*P (*G*) for any *G*' such that *G* ▶▶P *G*'. From this, an easy induction shows that *Sol*P (*S*) ⊆

*Sol*P (*G*) holds for each solved form *S* such that *G* ▶▶∗

P

*S*. Since *Sol*P (*S*)= *Sol*C(*S*),

the corollary is proved.

# Conclusions and Future Work

This paper contributes to the investigation of cooperation among solvers in declara- tive programming languages. A small survey of related work has been presented in the introduction. We have focused on coordinated goal solving techniques suitable for constraint functional logic languages such as T OY and *Curry*. Extending the particular proposal given in [[2](#_bookmark20)] to a quite general setting, we have presented coordi- nation domains C and a cooperative goal solving calculus *CCLNC*(C), thus showing that the *CFLP* (C) instances of the *CFLP* scheme [[11](#_bookmark30)] provide a fully sound formal framework for functional logic programming with cooperating solvers over various constraint domains. The computation model embodied in *CCLNC*(C) combines lazy narrowing with the coordinated action of various domain specific solvers.

Inspired by [[7](#_bookmark26),[8](#_bookmark27)], we have used projection operations for communication among different solvers. As a novelty w.r.t. Hofstedt’s work, projections in our setting are guided by so-called bridge constraints, provided by a mediatorial domain, which

7 See [[11](#_bookmark30)] for more details about deductions in the *CRW L*(D) constrained rewriting logic, which works in particular when D is a coordination domain C.

|  |  |
| --- | --- |
| *CFLP* (C) | *P etra Hofstedt*'*s Approach* |
| Polymorphic types | Monomorphic types (sorts) |
| Coordination domain C (with mediatorial do- main) | Combined computation domain (without media- torial domain) |
| H within C | ”The FLP Store” |
| Placing and solving constraints as independent actions | Placing and solving constraints as simultaneous actions (function *tell* ) |
| Projections guided by bridge constraints (pro- vided by a mediatorial domain) | Projections guided by common variables |
| Resistors example: Weak projections suffice (solvers can generate alternatives) | Resistors example: Strong projections claimed to be necessary |
| Declarative programming systems as goal sol- ving calculi on top of solvers for primitive cons- traints (Motivation: obtaining precise descrip- tion of goal solving procedures and strong se- mantic results) | Declarative programming systems viewed as solvers (Motivation: modeling the combination of programming languages as combination of solvers) |

Table 5

Comparison to Petra Hofstedt’s Approach

can be used to express equivalences between values of different base types. A com- parison between the *CCLNC*(C) computations for the resistors example shown in Subsection [4.3](#_bookmark15) above and the computations for the same example given in Section

3.1 of [[8](#_bookmark27)] reveals some differences between Hofstedt’s work and our approach, as summarized in Table 5. In particular, note that the *CCLNC*(C) computations can solve the resistors problem without resorting to the strong projections used for the same example in [[8](#_bookmark27)]. In our opinion, weak projections suffice for the cooperation between FD and R, since the generation of alternatives can be handled (at least in this particular but typical example) by the solvers.

As future work, we plan to implement cooperative goal solving with bridges and projections for *CFLP* (M⊕H⊕FD ⊕R) in the T OY system, by extending the im- plementation reported in [[2](#_bookmark20)]. As mentioned in Subsection [4.2](#_bookmark14), this implementation already supports bridges and a particular kind of projections, called propagations. On the other hand, we also plan to investigate completeness results for *CCLNC*(C). Obviously, the full soundness theorem [4.1](#_bookmark17) implies completeness under the additional hypothesis of a finite search space. We aim at stronger completeness results that hold under less restrictive hypotheses, like those found in [[10](#_bookmark29),[15](#_bookmark34)] and other related papers. Finally, we plan to investigate the behavior of iterated goal solving and projection operations under different strategies, which should be useful both for implemented systems and as a guide for completeness proofs.

# References

1. P. Arenas, F.J. L´opez-Fraguas, M. Rodr´ıguez-Artalejo. T OY*. A Multiparadigm Declarative Language. Version* 2*.*2*.*2 (2006), R. Caballero and J. S´anchez (Eds.), Available at [http://toy.sourceforge.net.](http://toy.sourceforge.net/)
2. S. Est´evez Mart´ın, A.J. Fern´andez, M.T. Hortal´a-Gonz´alez, M. Rodr´ıguez-Artalejo, F. Sa´enz-P´erez and R. del Vado-V´ırseda. *A Proposal for the Cooperation of Solvers in Constraint Functional Logic* *Programming*. Proceedings of PROLE’06, 14 pp. To appear in Electronic Notes on Theoretical Computer Science, 2007.
3. A.J. Fern´andez, M.T. Hortal´a-Gonz´alez, F. S´aenz-P´erez and R. del Vado-V´ırseda. *Constraint functional* *logic programming over finite domains*. To appear in the Journal of Theory and Practice of Logic Programming, volume 7(3), 2006. Available on-line in [http://arXiv.org/abs/cs/0601071.](http://arXiv.org/abs/cs/0601071)
4. J.C. Gonz´alez-Moreno, M.T. Hortal´a-Gonz´alez and M. Rodr´ıguez-Artalejo. *Polymorphic Types in Functional Logic Programming*. FLOPS’99 special issue of the Journal of Functional and Logic Programming, (2001). [http://danae.uni-muenster.de/lehre/kuchen/JFLP.](http://danae.uni-muenster.de/lehre/kuchen/JFLP)
5. L. Granvilliers, E. Monfroy, and F. Benhamou. *Cooperative solvers in constraint programming: a short introduction*. *ALP Newsletter*, 14(2), May 2001.
6. M. Hanus. *Curry: an Integrated Functional Logic Language*, Version 0.8.2, March 28, (2006).

<http://www-i2.informatik.uni-kiel.de/>∼curry/.

1. P. Hofstedt. *Cooperation and Coordination of Constraint Solvers*. Ph.D. thesis, Shaker Verlag, Aachen, 2001.
2. P. Hofstedt and P. Pepper. *Integration of Declarative and Constraint Programming*. Theory and Practice of Logic Programming, 2006.
3. J. Jaffar and M. Maher. *Constraint logic programming: a survey*. *The Journal of Logic Programming*, 19-20:503–581, 1994.
4. F.J. L´opez-Fraguas, M. Rodr´ıguez-Artalejo and R. del Vado-V´ırseda. *A Lazy Narrowing Calculus for Declarative Constraint Programming*. In Proc. ACM SIGPLAN of Int. Conf. on Principles and Practice of Declarative Programming (PPDP’04), ACM Press, pp. 43–54, 2004.
5. F.J. L´opez-Fraguas, M. Rodr´ıguez-Artalejo and R. del Vado-V´ırseda. *A New Generic Scheme for* *Functional Logic Programming with Constraints*. To appear in the Journal of *Higher-Order and Symbolic Computation*, volume 20(1/2), 2007.
6. M. Marin. *Functional Logic Programming with Distributed Constraint Solving*. PhD thesis, Johannes Kepler Universit¨at Linz, 2000.
7. M. Marin, T. Ida, and W. Schreiner. *CFLP: a Mathematica Implementation of a Distributed Constraint Solving System*. In *Third International Mathematica Symposium (IMS’99)*, Hagenberg, Austria, August 23–25 1999. Computational Mechanics Publications, WIT Press, Southampton, UK.
8. E. Monfroy. *Solver collaboration for constraint logic programming*. PhD thesis, Centre de Recherche en Informatique de Nancy, INRIA-Lorraine, November 1996.
9. R. del Vado-V´ırseda. *Declarative Constraint Programming with Definitional Trees*. In Proc. of the 5th International Conference on *Frontiers of Combining Systems* (FroCoS’05), volume 3717 of LNCS, pages 184-199, Springer, 2005.

# A Appendix: Proof of Theorem [4.1](#_bookmark17)

We present in this appendix the proof of Theorem [4.1](#_bookmark17). For each goal transforma- tion rule RL of the *CCLNC*(C) calculus, we prove the equality *Sol*P (*G*) = *l*

*k*=1

*Sol*P (*G*' ) under the assumption that *G*'

(1 ≤ *k* ≤ *l*) are the *l* admissible goals such

*k k*

that *G* ▶▶RL*,*P *G*' . Here, we restrict ourselves to the cases RL = SB, RL = PR

*k*

and RL = MS, corresponding to the goal transformation rules which are new w.r.t.

[[10](#_bookmark29)]. In each case, we assume that *G* and *G*'

*k*

(1 ≤ *k* ≤ *l*) are exactly as they appear

in the presentation of the corresponding rule RL in Subsection [4.2](#_bookmark14). Proofs for the other cases follow easily from the results in [[10](#_bookmark29)].

Rule SB:

In this case, *k* = 1. Let us write *G*' for *G*' . Assume that *μ* ∈ *Sol*P (*G*'). There exists *μ*' =\*V ,U μ* such that *μ*' is a solution for the result of dropping the existential prefix ∃*V, U* of *G*'. In particular, *μ*' ∈ *Sol*C(*M* ). Since *V* = *var*(*M* ') \ *var*(*M* ) are new variables not occurring in *G* and the logical conditions occurring in *G* under the existential prefix ∃*U* are the same as those occurring in *G*' except for the bridges in *M* ', we conclude that *μ* ∈ *Sol*P (*G*). This proves *Sol*P (*G*) ⊇ *Sol*P (*G*').

1

Assume now that *μ* ∈ *Sol*P (*G*). There exists *μ*' =\*U μ* such that *μ*' is a solu- tion for the result of dropping the existential prefix ∃*U* of *G*. In particular, *μ*' ∈ *Sol*D (*π*) and *μ*' ∈ *Sol*C(*M* ). By the *safety condition* postulated for *bridges*D*i*→D*j* (see Subsection [3.2](#_bookmark8)), it follows that *μ*' ∈ *Sol*C(∃*V.* (*π* ∧ *M* ∧ *M* ')) (where *M* and *M* ' are interpreted as conjunctions). Therefore, there exists *μ*'' =\*V μ*' such that *μ*'' ∈ *Sol*C(*π* ∧ *M* ∧ *M* '). Since the variables in *V* are new and did not occur in *G*, we can conclude that *μ*'' is a solution of all the logical conditions occurring in *G*' under the existential prefix ∃*V, U* . Therefore, we can conclude that *μ* ∈ *Sol*P (*G*'). This proves *Sol*P (*G*) ⊆ *Sol*P (*G*').

*i*

Rule PR:

Assume that *μ* ∈ *l*

*Sol*P (*G*' ). Then, *μ* ∈ *Sol*P (*G*' ) for some 1 ≤ *k* ≤ *l*,

*k*=1 *k k*

and there exists *μ*' =\*V ,U μ* such that *μ*' is a solution for the result of dropping

*k*

the existential prefix ∃*V k, U* of *G*' . In particular, *μ*' ∈ *Sol*C (*S*' )= *Sol*C(Π' ∧ Π*j*

*k j*

*θj*) ⊆ *Sol*C(Π*j* *θj*) = *Sol*C(*Sj*). Since *V* = *var*(*M* ') \ *var*(*M* ) are new variables not occurring in *G*, and the logical conditions occurring in *G* under the existential prefix ∃*U* are the same as those occurring in *G*' except for *S*' , we conclude that *μ*

*j*

∈ *Sol*P (*G*). This proves *Sol*P (*G*) ⊇ *l Sol*P (*G*' ).

*k*=1

*k*

Assume now that *μ* ∈ *Sol*P (*G*). There exists *μ*' =\*U μ* such that *μ*' is a solution for the result of dropping the existential prefix ∃*U* of *G*. In particular, *μ*' ∈ *Sol*C(*π*) and *μ*' ∈ *Sol*C(*M* ). By the *safety conditions* postulated for *projections*D*i*→D*j* (see Subsection [3.2](#_bookmark8)), there must be some projection step (*π, M* ) ▶▶*projections*D*i*→D*j*

∃*Y* '*.*Π' such that *μ*' ∈ *Sol*C(∃*Y* '*.* (*π* ∧ Π' ∧ *M* )) (where *M* and Π' are interpreted as conjunctions). Therefore, there exists *μ*'' =\*Y* ' *μ*' such that *μ*'' ∈ *Sol*C(*π* ∧ Π'

∧ *M* ). Choose a value of *k* such that 1 ≤ *k* ≤ *l* and *G* ▶▶PR*,*P *G*'

*k*

using precisely

the projection step (*π, M* ) ▶▶

*projections*

D*i*→D*j* ∃*Y* '*.* Π'. Since the variables in *Y* ' are

new and did not occur in *G*, we can conclude that *μ*'' is a solution of all the logical

conditions occurring in *G*'

*k*

under the existential prefix ∃*Y* '*, U* . Therefore, we can

*l*

conclude that *μ* ∈ *Sol*P (*G*' ). This proves *Sol*P (*G*) ⊆

*k*

*k*

*k*=1 *Sol*P (*G*' ).

Rule MS: We consider the four subcases of this rule one by one. Case MS1:

In this case *k* = 1. Let us write *G*' for *G*' . Assume that *μ* ∈ *Sol*P (*G*'). There

1

exists *μ*' =\*U* ' *μ* such that *μ*' is a solution for the result of dropping the existential

prefix ∃*U* ' of *G*'. Then *μ*' ∈ *Sol*(*σ*) (i.e., *Xμ*' = *u* = *uμ*') and thus *σμ*' = *μ*'.

Moreover, *μ*' ∈ *Sol*C (*X* #== *u*') because *Xμ*' = *u* ∈ UD*i* , *u*'*μ*' = *u*' ∈ UD*j*

*di,dj*

*di dj*

and *equiv*M

*d ,d*

*i j*

*u u*' → *true* by the applicability conditions of MS1. Therefore, *μ*' ∈

*Sol*P (*Pσ* *Cσ* *Mσ* *Hσ* *S*1*σ* *...* (Π*iσ* *θiσ*) *Snσ*). Since *σμ*'

= *μ*', we have *μ*' ∈ *Sol*P (*P* *C* *M* *H* *S*1 *...* (Π*i* *θi*) *...* *Sn*).

Since *Si* ≡ Π*i* *θi* and *μ*' ∈ *Sol*C (*X* #==*d ,d u*'), we also have *μ*' ∈ *Sol*P (*P* *C*

*i j*

*X* #==*d ,d u*'*,M* *H* *S*1 *...* *Sn*). Since *U* ' = *U* if *X* ∈*/ U* and *U* ' =

*i j*

*U* \ {*X*} otherwise, we deduce that *μ*' =\*U μ* and then *μ* ∈ *Sol*P (*G*). This proves

*Sol*P (*G*) ⊇ *Sol*P (*G*').

Assume now that *μ* ∈ *Sol*P (*G*). There exists *μ*' =\*U μ* such that *μ*' is a solution for the result of dropping the existential prefix ∃*U* of *G*. Then, *μ*' ∈

*Sol*C (*X* # ==*d ,d u*'), and thus *equiv*M

*Xμ*' *u*' → *true*. According to the appli-

*i j di,dj*

cability conditions of MS1, *Xμ*' = *u* with *u* ∈ UD*i* . Since *σ* = {*X* '→ *u*}, it follows

*di*

that *σμ*' = *μ*'. Then, *μ*' ∈ *Sol*P (*P* *C* *X* # ==*d ,d u*'*,M* *H* *S*1 *Sn*)

*i j*

⊆ *Sol*P (*P* *C* *M* *H* *S*1 *...* *Sn*). Since *μ*' = *σμ*', we also have *σμ*' ∈

*Sol*P (*P* *C* *M* *H* *S*1 *...* *Sn*) and then *μ*' ∈ *Sol*P ((*P* *C* *M* *H*

*S*1 *...* *Sn*)@*Siσ*). By definition of *U* ', we deduce that *μ*' =\*U* ' *μ* and then *μ*

∈ *Sol*P (*G*'). This proves *Sol*P (*G*) ⊆ *Sol*P (*G*').

Cases MS2 and MS3: Analogous to the case MS1. Case MS4:

In this case *k* = 0 and we must prove that *Sol*P (*G*) = ∅. Indeed, this is by

definition of *Sol*P (*G*), because *equiv*M *u u*' → *false* with *u* ∈ UD*i* and *u*' ∈ UD*j*

*di,dj di dj*

by the applicability conditions of MS4, and then *Sol*C(*u* #==*d ,d u*')= ∅.

*i j*