TLP **9** (4): 415–527, 2009.
doi:10.1017/S1471068409003780 Printed in the United Kingdom

On the cooperation of the constraint domains $\mathscr{H}$, $\mathscr{R}$, and $\mathscr{FD}$ in CFLP

S. ESTÉVEZ-MARTÍN, T. HORTALÁ-GONZÁLEZ,
M. RODRÍGUEZ-ARTALEJO and R. DEL VADO-VÍRSEDA*

Dpto. de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid, Spain
(*e-mail:* {s.estevez,teresa,mario,rdelvado}@sip.ucm.es)

F. SÁENZ-PÉREZ*

Dpto. de Ingeniería del Software e Inteligencia Artificial,
Universidad Complutense de Madrid, Spain
(*e-mail:* fernan@sip.ucm.es)

A. J. FERNÁNDEZ†

Dpto. de Lenguajes y Ciencias de la Computación,
Universidad de Málaga, Spain
(*e-mail:* afdez@lcc.uma.es)

submitted 26 September 2008; revised 6 February 2009; accepted 6 April 2009

Abstract

This paper presents a computational model for the cooperation of constraint domains and an implementation for a particular case of practical importance. The computational model supports declarative programming with lazy and possibly higher-order functions, predicates, and the cooperation of different constraint domains equipped with their respective solvers, relying on a so-called constraint functional logic programming (*CFLP*) scheme. The implementation has been developed on top of the *CFLP* system $\mathscr{TOY}$, supporting the cooperation of the three domains $\mathscr{H}$, $\mathscr{R}$, and $\mathscr{FD}$, which supply equality and disequality constraints over symbolic terms, arithmetic constraints over the real numbers, and finite domain constraints over the integers, respectively. The computational model has been proved sound and complete w.r.t. the declarative semantics provided by the *CFLP* scheme, while the implemented system has been tested with a set of benchmarks and shown to behave quite efficiently in comparison to the closest related approach we are aware of.

KEYWORDS: cooperating constraint domains, constraint functional logic programming, constrained lazy narrowing, implementation

* The work of these authors has been partially supported by projects MERIT-FORMS (TIN2005-09207-C03-03), PROMESAS-CAM (S-0505/TIC0407), and STAMP (TIN2008-06622-C03-01).
† The work of this author has been partially supported by projects TIN2008-05941 (from Spanish Ministry of Innovation and Science) and P06-TIC2250 (from Andalusia Regional Government).

1 Introduction

Constraint programming relies on *constraint solving* as a powerful mechanism for tackling practical applications. The well-known constraint logic programming (CLP) scheme (Jaffar and Lassez 1987; Jaffar and Maher 1994; Jaffar *et al.* 1998) provides a powerful and practical framework for constraint programming which inherits the clean semantics and declarative style of logic programming. Moreover, the combination of CLP with functional programming has given rise to various so-called $CFLP$ (constraint functional logic programming) schemes, developed since 1991 and aiming at a very expressive combination of the constraint, logical, and functional programming paradigms.

This paper tackles foundational and practical issues concerning the efficient use of constraints in $CFLP$ languages and systems. Both the CLP and $CFLP$ schemes must be instantiated by a parametrically given *constraint domain* $\mathcal{D}$ which provides specific data values, constraints based on specific primitive operations, and a dedicated constraint solver. Therefore, there are different *instances* $CLP(\mathcal{D})$ of the CLP scheme for various choices of $\mathcal{D}$, and analogously for $CFLP$, whose instances $CFLP(\mathcal{D})$ provide a declarative framework for any chosen domain $\mathcal{D}$. Useful "pure" constraint domains include the Herbrand domain $\mathcal{H}$ which supplies equality and disequality constraints over symbolic terms; the domain $\mathcal{R}$ which supplies arithmetic constraints over real numbers; and the domain $\mathcal{FD}$ which supplies arithmetic and finite domain constraints over integers. Practical applications, however, often involve more than one "pure" domain, and sometimes problem solutions have to be artificially adapted to fit a particular choice of domain and solver.

Combining decision procedures for theories is a well-known problem, thoroughly investigated since the seminal paper of Nelson and Oppen (1979). In constraint programming, however, the emphasis is placed in computing answers by the interaction of constraint solvers with user-given programs, rather than in deciding satisfiability of formulas. The cooperative combination of constraint domains and solvers has evolved during the last decade as a relevant research issue that is raising an increasing interest in the CLP community. Here we mention Baader and Schulz (1995), Benhamou (1996), Monfroy (1996, 1998), Granvilliers *et al.* (2001), Marin *et al.* (2001), Hofstedt (2001), Monfroy and Castro (2004), and Hofstedt and Pepper (2007) as a limited selection of references illustrating various approaches to the problem. An important idea emerging from the research in this area is that of "hybrid" constraint domain, built as a combination of simpler "pure" domains and designed to support the cooperation of its components, so that more declarative and efficient solutions for practical problems can be promoted.

1.1 Aims of this paper

The first aim of this paper is to present a computational model for the cooperation of constraint domains in the $CFLP$ context, where sophisticated functional programming features such as higher-order functions and lazy evaluation must collaborate with constraint solving. Our computational model is based on the $CFLP$ scheme

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

and goal-solving calculus recently proposed in López-Fraguas *et al.* (2004, 2007), which will be enriched with new mechanisms for modeling the intended cooperation. Moreover, we rely on the domain cooperation techniques proposed in our previous papers (Estévez-Martín *et al.* 2007a, 2007b, 2008a), where we have introduced so-called *bridges* as a key tool for communicating constraints between different domains.

Bridges are constraints of the form `X` $\#{==}_{d_i,d_j}$ `Y` which relate the values of two variables `X :: d`$_i$, `Y :: d`$_j$ of different base types, requiring them to be equivalent. For instance, `X` $\#{==}_{int,real}$ `Y` (abbreviated as `X #== Y` in the rest of the paper) constrains the real variable `Y :: real` to take an integral real value equivalent to that of the integer variable `X :: int`. Note that the two types `int` and `real` are kept distinct and their respective values are not confused.

Our cooperative computation model keeps different stores for constraints corresponding to various domains and solvers. In addition, there is a special store where the bridge constraints which arise during the computation are placed. A bridge constraint `X #== Y` available in the bridge store can be used to *project* constraints involving the variable `X` into constraints involving the variable `Y`, or vice versa. For instance, the $\mathcal{R}$ constraint `RX <= 3.4` (based on the inequality primitive `<=`—"less or equal"—for the type `real`) can be projected into the $\mathcal{FD}$ constraint `X #<= 3` (based on the inequality primitive `#<=`—"less or equal"—for the type `int`) in case that the bridge `X #== RX` is available. Projected constraints are submitted to their corresponding store, with the aim of improving the performance of the corresponding solver. In this way, projections behave as an important cooperation mechanism, enabling certain solvers to profit from (the projected forms) of constraints originally intended for other solvers.

We have borrowed the idea of constraint projection from the works of P. Hofstedt *et al.* (Hofstedt 2000a, 2000b, 2001; Hofstedt and Pepper 2007), adapting it to our *CFLP* scheme and adding bridge constraints as a novel technique which makes projections more flexible and compatible with type discipline. In order to formalize our computation model, we present a construction of *coordination domains* $\mathcal{C}$ as a special kind of "hybrid" domains built as a combination of various "pure" domains intended to cooperate. In addition to the specific constraints supplied by its various components, coordination constraints also supply bridge constraints. As particular case of practical interest, we present a coordination domain $\mathcal{C}$ tailored to the cooperation of the three pure domains $\mathcal{H}$, $\mathcal{R}$, and $\mathcal{FD}$.

Building upon the notion of coordination domain, we also present a formal goal-solving calculus called *CCLNC*($\mathcal{C}$) (standing for cooperative constraint lazy narrowing calculus over $\mathcal{C}$) which is sound and complete with respect to the instance *CFLP*($\mathcal{C}$) of the generic *CFLP* scheme. *CCLNC*($\mathcal{C}$) embodies computation rules for creating bridges, invoking constraint solvers, and performing constraint projections as well as other more ad hoc operations for communication among different constraint stores. Moreover, *CCLNC*($\mathcal{C}$) uses *lazy narrowing* (a combination of lazy evaluation and unification) for processing calls to program defined functions, ensuring that function calls are evaluated only as far as demanded by the resolution of the constraints involved in the current goal.

A second objective of the paper is to describe the implementation of a *CFLP* system which supports the cooperation of solvers via bridges and projections for the Herbrand domain $\mathcal{H}$ and the two numeric domains $\mathcal{R}$ and $\mathcal{FD}$, following the computational model provided by the $CCLNC(\mathcal{C})$ goal-solving calculus. The implementation follows the techniques summarized in our previous papers (Estévez-Martín *et al.* 2007a, 2007c, 2008b). It has been developed on top of the $\mathcal{TOY}$ system (Arenas *et al.* 2007), which is in turn implemented on top of SICStus Prolog (2007). The $\mathcal{TOY}$ system already supported noncooperative *CFLP* programming using the $\mathcal{FD}$ and $\mathcal{R}$ solvers provided by SICStus along with Prolog code for the $\mathcal{H}$ solver. This former system has been extended, including a store for bridges and implementing mechanisms for computing bridges and projections according to the $CCLNC(\mathcal{C})$ computation model.

Last but not least, another important aim of the paper is to provide some evidence on the practical use and performance of our implementation. To this purpose, we present some illustrative examples and a set of benchmarks tailored to test the performance of $CCLNC(\mathcal{C})$ as implemented in $\mathcal{TOY}$ in comparison with the closest related system we are aware of, namely the META-S tool (Frank *et al.* 2003a, 2003b, 2005) which implements Hofstedt's framework for solver cooperation (Hofstedt and Pepper 2007). The experimental results we have obtained are quite encouraging.

The present paper thoroughly revises, expands, and elaborates our previous related publications in many respects. In fact, Estévez-Martín *et al.* (2007a) was a very preliminary work which focused on presenting bridges and providing evidence for their usefulness. Building upon these ideas, Estévez-Martín *et al.* (2007b) introduced coordination domains and a cooperative goal-solving calculus over an arbitrary coordination domain, proving local soundness and completeness results, while Estévez-Martín *et al.* (2008a) further elaborated the cooperative goal-solving calculus, providing stronger soundness and completeness results and experimental data on an implementation tailored to the cooperation of the domains $\mathcal{H}$, $\mathcal{FD}$, and $\mathcal{R}$. Significant novelties in this paper include: technical improvements in the formalization of domains; a new notion of solver taking care of critical variables and well-typed solutions; a new notion of domain-specific constraint to clarify the behavior of coordination domains; various elaborations in the cooperative goal-solving transformations needed to deal with critical variables and domain-specific constraints; a more detailed presentation of the implementation results previously reported in Estévez-Martín *et al.* (2007a, 2007c, 2008b); and quite extensive comparisons to other related approaches.

1.2 Motivating examples

As a motivation for the rest of the paper, we present in this subsection a few simple examples, intended to illustrate the different cooperation mechanisms that are supported by the computation model $CCLNC(\mathcal{C})$, as well as the benefits resulting from the cooperation.

To start with, we present a small program written in $\mathcal{TOY}$ syntax, which solves the problem of searching for a two-dimensional (2-D) point lying in the intersection of a discrete grid and a continuous region. The program includes type declarations, equations for defining functions, and clauses for defining predicates. Type declarations are similar to those used in functional languages such as Haskell (Peyton-Jones 2002). Function applications use *curried notation*, also typical of Haskell and other higher-order functional languages. The equations used to define functions must be understood as conditional rewrite rules of the form $f\,\bar{t}_n \rightarrow r \Leftarrow \Delta$, whose condition Δ is a conjunction of constraints. Predicates are viewed as Boolean functions, and clauses are understood as an abbreviation of conditional rewrite rules of the form $f\,\bar{t}_n \rightarrow true \Leftarrow \Delta$, whose right-hand side is the Boolean constant *true*. Moreover, conditions consisting of a Boolean expression *exp* are understood as an abbreviation of the *strict equality* constraint *exp* == *true*, using the strict equality operator == which is a primitive operation supplied by the Herbrand domain $\mathcal{H}$. The program's text is as follows:

```
% Discrete versus continuous points:
type dPoint = (int, int)
type cPoint = (real, real)

% Sets and membership:
type setOf A = A -> bool
isIn :: setOf A -> A -> bool
isIn Set Element = Set Element

% Grids and regions as sets of points:
type grid = setOf dPoint
type region = setOf cPoint

% Predicate for computing intersections of regions and grids:
bothIn :: region -> grid -> dPoint -> bool
bothIn Region Grid (X, Y) :- X #== RX, Y #== RY,
     isIn Region (RX, RY), isIn Grid (X,Y), labeling [ ] [X,Y]

% Square grid (discrete):
square :: int -> grid
square N (X,Y) :- domain [X,Y] 0 N

% Triangular region (continuous):
triangle :: cPoint -> real -> real -> region
triangle (RX0,RY0) B H (RX,RY) :-
            RY >= RY0 - H,
            B * RY - 2 * H * RX <= B * RY0 - 2 * H * RX0,
            B * RY + 2 * H * RX <= B * RY0 + 2 * H * RX0
```

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

```
% Diagonal segment (discrete):
diagonal :: int -> grid
diagonal N (X,Y) :- domain [X,Y] 0 N, X == Y

% Parabolic line (continuous):
parabola :: cPoint -> region
parabola (RX0,RY0) (RX,RY) :- RY - RY0 == (RX - RX0) * (RX - RX0)
```

Because of all the conventions explained above, the clause for the `bothIn` predicate included in the program must be understood as an abbreviation of the rewrite rule

```
bothIn Region Grid (X,Y) -> true <==
        X #== RX, Y #== RY,
        isIn Region (RX,RY) == true, isIn Grid (X,Y) == true,
        labeling [ ] [X,Y]
```

whose condition includes two bridge constraints, two strict equality constraints provided by the domain $\mathscr{H}$, and a last constraint using the `labeling` primitive supplied by the domain $\mathscr{FD}$. The other clauses and equations in the program can be analogously understood as conditional rewrite rules whose conditions are constraints supported by some of the three domains $\mathscr{H}$, $\mathscr{R}$, or $\mathscr{FD}$.

Let us now discuss the intended meaning of the program. The `bothIn` predicate is intended to check if a given discrete point `(X,Y)` belongs to the intersection of the continuous region `Region` and the discrete grid `Grid` given as parameters, and the constraints occurring as conditions are designed to this purpose. More precisely, the two bridge constraints `X #== RX, Y #== RY` ensure that the discrete point `(X,Y)` and the continuous point `(RX,RY)` are equivalent; the two strict equality constraints `isIn Region (RX, RY) == true, isIn Grid (X,Y) == true` ensure membership to `Region` and `Grid`, respectively; and finally the $\mathscr{FD}$ constraint `labeling [ ] [X,Y]` ensures that the variables `X` and `Y` are bound to integer values.

Note that both grids and regions are represented as sets, represented themselves as Boolean functions. They can be passed as parameters because our programming framework supports higher-order programming features. The program also defines two functions `square` and `triangle`, intended to compute representations of square grids and triangular regions, respectively. Let us discuss them in turn. We first note that the type declaration for `triangle` can be written in the equivalent form `triangle :: cPoint -> real -> real -> (cPoint -> bool)`. A function call of the form `triangle (RX0,RY0) B H` is intended to return a Boolean function representing the region of all continuous 2-D points lying within the isosceles triangle with upper vertex `(RX0,RY0)`, base `B` and height `H`. Applying this Boolean function to the argument `(RX,RY)` yields a function call written as `triangle (RX0,RY0) B H (RX,RY)` and expected to return `true` in case that `(RX,RY)` lies within the intended isosceles triangle, whose three vertices are `(RX0,RY0)`, `(RX0-B/2,RY0-H)`, and `(RX0+B/2,RY0-H)`. The three sides of the triangle are mathematically characterized by the equations `RY = RY0-H`, `B*RY-2*H*RX = B*RY0-2*H*RX0` and `B*RY+2*H*RX`

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

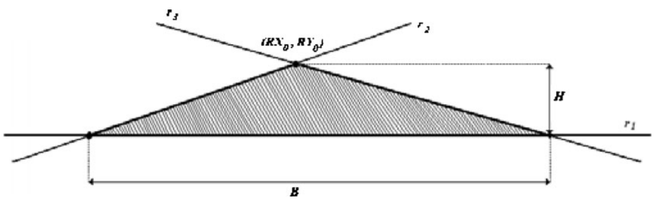

Fig. 1. Triangular region.

`= B*RY0+2*H*RX0` (corresponding to the lines r_1, r_2, and r_3 in Figure 1, respectively). Therefore, the conjunction of three linear inequality $\mathscr{R}$ constraints occurring as conditions in the clause for `triangle` succeeds for those points `(RX,RY)` lying within the intended triangle.

Similarly, the type declaration for `square` can be written in the equivalent form `square :: int -> (dPoint -> bool)`, and a function call of the form `square N` is intended to return a Boolean function representing the grid of all discrete 2-D points with coordinates belonging to the interval of integers between `0` and `N`. Therefore, a function call of the form `square N (X,Y)` must return `true` in case that `(X,Y)` lies within the intended grid, and for this reason the single $\mathscr{FD}$ constraint placed as condition in the clause for `square` has been chosen to impose the interval of integers between `0` and `N` as the domain of possible values for the variables `X` and `Y`.

Finally, the last two functions `diagonal` and `parabola` are defined in such a way that `diagonal N` returns a Boolean function representing the diagonal of the grid represented by `square N`, while `parabola (RX0,RY0)` returns a Boolean function representing the parabola whose equation is `RY-RY0 = (RX-RX0)*(RX-RX0)`. The type declarations and clauses for these functions can be understood similarly to the case of `square` and `triangle`.

Different *goals* can be posed and solved using the small program just described and the cooperative goal-solving calculus $CCLNC(\mathscr{C})$ as implemented in the $\mathscr{TOY}$ system. For the sake of discussing some of them, assume two fixed positive integer values `d` and `n` such that `n = 2*d`. Then `(d,d)` is the middle point of the grid `(square n)`, which includes $(\mathtt{n+1})^2$ discrete points. The following three goals ask for points in the intersection of this fixed square grid with three different triangular regions:

- **Goal 1**: `bothIn (triangle (d, d+0.75) n 0.5) (square n) (X,Y)`.
  This goal fails.
- **Goal 2**: `bothIn (triangle (d, d+0.5) 2 1) (square n) (X,Y)`.
  This goal computes one solution for `(X,Y)`, corresponding to the point `(d,d)`.
- **Goal 3**: `bothIn (triangle (d, d+0.5) (2*n) 1) (square n) (X,Y)`.
  This goal computes `n+1` solutions for `(X,Y)`, corresponding to the points `(0,d), (1,d), ..., (n,d)`.

These three goals are illustrated in Figure 2 for the particular case `n = 4` and `d = 2`, although the shapes and positions of the three triangles with respect to the

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

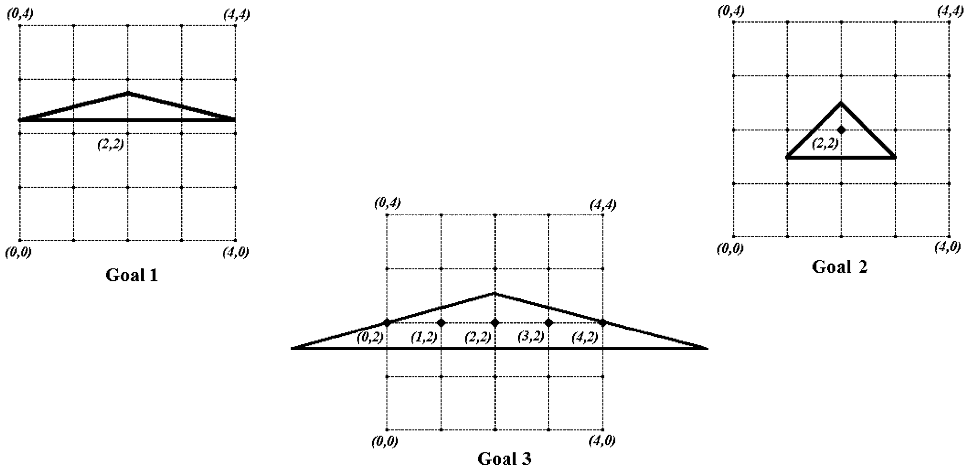

Fig. 2. Intersection of a fixed square grid with three different triangular regions.

middle point of the grid would be the same for any even positive integer `n = 2*d`. The expected solutions for each of the three goals are clear from the figures.

In the three cases, cooperation between the $\mathcal{R}$ solver and the $\mathcal{FD}$ solver is crucial for the efficiency of the computation. In the case of **Goal 2**, cooperative goal solving implemented in $\mathcal{TOY}$ according to the $CCLNC(\mathcal{C})$ computation model uses the clauses in the program and eventually reduces the problem of solving the goal to the problem of solving a constraint system that, suitably simplified, becomes:

```
X #== RX, Y #== RY,
RY >= d-0.5, RY-RX <= 0.5, RY+RX <= n+0.5,
domain [X,Y] 0 n, labeling [] [X,Y].
```

The $\mathcal{TOY}$ system has the option to enable or disable the computation of projections. When projections are disabled, the two bridges do still work as constraints, and the last $\mathcal{FD}$ constraint `labeling [] [X,Y]` forces the enumeration of all possible values for `X` and `Y` within their domains, eventually finding the unique solution `X = Y = d` after $\mathcal{O}(n^2)$ steps. When projections are enabled, the available bridges are used to project the $\mathcal{R}$ constraints `RY >= d-0.5, RY-RX <= 0.5, RY+RX <= n+0.5` into the $\mathcal{FD}$ constraints `Y #>= d, Y#-X #<= 0, Y#+X #<= n`. Since `n = 2*d`, the only possible solution of these inequalities is `X = Y = d`. Therefore, the $\mathcal{FD}$ solver drastically prunes the domains of `X` and `Y` to the singleton set {`d`}, and solving the last labeling constraint leads to the unique solution with no effort. For a big value of `n = 2*d` the performance of the computation is greatly enhanced in comparison to the case where projections are disabled, as confirmed by the experimental results given in Subsection 5.2. The expected speedup in execution time corresponds to the improvement from the $\mathcal{O}(n^2)$ steps needed to execute the labeling constraint `labeling [] [X,Y]` when the domains of both `X` and `Y` have size $\mathcal{O}(n)$, to the $\mathcal{O}(1)$ steps needed to execute the same constraint when the domains of both `X` and `Y` have

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

been pruned to size $\mathcal{O}(1)$. Similar speedups are observed when solving **Goal 1** (which finitely fails, and where the expected execution time also improves from $\mathcal{O}(\mathtt{n}^2)$ to $\mathcal{O}(1)$) and **Goal 3** (which has just `n+1` solutions, and where the expected execution time reduces from $\mathcal{O}(\mathtt{n}^2)$ to $\mathcal{O}(\mathtt{n})$).

The three goals just discussed mainly illustrate the benefits obtained by the $\mathcal{FD}$ solver from the projection of $\mathcal{R}$ constraints. In fact, when $\mathcal{TOY}$ solves these three goals according to the cooperative computation model $CCLNC(\mathcal{C})$, the available bridge constraints also allow to project the $\mathcal{FD}$ constraint `domain [X,Y] 0 n` into the conjunction of the $\mathcal{R}$ constraints `0 <= RX, RX <= n, 0 <= RY, RY <= n`. These constraints, however, are not helpful for optimizing the resolution of the previously computed $\mathcal{R}$ constraints `RY >= d-0.5, RY-RX <= 0.5, RY+RX <= n+0.5`.

In general, it seems easier for the $\mathcal{FD}$ solver to profit from the projection of $\mathcal{R}$ constraints than the other way round. This is because the solution of many practical problems is arranged to finish with solving $\mathcal{FD}$ labeling constraints, which means enumerating values for integer variables, and this process can greatly benefit from a reduction of the variables' domains due to previous projections of $\mathcal{R}$ constraints. However, the projection of $\mathcal{FD}$ constraints into $\mathcal{R}$ constraints can help to define the intended solutions even if the performance of the $\mathcal{R}$ solver does not improve. For instance, assume that the value chosen for `n = 2*d` is big, and consider the goal

- **Goal 4**: `bothIn (triangle (d,d) n d) (square 4) (X,Y).`

whose resolution eventually reduces to the problem of solving a constraint system that, suitably simplified, becomes:

```
X #== RX, Y #== RY,
RY >= 0, RY-RX <= 0, RY+RX <= n,
domain [X,Y] 0 4, labeling [] [X,Y].
```

The solutions correspond to the points lying in the intersection of a big isosceles triangle and a tiny square grid. Projecting `RY >= 0, RY-RX <= 0, RY+RX <= n` into $\mathcal{FD}$ constraints via the two bridges `X #== RX, Y #== RY` brings no significant gains to the $\mathcal{R}$ solver whose task is anyhow trivial. The $\mathcal{R}$ constraints projected from `domain [X,Y] 0 4` (i.e., `0 <= RX, RX <= 4, 0 <= RY, RY <= 4`) do not improve the performance of the $\mathcal{R}$ solver either, but they help to define the intended solutions. In this example, the last labeling constraint eventually enumerates the right solutions even if the projection of the domain constraint to $\mathcal{R}$ does not take place, but this projection would allow the $\mathcal{R}$ solver to compute suitable constraints as solutions in case that the labeling constraint were removed.

There are also some cases where the performance of the $\mathcal{R}$ solver can benefit from the cooperation with the $\mathcal{FD}$ domain. Consider, for instance, the goal

- **Goal 5**: `bothIn (parabola (2,0)) (diagonal 4) (X,Y).`

asking for points in the intersection of the discrete diagonal segment of size 4 and a parabola with vertix `(2,0)` (see Fig. 3). Solving this goal eventually reduces to

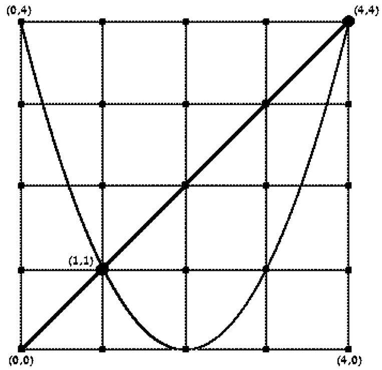

Fig. 3. Intersection of a parabolic line and a diagonal segment.

solving a constraint system that, suitably simplified, becomes:

```
X #== RX, Y #== RY,
RY == (RX-2)*(RX-2),
domain [X,Y] 0 4, X == Y, labeling [] [X,Y].
```

Cooperative goal solving as implemented in $\mathcal{TOY}$ processes the constraints within the current goal in left-to-right order, performing projections whenever possible, and sometimes delaying a constraint that cannot be managed by the available solvers. In this case, the quadratic $\mathcal{R}$ constraint `RY == (RX-2)*(RX-2)` is delayed because the $\mathcal{R}$ solver used by $\mathcal{TOY}$ (inherited form SICStus Prolog) cannot solve nonlinear constraints. However, since this strict equality relates expressions of type `real`, it is accepted as a $\mathcal{R}$ constraint and projected via the available bridges, producing the $\mathcal{FD}$ constraint `Y == (X-2)*(X-2)`, which is submitted to the $\mathcal{FD}$ solver. Next, projecting the $\mathcal{FD}$ constraint `domain [X,Y] 0 4` and solving `X == Y` causes the $\mathcal{R}$ constraints `0 <= RX, RX <= 4, 0 <= RY, RY <= 4` to be submitted to the $\mathcal{R}$ solver, and the variable `X` to be substituted in place of `Y` all over the goal. The bridges `X #== RX, Y #== RY` become then `X #== RX, X #== RY`, and the labeling constraint becomes `labeling [] [X,X]`. An especial mechanism called *bridge unification* infers from the two bridges `X #== RX, X #== RY` the strict equality constraint `RX == RY`, which is solved by substituting `RX` for `RY` all over the current goal. At this point, the delayed $\mathcal{R}$ constraint becomes `RX == (RX-2)*(RX-2)`. Finally, the $\mathcal{FD}$ constraint `labeling [] [X,X]` is solved by enumerating all the possible values for `X` allowed by its domain, and continuing a different alternative computation with each of them. Because of the bridge `X #== RX`, each integer value `v` assigned to `X` by the labeling process causes the variable `RX` to be bound to the integral real number `rv` equivalent to `v` (in our computation model, this is part of the behavior of a solver in charge of solving bridge constraints). The binding of `RX` to `rv` awakens the delayed constraint `RX == (RX-2)*(RX-2)`, which becomes the linear (and even ground) constraint `rv == (rv-2)*(rv-2)` and succeeds if `rv` is an integral solution of the delayed quadratic equation. In this way, the two solutions of **Goal 5** are eventually computed, corresponding to the two points `(X,Y)` lying in the intersection of the parabolic line and the diagonal segment: `(1,1)` and `(4,4)`, as seen in Figure 3.

All the computations described in this subsection can be actually executed in the $\mathcal{TOY}$ system and also formally represented in the cooperative goal-solving calculus $CCLNC(\mathcal{C})$. The formal representation of goal-solving computations in $CCLNC(\mathcal{C})$ performs quite many detailed intermediate steps. In particular, constraints are transformed into a flattened form (without nested calls to primitive functions) before performing projections, and especial mechanisms for creating new bridges in some intermediate steps are provided. Detailed explanations and examples are given in Section 3.

1.3 Structure of the paper

To finish the introduction, we summarize the organization of the rest of the paper. Section 2 starts by presenting the main features of the *CFLP* scheme, including a mathematical formalization of constraint domains and solvers. The presentation follows López-Fraguas *et al.* (2007), adding an explicit consideration of type discipline and an improved presentation of constraint domains, solvers, and their formal properties. The rest of the section is new with respect to previous presentations of *CFLP* schemes: it discusses bridge constraints and the construction of coordination domains, concluding with a presentation of a particular coordination domain $\mathcal{C}$ tailored to the cooperation of the domains $\mathcal{H}$, $\mathcal{R}$, and $\mathcal{FD}$. In the subsequent sections, $\mathcal{C}$ always refers to this particular coordination domain.

Section 3 presents our proposal of a computational model for cooperative programming and goal solving in $CFLP(\mathcal{C})$. Programs and goals are introduced, the cooperative goal-solving calculus $CCLNC(\mathcal{C})$ is discussed in detail, and its main formal properties (namely *soundness* and *limited completeness* w.r.t. the declarative semantics of $CFLP(\mathcal{C})$ provided by the *CFLP* scheme) are presented.

Section 4 sketches the implementation of the $CCLNC(\mathcal{C})$ computational model on top of the $\mathcal{TOY}$ system (Arenas *et al.* 2007), which is itself implemented on top of SICStus Prolog (2007). The architectural components of the current $\mathcal{TOY}$ system are described, and the extensions of $\mathcal{TOY}$ responsible for the treatment of bridges and projections according to the formal model provided by the previous section are briefly discussed.

Section 5 discusses the practical use of the $\mathcal{TOY}$ system for solving problems involving the cooperation of the domains $\mathcal{H}$, $\mathcal{R}$, and $\mathcal{FD}$. A significant set of benchmarks is analyzed in order to study how the cooperation mechanisms affect to the performance of the system, and a detailed comparison to the performance of the META-S tool is also presented.

Section 6 is devoted to a discussion of related work, trying to give an overview of different approaches in the area of cooperative constraint solving. Section 7 summarizes the main results of the paper, points out to some limitations of the current $\mathcal{TOY}$ implementation, and presents a brief outline of planned future work.

The results reported in this paper are supported by the experimental results presented in Section 5 and a number of mathematical proofs, most of which have been collected in the Appendices A.1 and A.2. In the case of reasonings concerning type discipline, we have refrained from providing full details, that would be technically tedious and distract from the main emphasis of the paper. More detailed proofs could be worked out, if desired, by adapting the techniques from González-Moreno *et al.* (2001).

2 Coordination of constraint domains in the *CFLP* scheme

The scheme presented in López-Fraguas *et al.* (2007) serves as a logical and semantic framework for lazy *CFLP* over a parametrically given constraint domain. The aim of this section is to model the coordination of several constraint domains with their respective solvers using instances $CFLP(\mathscr{C})$ of the *CFLP* scheme, where $\mathscr{C}$ is a so-called *coordination domain* built as a suitable combination of the various domains intended to cooperate. We use an enhanced version of the *CFLP* scheme, extending (López-Fraguas *et al.* 2007) with an explicit treatment of a polymorphic type discipline in the style of Hindley–Milner–Damas and an improved presentation of constraint domains, solvers, and their formal properties. In this setting, we discuss the three "pure" constraint domains $\mathscr{H}$, $\mathscr{R}$, and $\mathscr{FD}$ along with their solvers. Next, we present bridge constraints and the construction of coordination domains, concluding with the construction of a particular coordination domain $\mathscr{C}$ tailored to the cooperation of the domains $\mathscr{H}$, $\mathscr{R}$, and $\mathscr{FD}$, which is the topic of the rest of the paper.

2.1 Signatures and types

We assume a *universal signature* $\Omega = \langle TC, BT, DC, DF, PF \rangle$ consisting of five pairwise disjoint sets of symbols, where

- $TC = \bigcup_{n\in\mathbb{N}} TC^n$ is a family of countable and mutually disjoint sets of *type constructors*, indexed by arities.
- BT is a set of *base types*.
- $DC = \bigcup_{n\in\mathbb{N}} DC^n$ is a family of countable and mutually disjoint sets of *data constructors*, indexed by arities.
- $DF = \bigcup_{n\in\mathbb{N}} DF^n$ is a family of countable and mutually disjoint sets of *defined function symbols*, indexed by arities.
- $PF = \bigcup_{n\in\mathbb{N}} PF^n$ is a family of countable and mutually disjoint sets of *primitive function symbols*, indexed by arities.

The idea is that base types and primitive function symbols are related to specific constraint domains, while type constructors, data constructors, and defined function symbols are related to user-given programs. For each choice of a specific family of base types $SBT \subseteq BT$ and a specific family of primitive function symbols $SPF \subseteq PF$, we will say that $\Sigma = \langle TC, SBT, DC, DF, SPF \rangle$ is a *domain specific signature*. Note that any domain-specific signature Σ inherits all the type constructors, data constructors, and defined function symbols from the universal signature Ω, since different programs over a given constraint domain of signature Σ

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

might use them. All symbols belonging to the family $DC \cup DF \cup SPF$ are collectively called *function symbols*.

All along the paper we will work with a static type discipline based on the Hindley–Milner–Damas type system (Hindley 1969; Milner 1978; Damas and Milner 1982). A detailed study of polymorphic type discipline in the context of functional logic programming (without constraints) can be found in González-Moreno *et al.* (2001). In the sequel, we assume a countably infinite set $\mathcal{TV}ar$ of *type variables*. *Types* $\tau \in Type_\Sigma$ have the syntax $\tau ::= A \mid d \mid (c_t\ \tau_1 \ldots \tau_n) \mid (\tau_1, \ldots, \tau_n) \mid (\tau_1 \rightarrow \tau_0)$, where $A \in \mathcal{TV}ar$, $d \in SBT$, and $c_t \in TC^n$. By convention, parenthesis are omitted when there is no ambiguity, $c_t\ \overline{\tau}_n$ abbreviates $c_t\ \tau_1 \ldots \tau_n$, and "$\rightarrow$" associates to the right, $\overline{\tau}_n \rightarrow \tau$ abbreviates $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$. Types $c_t\ \overline{\tau}_n$, $(\tau_1, \ldots, \tau_n)$, and $\tau_1 \rightarrow \tau_0$ are used to represent constructed values, tuples, and functions, respectively. A type without any occurrence of "$\rightarrow$" is called a *datatype*.

Type substitutions $\sigma_t, \theta_t \in TSub_\Sigma$ are mappings from $\mathcal{TV}ar$ into $Type_\Sigma$, extended to mappings from $Type_\Sigma$ into $Type_\Sigma$ in the natural way. By convention, we write $\tau\sigma_t$ instead of $\sigma_t(\tau)$ for any type τ. Whenever $\tau' = \tau\sigma_t$ for some σ_t, we say that τ' is an *instance* of τ (or also that τ is more general than τ') and we write $\tau \preceq \tau'$.

The set of type variables occurring in τ is written $tvar(\tau)$. A type τ is called *monomorphic* iff $tvar(\tau) = \emptyset$, and *polymorphic* otherwise. A polymorphic type τ must be understood as representing all its possible monomorphic instances τ'.

Function symbols in any signature Σ are required to come along with a so-called *principal type declaration*, which indicates its most general type. More precisely,

- Each n-ary $c \in DC^n$ must have attached a principal type declaration of the form $c :: \overline{\tau}_n \rightarrow c_t\,\overline{A}_k$, where $n, k \geqslant 0$, $A_1, \ldots, A_k$ are pairwise different type variables, $c_t \in TC^k$, $\tau_1, \ldots, \tau_n$ are datatypes, and $\bigcup_{i=1}^{n} tvar(\tau_i) \subseteq \{A_1, \ldots, A_k\}$ (so-called *transparency property*).
- Each n-ary $f \in DF^n$ must have attached a principal type declaration of the form $f :: \overline{\tau}_n \rightarrow \tau$, where $\tau_i\,(1 \leqslant i \leqslant n)$ and τ are arbitrary types.
- Each n-ary $p \in SPF^n$ must have attached a principal type declaration of the form $p :: \overline{\tau}_n \rightarrow \tau$, where τ_1, …, τ_n, τ are datatypes and τ is not a type variable.

For the sake of semantic considerations, we assume a special data constructor $(\bot :: A) \in DC^0$, intended to represent an *undefined value* that belongs to any type. The type and data constructors needed to work with Boolean values and lists are also assumed to be present in the universal signature Ω. We also assume that SPF^2 includes the polymorphic primitive function symbol `== :: A -> A -> bool`, that will be written in infix notation and used to express *strict equality constraints* in those domains where it is available.

In concrete programming languages such as $\mathcal{TOY}$ (Arenas *et al.* 2007) and Curry (Hanus 2006), data constructors and their principal types are introduced by datatype declarations, the principal types of defined functions can be either declared or inferred by the compiler, the principal types of primitive functions are predefined and known to the users, and $\bot$ does not textually occur in programs.

Example 2.1 (*Signatures and Types*)
In order to illustrate the main notions concerning signatures and types, let us consider the signature Σ underlying the program presented in Subsection 1.2. There we find:

- Two base types `int` and `real` for the integer and real numeric values, respectively.
- A nullary type constructor `bool` for the type of Boolean values, and a unary type constructor `list` for the type of polymorphic lists. The concrete syntax for `list A` is `[A]`.
- `[A]` is a datatype, since it has no occurrences of the type constructor `->`. Moreover, it is polymorphic, since it includes a type variable. Among the instances of `[A]` we can find `[int]` (for lists of integers) and `[int -> int]` (for lists of functions of type `int -> int`). Note that an instance of a datatype must not be a datatype.
- Two nullary data constructors `false, true :: bool` (for Boolean values); a nullary data constructor `nil :: [A]` (for the empty list); and a binary data constructor `cons :: A -> [A] -> [A]` (for nonempty lists). The concrete syntax for `nil` (resp. `cons`) is `[]` (resp. `:`), where `:` is intended to be used as an infix operator.
- The principal types of the constructors in the previous item can be derived from the datatype declarations

```
data bool = false | true
data [A] = [] | (A : [A])
```

  which are predefined and do not need to be included within programs.
- In the program presented in Subsection 1.2 there are also *type alias* declarations, such as

```
type dPoint = (int,int)
type setOf A = A -> bool
type region = setOf dPoint
```

  Such declarations are just a practical convenience for naming certain types. They cannot involve recursion, and the names of type alias so introduced are not considered to belong to the signature.
- Defined function symbols of various arities, as, e.g., `isIn`, `square` $\in DF^2$. These two function symbols are binary because the rewrite rules given for them within the program expect two formal parameters at their left-hand sides. In general, rewrite rules included in programs for defining the behavior of symbols $f \in DF^n$ are expected to have n formal parameters at their left-hand sides. In some cases, this n may not identically correspond to the number of arrows observed in the principal type of f. For instance, although `square` $\in DF^2$, the principal type is `square :: int -> grid`. The apparent contradiction disappears by noting that `grid` is declared as a type alias for `(int,int) -> bool`. Since the type constructor `->` associates to the right, we have in fact `square :: int -> (int,int) -> bool`.
- Primitive function symbols of various arities, as, e.g., the binary primitives `#==`, `labeling`, `+` and `<=`, and the ternary primitive `domain`. The concrete

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

syntax requires `#==`, `+` and `<=` to be used in infix notation. Each primitive has a predefined principal type. For instance, `#== :: int -> real -> bool`, `+ :: real -> real -> real` and `domain :: [int] -> int -> int -> bool`. These declarations do not need to be included within programs.

2.2 Expressions and substitutions

For any domain of specific signature Σ, constraint programming will use expressions which may have occurrences of certain values of base type. Therefore, in order to define the syntax of expressions we assume a SBT-indexed family $\mathcal{B} = \{\mathcal{B}_d\}_{d \in SBT}$, where each $\mathcal{B}_d$ is a nonempty set whose elements are understood as *base values* of type d. In the sequel, we will use letters $u, v, \ldots$ to indicate base values. By an abuse of notation, we will also write $u \in \mathcal{B}$ instead of $u \in \bigcup_{d \in SBT} \mathcal{B}_d$.

Moreover, we also assume a countable infinite set $\mathcal{V}ar$ of *data variables* (disjoint from $\mathcal{TV}ar$ and Σ), and we define *applicative* Σ*-expressions* $e \in Exp_\Sigma(\mathcal{B})$ over $\mathcal{B}$ with the syntax $e ::= X \mid u \mid h \mid (e_1, \ldots, e_n) \mid (e\, e_1)$, where $X \in \mathcal{V}ar$, $u \in \mathcal{B}$, and $h \in DC \cup DF \cup SPF$.

Expressions $(e_1, \ldots, e_n)$ represent ordered n-tuples, while expressions $(e\, e_1)$—not to be confused with ordered pairs (e, e_1)—stand for the *application* of the function represented by e to the argument represented by e_1. Following a usual convention, we assume that application associates to the left, and we use the notation $(e\, \overline{e}_n)$ to abbreviate $(e\, e_1 \ldots e_n)$. More generally, parenthesis can be omitted when there is no ambiguity. Applicative syntax is common in higher-order functional languages. The usual first-order syntax for expressions can be translated to applicative syntax by means of so-called *curried notation*. For instance, $f(X, g(Y))$ becomes $(f\, X\, (g\, Y))$.

Expressions without repeated variable occurrences are called *linear*, variable-free expressions are called *ground*, and expressions without any occurrence of $\perp$ are called *total*. Some particular expressions are intended to represent data values that do not need to be evaluated. Such expressions are called Σ-patterns $t \in Pat_\Sigma(\mathcal{B})$ over $\mathcal{B}$ and have the syntax $t ::= X \mid u \mid (t_1, \ldots, t_n) \mid c\, \overline{t}_m \mid f\, \overline{t}_m \mid p\, \overline{t}_m$, where $X \in \mathcal{V}ar$, $u \in \mathcal{B}$, $c \in DC^n$ for some $m \leqslant n$, $f \in DF^n$ for some $n > m$, and $p \in SPF^n$ for some $n > m$. The restrictions concerning arities in the last three cases are motivated by the idea that an expression of the form $h\, \overline{t}_n$ (where $h \in DF^n \cup SPF^n$) is potentially evaluable and therefore not to be viewed as representing data.

The set of all ground patterns over $\mathcal{B}$ is noted $GPat_\Sigma(\mathcal{B})$. Sometimes we will write $\mathcal{U}_\Sigma(\mathcal{B})$ in place of $GPat_\Sigma(\mathcal{B})$, viewing this set as the *universe of values* over $\mathcal{B}$. The following classification of expressions is also useful: $(X\, \overline{e}_m)$ (with $X \in \mathcal{V}ar$ and $m \geqslant 0$) is called a *flexible* expression; while $u \in \mathcal{B}$ and all expressions of the form $(h\, \overline{e}_m)$ (with $h \in DC \cup DF \cup SPF$) are called *rigid*. Moreover, a rigid expression $(h\, \overline{e}_m)$ is called *passive* iff $h \in DF^n \cup SPF^n$ and $m < n$, and *active* otherwise. Tuples $(e_1, \ldots, e_n)$ are also considered as passive expressions. The idea is that any passive expression has the outermost appearance of a pattern, although it might not be a pattern in case that any of its inner subexpressions is active.

As illustrated by the program presented in Subsection 1.2, tuples are useful for programming and therefore the tuple syntax is supported by many programming languages, including $\mathcal{TOY}$. On the other hand, tuples can be treated as a particular case of constructed values, just by assuming data constructors $tup_n \in DC^n$ in the universal signature and viewing any tuple $(e_1, \ldots, e_n)$ as syntactic sugar for $tup_n\, e_1 \ldots e_n$. For this reason, in the rest of the paper we will omit the explicit mention to tuples, although we will continue to use them in examples.

As usual in programming languages that adopt a static type discipline, all expressions occurring in programs are expected to be well-typed. Deriving or checking the types of expressions relies on two kinds of information: first, the principal types of symbols belonging to the signature that we assume to be attached to the signature itself; and second, the types of variables occurring in the expression. In order to represent this second kind of information, we will use *type environments* $\Gamma = \{X_1 :: \tau_1, \ldots, X_n :: \tau_n\}$, representing the assumption that variable X_i has type τ_i for all $1 \leqslant i \leqslant n$. Following well-known ideas stemming from the work of Hindley (1969), Milner (1978), and Damas and Milner (1982), it is possible to define type inference rules for deriving *type judgements* of the form $\Sigma, \Gamma \vdash_{WT} e :: \tau$ meaning that the assertion $e :: \tau$ (in words, "e has type τ") can be deduced from the type assumptions for symbols resp. variables given in Σ resp. Γ. The reader is referred to González-Moreno *et al.* (2001) for a presentation of type inference rules well suited to functional logic languages without constraints. Adding the treatment of constraints would be a relatively straightforward task. An expression e is called *well-typed* iff there is some type environment Γ such that $\Sigma, \Gamma \vdash_{WT} e :: \tau$ can be derived for at least one type τ. Although this τ is not unique in general, it can be proved that a *most general type* τ (called the *principal type* of e and unique up to renaming of type variables) can be derived for any well-typed expression e. In practice, principal types of well-typed expressions can be automatically inferred by compilers.

We will write $\Sigma, \Gamma \vdash_{WT} \overline{e}_n :: \overline{\tau}_n$ to indicate that $\Sigma, \Gamma \vdash_{WT} e_i :: \tau_i$ can be derived for all $1 \leqslant i \leqslant n$, and $\Sigma, \Gamma \vdash_{WT} a :: \tau :: b$ to indicate that both $\Sigma, \Gamma \vdash_{WT} a :: \tau$ and $\Sigma, \Gamma \vdash_{WT} b :: \tau$ hold. An expression e is called *well-typed* iff $\Sigma, \Gamma \vdash_{WT} e :: \tau$ can be derived for some type τ using the underlying signature Σ and some suitable type environment Γ. Sometimes we will write simply $e :: \tau$, meaning that $\Sigma, \Gamma \vdash_{WT} e :: \tau$ can be derived using the underlying Σ and some proper choice of Γ (which can be just $\emptyset$ if e is ground).

For the sake of semantic considerations, it is useful to define an *information ordering* $\sqsubseteq$ over $Exp_\Sigma(\mathcal{B})$, such that $e \sqsubseteq e'$ is intended to mean that the information provided by e' is greater or equal than the information provided by e. Mathematically, $\sqsubseteq$ is defined as the least partial ordering over $Exp_\Sigma(\mathcal{B})$ such that $\bot \sqsubseteq e'$ for all $e' \in Exp_\Sigma(\mathcal{B})$ and $(e\, e_1) \sqsubseteq (e'\, e'_1)$ whenever $e \sqsubseteq e'$ and $e_1 \sqsubseteq e'_1$. For later use, we accept without proof the following lemma. It is similar to the *Typing Monotonicity Lemma* in González-Moreno *et al.* (2001) and it says that the type of any expression is also valid for its semantic approximations. It can be proved thanks to the fact that the undefined value $\bot$ belongs to all the types.

Lemma 1 (*Type Preservation Lemma*)
Assume that $\Sigma, \Gamma \vdash_{WT} e' :: \tau$ and $e \sqsubseteq e'$ hold. Then $\Sigma, \Gamma \vdash_{WT} e :: \tau$ is also true.

As part of the definition of signatures Σ, we have required a transparency property for the principal types of data constructors. Because of transparency, the types of the variables occurring in a data term t can be deduced from the type of t. It is useful to isolate those patterns that have a similar property. To this purpose, we adapt some definitions from González-Moreno *et al.* (2001). A type which can be written as $\overline{\tau}_m \rightarrow \tau$ is called *m-transparent* iff $tvar(\overline{\tau}_m) \subseteq tvar(\tau)$ and *m-opaque* otherwise. Also, defined function symbols f and primitive function symbols p are called m-transparent iff their principal types are m-transparent and m-opaque otherwise. Note that a data constructor c is always m-transparent for all $m \leqslant ar(c)$.

Then, *transparent patterns* can be defined as those having the syntax $t ::= X \mid u \mid c\,\overline{t}_m \mid f\,\overline{t}_m \mid p\,\overline{t}_m$, with $X \in \mathcal{V}ar$, $u \in \mathcal{B}$, $c \in DC^n$ for some $m \leqslant n$, $f \in DF^n$ for some $n > m$, and $p \in SPF^n$ for some $n > m$, where the subpatterns t_i in $(c\ \overline{t}_m)$, $(f\ \overline{t}_m)$ and $(p\,\overline{t}_m)$ must be recursively transparent, and the principal types of both the defined function symbol f in $(f\ \overline{t}_m)$, and the primitive function symbol p in $(p\,\overline{t}_m)$ must be m-transparent.

For instance, assume a defined function symbol with principal type declaration `snd :: A -> B -> B`. Then `snd` is 1-opaque and the pattern `(snd X)` is also opaque. In fact, the principal type `B -> B` of `(snd X)` reveals no information on the type of `X`, and different instances of `(snd X)` keep the principal type `B -> B` independently of the type of the expression substituted for `X`. Such a behavior is not possible for transparent patterns due to the *Transparency Lemma* stated without proof below. Similar results were proved in González-Moreno *et al.* (2001) in a slightly different context.

Lemma 2 (*Transparency Lemma*)

(1) Assume a transparent pattern t and two type environments Γ_1, Γ_2 such that $\Sigma, \Gamma_1 \vdash_{WT} t :: \tau$ and $\Sigma, \Gamma_2 \vdash_{WT} t :: \tau$, for a common type τ.
Then, $\Gamma_1(X) = \Gamma_2(X)$ holds for every $X \in var(t)$.

(2) Assume that $\Sigma, \Gamma \vdash_{WT} h\ \overline{a}_m\ ::\ \tau\ ::\ h\ \overline{b}_m$ holds for some m-transparent $h \in DC \cup DF \cup PF$ and some common type τ.
Then, there exist types τ_i such that $\Sigma, \Gamma \vdash_{WT} a_i\ ::\ \tau_i\ ::\ b_i$ holds for all $1 \leqslant i \leqslant m$.

Substitutions $\sigma, \theta \in Sub_\Sigma(\mathcal{B})$ over $\mathcal{B}$ are mappings from $\mathcal{V}ar$ to $Pat_\Sigma(\mathcal{B})$, extended to mappings from $Exp_\Sigma(\mathcal{B})$ to $Exp_\Sigma(\mathcal{B})$ in the natural way. For given $e \in Exp_\Sigma(\mathcal{B})$ and $\sigma \in Sub_\Sigma(\mathcal{B})$, we will usually write $e\sigma$ instead of $\sigma(e)$. Whenever $e' = e\sigma$ for some substitution σ, we say that e' is an *instance* of e (or also that e is more general than e') and we write $e \preceq e'$.

We write ε for the identity substitution and $\sigma\theta$ for the *composition* of σ and θ, such that $e(\sigma\theta) = (e\sigma)\theta$ for any expression e. A substitution σ such that $\sigma\sigma = \sigma$ is called *idempotent*. The *domain* $vdom(\sigma)$ and the *variable range* $vran(\sigma)$ of a substitution are defined as usual: $vdom(\sigma) = \{X \in \mathcal{V}ar \mid X\sigma \neq X\}$ and $vran(\sigma) = \bigcup_{X \in vdom(\sigma)} var(X\sigma)$.

A substitution σ is called *finite* iff $vdom(\sigma)$ is a finite set, and *ground* iff $X\sigma$ is a ground pattern for all $X \in vdom(\sigma)$. In the sequel, we will assume that the substitutions we work with are finite, unless otherwise said. We adopt the usual notation $\sigma = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$, whenever $vdom(\sigma) = \{X_1, \ldots, X_n\}$ and $X_i\sigma = t_i$

for all $1 \leqslant i \leqslant n$. In particular, $\varepsilon = \{\} = \emptyset$. We also write $\sigma[X \mapsto t]$ for the substitution σ' such that $X\sigma' = t$ and $Y\sigma' = Y\sigma$ for any variable $Y \in \mathcal{V}ar \setminus \{X\}$.

For any set of variables $\mathcal{X} \subseteq \mathcal{V}ar$, we define the *restriction* $\sigma \upharpoonright_{\mathcal{X}}$ as the substitution σ' such that $vdom(\sigma') = \mathcal{X}$ and $\sigma'(X) = \sigma(X)$ for all $X \in \mathcal{X}$. We use the notation $\sigma =_{\mathcal{X}} \theta$ to indicate that $\sigma \upharpoonright_{\mathcal{X}} = \theta \upharpoonright_{\mathcal{X}}$, and we abbreviate $\sigma =_{\mathcal{V}ar \setminus \mathcal{X}} \theta$ as $\sigma =_{\setminus \mathcal{X}} \theta$.

Given two substitutions σ and θ, we define the *application* of θ to σ as the substitution $\sigma \star \theta =_{def} \sigma\theta \upharpoonright vdom(\sigma)$. In other words, for any $X \in \mathcal{V}ar$, $X(\sigma \star \theta) = X\sigma\theta$ if $X \in vdom(\sigma)$ and $X(\sigma \star \theta) = X$ otherwise.

We consider two different ways of comparing given substitutions $\sigma, \sigma' \in Sub_\Sigma(\mathcal{B})$:

- σ is said to be more general than σ' over $\mathcal{X} \subseteq \mathcal{V}ar$ (in symbols, $\sigma \preceq_{\mathcal{X}} \sigma'$) iff $\sigma\theta =_{\mathcal{X}} \sigma'$ for some $\theta \in Sub_\Sigma(\mathcal{B})$. We abbreviate $\sigma \preceq_{\mathcal{V}ar} \sigma'$ as $\sigma \preceq \sigma'$ and $\sigma \preceq_{\mathcal{V}ar \setminus \mathcal{X}} \sigma'$ as $\sigma \preceq_{\setminus \mathcal{X}} \sigma'$.
- σ is said to bear less information than σ' over $\mathcal{X} \subseteq \mathcal{V}ar$ (in symbols, $\sigma \sqsubseteq_{\mathcal{X}} \sigma'$) iff $\sigma(X) \sqsubseteq \sigma'(X)$ for all $X \in \mathcal{X}$. We abbreviate $\sigma \sqsubseteq_{\mathcal{V}ar} \sigma'$ as $\sigma \sqsubseteq \sigma'$ and $\sigma \sqsubseteq_{\mathcal{V}ar \setminus \mathcal{X}} \sigma'$ as $\sigma \sqsubseteq_{\setminus \mathcal{X}} \sigma'$.

Example 2 (*Well-typed Expressions*)
Let us consider the specific signature Σ and the family of base values $\mathcal{B}$ underlying the program presented in Subsection 1.2. There we find:

- The sets of base values $\mathcal{B}_{int} = \mathbb{Z}$ and $\mathcal{B}_{real} = \mathbb{R}$.
- Well-typed expressions such as `square 4 (2,3) :: bool`, `RX-RY :: real`, `(RY-RX <= RY0-RX0) :: bool`.
- Well-typed patterns such as `3 :: int`, `3.01 :: real`, `[X,Y] :: [int]`, `square 4 :: dPoint -> bool`. Note that `[X,Y]` abbreviates `(X:(Y:[]))`, as usual in functional languages that use an infix list constructor.
- Finally, note that $\perp \sqsubseteq$ `(0 :` $\perp$`)` $\sqsubseteq$ `(0 : (1 :` $\perp$`))` $\sqsubseteq$ `...` illustrates the behavior of the information ordering $\sqsubseteq$ when restricted to the comparison of patterns belonging to the universe $\mathcal{U}_\Sigma(\mathcal{B})$. The list patterns of type `[int]` used in this example are not allowed to occur textually in programs because of the occurrences of the undefined value $\perp$, but they are meaningful as semantic representations of partially computed lists of integers.

2.3 Domains, constraints, and solutions

Intuitively, a *constraint domain* provides data values and constraints oriented to some particular application domain. Different approaches have been proposed for formalizing the notion of constraint domain, using mathematical notions borrowed from algebra, logic, and category theory (see, e.g., Jaffar and Lassez 1987; Saraswat 1992; Jaffar and Maher 1994; Jaffar *et al.* 1998). The following definition is an elaboration of the domain notion given in López-Fraguas *et al.* (2007):

Definition 1 (*Constraint Domain*)
A constraint domain of specific signature Σ (shortly, Σ-domain) is a structure $\mathcal{D} = \langle \mathcal{B}^{\mathcal{D}}, \{p^{\mathcal{D}}\}_{p \in SPF} \rangle$, where $\mathcal{B}^{\mathcal{D}} = \{\mathcal{B}^{\mathcal{D}}_d\}_{d \in SBT}$ is a SBT-indexed family of sets of base values and the *interpretation* $p^{\mathcal{D}}$ of each primitive function symbol $p :: \overline{\tau}_n \to \tau$

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

in SPF^n is required to be a set of $(n+1)$-tuples $p^{\mathcal{D}} \subseteq \mathcal{U}_\Sigma(\mathcal{B}^{\mathcal{D}})^{n+1}$. In the sequel, we abbreviate $\mathcal{U}_\Sigma(\mathcal{B}^{\mathcal{D}})$ as $\mathcal{U}_{\mathcal{D}}$ (called the *universe of values* of $\mathcal{D}$), and we write $p^{\mathcal{D}}\bar{t}_n \to t$ to indicate $(\bar{t}_n, t) \in p^{\mathcal{D}}$. The intended meaning of "$p^{\mathcal{D}}\bar{t}_n \to t$" is that the primitive function $p^{\mathcal{D}}$ with given arguments $\bar{t}_n$ can return a result t. Moreover, the interpretations of primitive symbols are required to satisfy four conditions:

(1) **Polarity**: For all $p \in SPF$, "$p^{\mathcal{D}}\bar{t}_n \to t$" behaves monotonically w.r.t. the arguments $\bar{t}_n$ and antimonotonically w.r.t. the result t.
Formally: For all $\bar{t}_n, \overline{t'}_n, t, t' \in \mathcal{U}_{\mathcal{D}}$ such that $p^{\mathcal{D}}\bar{t}_n \to t$, $\bar{t}_n \sqsubseteq \overline{t'}_n$ and $t \sqsupseteq t'$, $p^{\mathcal{D}}\overline{t'}_n \to t'$ also holds.

(2) **Radicality**: For all $p \in SPF$, as soon as the arguments given to $p^{\mathcal{D}}$ have enough information to return a result other than $\bot$, the same arguments suffice already for returning a total result.
Formally: For all $\bar{t}_n, t \in \mathcal{U}_{\mathcal{D}}$, if $p^{\mathcal{D}}\bar{t}_n \to t$ then $t = \bot$ or else there is some total $t' \in \mathcal{U}_{\mathcal{D}}$ such that $p^{\mathcal{D}}\bar{t}_n \to t'$ and $t' \sqsupseteq t$.

(3) **Well-typedness**: For all $p \in SPF$, the behavior of $p^{\mathcal{D}}$ is well-typed w.r.t. any monomorphic instance of p's principal type.
Formally: For any monomorphic type instance $(\overline{\tau'}_n \to \tau') \succeq (\bar{\tau}_n \to \tau)$ and for all $\bar{t}_n, t \in \mathcal{U}_{\mathcal{D}}$ such that $\Sigma \vdash_{WT} \bar{t}_n :: \overline{\tau'}_n$ and $p^{\mathcal{D}}\bar{t}_n \to t$, the type judgement $\Sigma \vdash_{WT} t :: \tau'$ also holds.

(4) **Strict Equality**: The primitive == (in case that it belongs to SPF) is interpreted as *strict equality* over $\mathcal{U}_{\mathcal{D}}$, so that for all $t_1, t_2, t \in \mathcal{U}_{\mathcal{D}}$, one has $t_1 ==^{\mathcal{D}} t_2 \to t$ iff some of the following three cases hold:

   (a) t_1 and t_2 are one and the same total pattern, and $true \sqsupseteq t$.
   (b) t_1 and t_2 have no common upper bound in $\mathcal{U}_{\mathcal{D}}$ w.r.t. the information ordering $\sqsubseteq$, and $false \sqsupseteq t$.
   (c) $t = \bot$.

   With this definition, it is easy to check that $==^{\mathcal{D}}$ satisfies the polarity, radicality, and well-typedness conditions.

In Subsection 2.4 we will introduce the notion of solver, and we will see that the three domains $\mathcal{H}$, $\mathcal{R}$, and $\mathcal{FD}$ mentioned in the introduction can be formalized according to the previous definition. In the rest of this subsection, we discuss how to work with constraints over a given domain.

For any given domain $\mathcal{D}$ of signature Σ, the set $\mathcal{U}_{\mathcal{D}} = \mathcal{U}_\Sigma(\mathcal{B}^{\mathcal{D}}) = GPat_\Sigma(\mathcal{B}^{\mathcal{D}})$ is called the *universe of values* of the domain $\mathcal{D}$. We will also write $Exp_{\mathcal{D}}$, $Pat_{\mathcal{D}}$, and $Sub_{\mathcal{D}}$ in place of $Exp_\Sigma(\mathcal{B}^{\mathcal{D}})$, $Pat_\Sigma(\mathcal{B}^{\mathcal{D}})$, and $Sub_\Sigma(\mathcal{B}^{\mathcal{D}})$, respectively. Note that requirement (4) in Definition 1 imposes a fixed interpretation of == as the strict equality operation $==^{\mathcal{D}}$ over $\mathcal{U}_{\mathcal{D}}$, for every domain $\mathcal{D}$ whose specific signature includes this primitive. It is easy to check that the polarity, radicality, and well-typedness requirements are satisfied by strict equality. The following definition will be useful:

Definition 2 (*Conservative Extension of a Given Domain*)
Given two domains $\mathcal{D}$, $\mathcal{D}'$ with respective signatures Σ and Σ', $\mathcal{D}'$ is called a *conservative extension* of $\mathcal{D}$ iff the following conditions hold:

(1) $\Sigma \subseteq \Sigma'$, i.e., $SBT \subseteq SBT'$ and $SPF \subseteq SPF'$.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

(2) For all $d \in SBT$, one has $\mathscr{B}_d^{\mathscr{D}'} = \mathscr{B}_d^{\mathscr{D}}$.
(3) For all $p \in SPF^n$ other than == and for every $\bar{t}_n, t \in \mathscr{U}_{\mathscr{D}}$, one has $p^{\mathscr{D}'}\,\bar{t}_n \to t$ iff $p^{\mathscr{D}}\,\bar{t}_n \to t$.

As usual in constraint programming, we define *constraints* over a given domain $\mathscr{D}$ as logical formulas built from atomic constraints by means of conjunction $\wedge$ and existential quantification $\exists$. More precisely, constraints $\delta \in Con_{\mathscr{D}}$ over the constraint domain $\mathscr{D}$ have the syntax $\delta ::= \alpha \mid (\delta_1 \wedge \delta_2) \mid \exists X \delta$, where α is any atomic constraint over $\mathscr{D}$ and $X \in \mathscr{V}ar$ is any variable. We allow two kinds of *atomic constraints* α over $\mathscr{D}$: (a) $\diamond$ and $\blacklozenge$, standing for truth (success) and falsity (failure), respectively; and (b) atomic constraints of the form $p\,\bar{e}_n \to!\, t$ with $p \in SPF^n$, where $\bar{e}_n \in Exp_{\mathscr{D}}$, $t \in Pat_{\mathscr{D}}$, and t is required to be total (i.e., without any occurrences of $\bot$). The intended meaning of $p\,\bar{e}_n \to!\, t$ constrains the value returned by the call $p\,\bar{e}_n$ to be a total pattern matching the form of t.

By convention, constraints of the form $p\,\bar{e}_n \to!\, true$ are abbreviated as $p\,\bar{e}_n$. Sometimes constraints of the form $p\,\bar{e}_n \to!\, false$ are abbreviated as $p'\,\bar{e}_n$, using some symbol p' to suggest the "negation" of p. In particular, *strict equality constraints* $e_1 == e_2$ and *strict disequality constraints* e_1 /= e_2 are understood as abbreviations of $e_1 == e_2 \to!\, true$ and $e_1 == e_2 \to!\, false$, respectively. The next definition introduces some useful notations for different kinds of constraints.

Definition 3 (*Notations for Various Kinds of Constraints*)
Given two domains $\mathscr{D}$, $\mathscr{D}'$ with respective signatures Σ and Σ', such that $\mathscr{D}'$ is a *conservative extension* of $\mathscr{D}$. Let $SPF \subseteq SPF'$ be the sets of specific primitive function symbols of $\mathscr{D}$ and $\mathscr{D}'$, respectively. We define:

(1) $ACon_{\mathscr{D}} \subseteq Con_{\mathscr{D}}$ is the set of all *atomic constraints* over $\mathscr{D}$.
(2) $APCon_{\mathscr{D}} \subseteq ACon_{\mathscr{D}}$ is the set of all *atomic primitive constraints* over $\mathscr{D}$. By definition, $\alpha \in APCon_{\mathscr{D}}$ iff α has the form $\diamond$, $\blacklozenge$ or $p\,\bar{t}_n \to!\, t$, where $\bar{t}_n \in Pat_{\mathscr{D}}$ are patterns.
(3) $PCon_{\mathscr{D}} \subseteq Con_{\mathscr{D}}$ is the set of all *primitive constraints* π over $\mathscr{D}$. By definition, a constraint $\pi \in Con_{\mathscr{D}}$ is called primitive iff all the atomic parts of π are primitive. Note that $APCon_{\mathscr{D}} = ACon_{\mathscr{D}} \cap PCon_{\mathscr{D}}$.
(4) $Con_{\mathscr{D}'} \upharpoonright SPF$ is the set of all SPF*-restricted* constraints over $\mathscr{D}'$. By definition, a constraint $\delta \in Con_{\mathscr{D}'}$ is called SPF-restricted iff all the atomic parts of δ have the form $\diamond$, $\blacklozenge$ or $p\,\bar{e}_n \to!\, t$, where $p \in SPF^n$. The subsets $APCon_{\mathscr{D}'} \upharpoonright SPF \subseteq ACon_{\mathscr{D}'} \upharpoonright SPF \subseteq Con_{\mathscr{D}'} \upharpoonright SPF$ are defined in the natural way. In particular, $APCon_{\mathscr{D}'} \upharpoonright SPF$ is the set of all the SPF-restricted atomic primitive constraints over $\mathscr{D}'$, which have the form $\diamond$ or $\blacklozenge$ or $p\,\bar{t}_n \to!\, t$, with $p \in SPF^n$, $\bar{t}_n$, $t \in Pat_{\mathscr{D}'}$ and t total.

A particular occurrence of a variable X within a constraint δ is called *free* iff it is not affected by any quantification, and *bound* otherwise. In the sequel, we will write $var(\delta)$ (resp. $fvar(\delta)$) for the set of all variables having some occurrence (resp. free occurrence) in the constraint δ. The notations $var(\Delta)$ and $fvar(\Delta)$ for a set of constraints $\Delta \subseteq Con_{\mathscr{D}}$ have a similar meaning.

The type inference rules mentioned in Subsection 2.2 can also be naturally extended to derive type judgments of the form $\Sigma, \Gamma \vdash_{WT} \delta$, meaning that the constraint δ is well-typed w.r.t. the type assumptions for symbols resp. variables given in Σ resp. Γ. Sometimes we will simply claim that δ is *well-typed* to indicate that $\Sigma, \Gamma \vdash_{WT} \delta$ can be derived using the underlying signature Σ and some suitable type environment Γ (which can be just $\emptyset$ if δ has no free variables).

The set of *valuations* $Val_{\mathcal{D}}$ over the domain $\mathcal{D}$ consists of all ground substitutions η such that $vran(\eta) \subseteq \mathcal{U}_{\mathcal{D}}$. Those valuations which satisfy a given constraint are called *solutions*. For those constraints δ that include subexpressions of the form $f\,\overline{e}_n$ for some $f \in DF^n$, the solutions of δ depend on the behavior of f, which is not included in the domain $\mathcal{D}$, but must be deduced from some user-given program, as we will see in Section 3. However, the solutions of primitive constraints depend only on the domain $\mathcal{D}$. More precisely:

Definition 4 (*Solutions of Primitive Constraints*)

(1) The *set of solutions* of a primitive constraint $\pi \in PCon_{\mathcal{D}}$ is a subset $Sol_{\mathcal{D}}(\pi) \subseteq Val_{\mathcal{D}}$ defined by recursion on the syntactic structure of π as follows:

- $Sol_{\mathcal{D}}(\diamond) = Val_{\mathcal{D}}$; $Sol_{\mathcal{D}}(\blacklozenge) = \emptyset$.
- $Sol_{\mathcal{D}}(p\,\overline{t}_n \rightarrow!\, t) = \{\eta \in Val_{\mathcal{D}} \mid (p\,\overline{t}_n \rightarrow! t)\eta \text{ ground}, p^{\mathcal{D}}\overline{t}_n\eta \rightarrow t\eta, t\eta \text{ total}\}$.
- $Sol_{\mathcal{D}}(\pi_1 \wedge \pi_2) = Sol_{\mathcal{D}}(\pi_1) \cap Sol_{\mathcal{D}}(\pi_2)$.
- $Sol_{\mathcal{D}}(\exists X \pi) = \{\eta \in Val_{\mathcal{D}} \mid \text{exists}\, \eta' \in Sol_{\mathcal{D}}(\pi) \text{ s.t. } \eta' =_{\backslash\{X\}} \eta\}$.

(2) Any set $\Pi \subseteq PCon_{\mathcal{D}}$ is interpreted as a conjunction, and therefore $Sol_{\mathcal{D}}(\Pi) = \bigcap_{\pi \in \Pi} Sol_{\mathcal{D}}(\pi)$.

(3) The *set of well-typed solutions* of a primitive constraint $\pi \in PCon_{\mathcal{D}}$ is a subset $WTSol_{\mathcal{D}}(\pi) \subseteq Sol_{\mathcal{D}}(\pi)$ consisting of all $\eta \in Sol_{\mathcal{D}}(\pi)$ such that $\pi\eta$ is well-typed.

(4) Finally, for any $\Pi \subseteq PCon_{\mathcal{D}}$ we define $WTSol_{\mathcal{D}}(\Pi) = \bigcap_{\pi \in \Pi} WTSol_{\mathcal{D}}(\pi)$.

Note that any solution $\eta \in Sol_{\mathcal{D}}(\pi)$ must verify $vdom(\eta) \supseteq fvar(\pi)$. For later use, we accept the following two technical lemmata. The first one can be easily proved by induction on the syntactic structure of Π and the second one is a simple consequence of the polarity properties of primitive functions. The notation $(WT)Sol$ used in both lemmata is intended to indicate that they are valid both for plain solutions and for well-typed solutions.

Lemma 3 (*Substitution Lemma*)
For any given $\Pi \subseteq PCon_{\mathcal{D}}$, $\sigma \in Sub_{\mathcal{D}}$ and $\eta \in Val_{\mathcal{D}}$, the equivalence $\eta \in (WT)Sol_{\mathcal{D}}(\Pi\sigma) \Leftrightarrow \sigma\eta \in (WT)Sol_{\mathcal{D}}(\Pi)$ is valid.

Lemma 4 (*Monotonicity Lemma*)
For any given $\Pi \subseteq PCon_{\mathcal{D}}$ and $\eta, \eta' \in Val_{\mathcal{D}}$ such that $\eta \sqsubseteq \eta'$ and $\eta \in (WT)Sol_{\mathcal{D}}(\Pi)$, one also has $\eta' \in (WT)Sol_{\mathcal{D}}(\Pi)$.

A given solution $\eta \in Sol_{\mathcal{D}}(\Pi)$ can bind some variables X to the undefined value $\bot$. Intuitively, this will happen whenever the value of X is not needed for checking the satisfaction of the constraints in Π. Formally, a variable X is *demanded* by a set of constraints $\Pi \subseteq PCon_{\mathcal{D}}$ iff $\eta(X) \neq \bot$ for all $\eta \in Sol_{\mathcal{D}}(\Pi)$. We write $dvar_{\mathcal{D}}(\Pi)$ to denote the set of all $X \in fvar(\Pi)$ such that X is demanded by Π.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

In practice, *CFLP* programming requires effective procedures for recognizing "obvious" occurrences of demanded variables in the case that Π is a set of atomic primitive constraints. We assume that for any practical constraint domain $\mathcal{D}$ and any primitive atomic constraint $\pi \in APCon_{\mathcal{D}}$ there is an effective way of computing a subset $odvar_{\mathcal{D}}(\pi) \subseteq dvar_{\mathcal{D}}(\pi)$. Variables $X \in odvar_{\mathcal{D}}(\pi)$ will be said to be *obviously demanded* by π. We extend the notion to finite constraint sets $\Pi \subseteq APCon_{\mathcal{D}}$ by defining the set $odvar_{\mathcal{D}}(\Pi)$ of all variables *obviously demanded* by Π as $\bigcup_{\pi \in \Pi} odvar_{\mathcal{D}}(\pi)$. In this way, it is clear that $odvar_{\mathcal{D}}(\Pi) \subseteq dvar_{\mathcal{D}}(\Pi)$ holds for any $\Pi \subseteq APCon_{\mathcal{D}}$; i.e., obviously demanded variables are always demanded. The inclusion is strict in general.

In particular, for any constraint domain $\mathcal{D}$ whose specific signature includes the strict equality primitive == and any primitive atomic constraint of the form $\pi = (t_1 \texttt{==} t_2 \rightarrow!\, t)$, $odvar_{\mathcal{D}}(\pi)$ is defined by a case distinction as follows:

- $odvar_{\mathcal{D}}(t_1 \texttt{==} t_2 \rightarrow!\, R) = \{R\}$, if $R \in \mathcal{V}ar$.
- $odvar_{\mathcal{D}}(X \texttt{==} Y) = \{X, Y\}$, if $X, Y \in \mathcal{V}ar$.
- $odvar_{\mathcal{D}}(X \texttt{==} t) = odvar_{\mathcal{D}}(t \texttt{==} X) = \{X\}$, if $X \in \mathcal{V}ar$ and $t \notin \mathcal{V}ar$.
- $odvar_{\mathcal{D}}(t_1 \texttt{==} t_2) = \emptyset$, otherwise.
- $odvar_{\mathcal{D}}(X \texttt{/=} Y) = \{X, Y\}$, if $X, Y \in \mathcal{V}ar$, X and Y not identical.
- $odvar_{\mathcal{D}}(X \texttt{/=} t) = odvar_{\mathcal{D}}(t \texttt{/=} X) = \{X\}$, if $X \in \mathcal{V}ar$ and $t \notin \mathcal{V}ar$.
- $odvar_{\mathcal{D}}(t_1 \texttt{/=} t_2) = \emptyset$, otherwise.

The inclusion $odvar_{\mathcal{D}}(\pi) \subseteq dvar_{\mathcal{D}}(\pi)$ is easy to check by considering the behavior of the interpreted strict equality operation $\texttt{==}^{\mathcal{D}}$. The method for computing $odvar_{\mathcal{D}}(\pi)$ for atomic primitive constraints based on primitive functions other than equality must be given as part of a practical presentation of the corresponding domain $\mathcal{D}$. In the sequel, we will call *critical* to those variables occurring in Π which are not obviously demanded, and we will write $cvar_{\mathcal{D}}(\Pi) = var(\Pi) \setminus odvar_{\mathcal{D}}(\Pi)$ for the set of all critical variables. As we will see in Section 3, goal-solving methods for *CFLP* programming rely on the effective recognition of critical variables. Therefore, the proper behavior of goal solving depends on well-defined methods for the computation of obviously demanded variables.

In the rest of the paper we will often use *constraint stores* of the form $S = \Pi \,\square\, \sigma$, where $\Pi \subseteq APCon_{\mathcal{D}}$ and σ is an idempotent substitution such that $vdom(\sigma) \cap var(\Pi) = \emptyset$. We will need to work with solutions of constraint stores, possibly affected by an existential prefix. This notion is defined as follows:

Definition 5 (*Solutions of Constraint Stores*)

(1) $Sol_{\mathcal{D}}(\exists \overline{Y}(\Pi \,\square\, \sigma)) = \{\eta \in Val_{\mathcal{D}} \mid \text{exists}\, \eta' \in Sol_{\mathcal{D}}(\Pi \,\square\, \sigma), \text{ s.t. } \eta' =_{\setminus \overline{Y}} \eta\}$.

(2) $Sol_{\mathcal{D}}(\Pi \,\square\, \sigma) = Sol_{\mathcal{D}}(\Pi) \cap Sol(\sigma)$.

(3) $Sol(\sigma) = \{\eta \in Val_{\mathcal{D}} \mid \eta = \sigma\eta\}$
(Note that $\eta = \sigma\eta$ holds iff $X\eta = X\sigma\eta$ for all $X \in vdom(\sigma)$).

(4) $WTSol_{\mathcal{D}}(\exists \overline{Y}(\Pi \,\square\, \sigma)) = \{\eta \in Val_{\mathcal{D}} \mid \text{ex.}\, \eta' \in WTSol_{\mathcal{D}}(\Pi \,\square\, \sigma), \text{ s.t. } \eta' =_{\setminus \overline{Y}} \eta\}$.

(5) $WTSol_{\mathcal{D}}(\Pi \,\square\, \sigma) = \{\eta \in Sol_{\mathcal{D}}(\Pi \,\square\, \sigma) \mid (\Pi \,\square\, \sigma) \star \eta \text{ is well-typed}\}$, where $(\Pi \,\square\, \sigma) \star \eta =_{def} \Pi\eta \,\square\, (\sigma \star \eta)$.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Example 3 (*Constraints and Their Solutions*)
Let us now illustrate different notions concerning constraints by referring again to the motivating example from Subsection 1.2. The domain $\mathcal{C}$ underlying this example is a "hybrid" domain supporting the cooperation of three "pure" domains named $\mathcal{H}$, $\mathcal{R}$, and $\mathcal{FD}$, as we will see in Subsections 2.4 and 2.5. For the moment, note that $\mathcal{C}$ allows to work with four different kinds of constraints, namely bridge constraints and the specific constraints supplied by $\mathcal{H}$, $\mathcal{R}$, and $\mathcal{FD}$, as explained in Section 1.

(1) Concerning well-typed constraints, we note that the small program in this example is well-typed. Therefore, all the constraints occurring there are also well-typed. For instance:
   - `domain [X,Y] 0 N` is well-typed (w.r.t. any type environment which includes the type assumptions `X :: int, Y :: int, N :: int`).
   - `RY+RX <= RY0+RX0` is also well-typed (w.r.t. any type environment which includes the type assumptions `RY :: real, RX :: real, RY0 :: real, RX0 :: real`).

   Of course, the signature underlying the example allows to write constraints such as `domain [X,Y] true 3.2`, which cannot be well-typed in any type environment. Because of static type discipline, the compiler will reject programs including ill-typed constraints.
(2) Concerning constraint solutions, note that computing by means of the cooperative goal-solving calculus presented in Section 3 eventually triggers the computation of solutions for primitive constraints. As already discussed in Subsection 1.2, solving **Goal 2** eventually leads to the following set Π of primitive constraints (understood as logical conjunction):

```
X #== RX, Y #== RY,
RY >= d-0.5, RY-RX <= 0.5, RY+RX <= n+0.5,
domain [X,Y] 0 n, labeling [] [X,Y].
```

   Π happens to be the union of three sets of primitive constraints corresponding to the three lines above: a set of two *bridge constraints* Π_M, a set of three *real arithmetical constraints* Π_R, and a set of two *finite domain constraints* Π_F. Therefore, $Sol_{\mathcal{C}}(\Pi) = Sol_{\mathcal{C}}(\Pi_M) \cap Sol_{\mathcal{C}}(\Pi_R) \cap Sol_{\mathcal{C}}(\Pi_F)$. As we have seen in Subsection 1.2, the only possibility for $\eta \in Sol_{\mathcal{C}}(\Pi)$ is η`(X) =` η`(Y) = d`, and the computation proceeds with the help of *constraint solvers* and *projections*, among other mechanisms.
(3) Concerning obviously demanded variables, let us remark that all the variables occurring in the constraint set Π shown in the previous item are obviously demanded. This will become clear from the discussion of the domains $\mathcal{H}$, $\mathcal{R}$, and $\mathcal{FD}$ in Subsection 2.4.
(4) Concerning critical variables, note that a variable may be critical either because it is demanded but not obviously demanded, or else because it is not demanded at all. For instance, variables `A` and `B` are demanded but not obviously demanded by the strict equality constraint `(A,2) == (1,B)`. Therefore, they are critical variables. To illustrate the case of critical but not demanded variables, consider the primitive constraint $\pi =$ `L /= X:Xs`. Because

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

of the definition of "obvious demand" for strict disequality constraints, variable $\mathtt{L}$ is obviously demanded by π, while $\mathtt{X}$ and $\mathtt{Xs}$ are not obviously demanded, and therefore critical. Moreover, it can be argued that neither $\mathtt{X}$ nor $\mathtt{Xs}$ is demanded by π. Variable $\mathtt{X}$ is not demanded because there exist solutions $\eta \in Sol_{\mathscr{D}}(\pi)$ such that $\eta(\mathtt{X}) = \bot$ (either with $\eta(\mathtt{L}) = [\,]$ or else with $\eta(\mathtt{L}) = \mathtt{t} : \mathtt{ts}$ such that $\eta(\mathtt{Xs})$ is different from $\mathtt{ts}$). Variable $\mathtt{Xs}$ is not demanded because of similar reasons.

2.4 Pure domains and their solvers

In order to be helpful for programming purposes, constraint domains must provide so-called *constraint solvers*, which process the constraints arising in the course of a computation. For some theoretical purposes, it suffices to model a solver as a function which maps any given constraint to one of the three different values: *true*, *false*, or *unknown* (see, e.g., Jaffar *et al.* 1998). In practice, however, solvers are expected to have the ability of reducing primitive constraints to so-called *solved forms*, which are simpler and can be shown as computed answers to the users. As discussed in the introduction (see, in particular, Subsection 1.2), the constraint domain underlying many practical problems may involve heterogeneous primitives related to different base types. In such cases, it may be not realistic to expect that a single solver for the whole domain is directly available.

In the sequel, we will make a pragmatic distinction between *pure constraint domains* which are given "in one piece" and come equipped with a solver, and *hybrid constraint domains* which are built as a combination of simpler domains and must rely on the solvers of their components. In the rest of this subsection, we give a mathematical formalization of the notion of solver tailored to the needs of the *CFLP* scheme, followed by a presentation of $\mathscr{H}$, $\mathscr{R}$, and $\mathscr{FD}$ as pure domains equipped with solvers. In the case of $\mathscr{R}$ and $\mathscr{FD}$, we limit ourselves to describe their most basic primitives, although other useful facilities are available in the $\mathscr{TOY}$ implementation. A proposal for the construction of so-called *coordination domains* as a particular kind of hybrid domains will be presented in Subsection 2.5.

2.4.1 Constraint solvers

For any pure constraint domain $\mathscr{D}$, we postulate a *constraint solver* which can reduce any given finite set Π of atomic primitive constraints to an equivalent simpler form while taking proper care of critical variables occurring in Π. Since the value of a critical variable X may be needed by some solutions of Π and irrelevant for some other solutions, we require that solvers have the ability to compute a distinction of cases discriminating such situations.

Definition 6 (*Formal Requirements for Solvers*)
A constraint solver for the domain $\mathscr{D}$ is modeled as a function $solve^{\mathscr{D}}$ which can be applied to pairs of the form $(\Pi, \mathscr{X})$, where $\Pi \subseteq APCon_{\mathscr{D}}$ is a finite set of atomic primitive constraints and $\mathscr{X} \subseteq cvar_{\mathscr{D}}(\Pi)$ is a finite set including some of the critical variables in Π, where the two extreme cases $\mathscr{X} = \emptyset$ and $\mathscr{X} = cvar_{\mathscr{D}}(\Pi)$ are

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

allowed. By convention, we may abbreviate $solve^{\mathcal{D}}(\Pi, \emptyset)$ as $solve^{\mathcal{D}}(\Pi)$. We require that any *solver invocation* $solve^{\mathcal{D}}(\Pi, \mathcal{X})$ returns a finite disjunction $\bigvee_{j=1}^{k} \exists \overline{Y}_j(\Pi_j \Box \sigma_j)$ of existentially quantified constraint stores, fulfilling the following conditions:

(1) **Fresh Local Variables:** For all $1 \leqslant j \leqslant k$: $(\Pi_j \Box \sigma_j)$ is a store, $\overline{Y}_j = var(\Pi_j \Box \sigma_j) \setminus var(\Pi)$ are fresh local variables and $vdom(\sigma_j) \cup vran(\sigma_j) \subseteq var(\Pi) \cup \overline{Y}_j$.
(2) **Solved Forms:** For all $1 \leqslant j \leqslant k$: $\Pi_j \Box \sigma_j$ is in solved form w.r.t. $\mathcal{X}$. By definition, this means that $solve^{\mathcal{D}}(\Pi_j, \mathcal{X}) = \Pi_j \Box \varepsilon$.
(3) **Safe Bindings:** For all $1 \leqslant j \leqslant k$ and for all $X \in \mathcal{X} \cap vdom(\sigma_j)$: $\sigma_j(X)$ is a constant.
(4) **Discrimination:** Each computed $\mathcal{X}$-solved form $\Pi_j \Box \sigma_j$ $(1 \leqslant j \leqslant k)$ must satisfy: either $\mathcal{X} \cap odvar_{\mathcal{D}}(\Pi_j) \neq \emptyset$ or else $\mathcal{X} \cap var(\Pi_j) = \emptyset$ (i.e., either some critical variable in $\mathcal{X}$ becomes obviously demanded, or else all critical variables in $\mathcal{X}$ disappear).
(5) **Soundness:** $Sol_{\mathcal{D}}(\Pi) \supseteq \bigcup_{j=1}^{k} Sol_{\mathcal{D}}(\exists \overline{Y}_j(\Pi_j \Box \sigma_j))$.
(6) **Completeness:** $WTSol_{\mathcal{D}}(\Pi) \subseteq \bigcup_{j=1}^{k} WTSol_{\mathcal{D}}(\exists \overline{Y}_j(\Pi_j \Box \sigma_j))$.

Moreover, $solve^{\mathcal{D}}$ is called an *extensible solver* iff the solver invocation $solve^{\mathcal{D}}(\Pi, \mathcal{X})$ is defined and satisfies the conditions listed in this definition not just for $\Pi \subseteq APCon_{\mathcal{D}}$ and $\mathcal{X} \subseteq cvar_{\mathcal{D}}(\Pi)$, but more generally for $\Pi \subseteq APCon_{\mathcal{D}'} \upharpoonright SPF$ and $\mathcal{X} \subseteq cvar_{\mathcal{D}'}(\Pi)$, where $\mathcal{D}'$ is any conservative extension of $\mathcal{D}$. The idea is that an extensible solver can deal with constraints involving the primitives in $\mathcal{D}$ and values described by patterns over arbitrary conservative extensions of $\mathcal{D}$.

The presentation of goal solving in Section 3 will discuss the proper way of choosing a set $\mathcal{X}$ of critical variables for each particular solver invocation. The idea is that $\mathcal{X}$ should include all critical variables which are waiting to be bound to the result of evaluating some expression at some other place within the goal. This idea also motivates the *safe bindings* condition.

Operationally, the alternatives within the disjunctions returned by solver invocations are usually explored in some sequential order with the help of a backtracking mechanism. Assuming that $solve^{\mathcal{D}}(\Pi, \mathcal{X}) = \bigvee_{j=1}^{k} \exists \overline{Y}_j(\Pi_j \Box \sigma_j)$, we will sometimes use the following notations:

- $\Pi \Vdash_{solve^{\mathcal{D}}_{\mathcal{X}}} \exists \overline{Y'}(\Pi' \Box \sigma')$ to indicate that $\exists \overline{Y'}(\Pi' \Box \sigma')$ is $\exists \overline{Y}_j(\Pi_j \Box \sigma_j)$ for some $1 \leqslant j \leqslant k$. In this case we will speak of a *successful solver invocation*.
- $\Pi \Vdash_{solve^{\mathcal{D}}_{\mathcal{X}}} \blacksquare$ to indicate that $k = 0$. In this case we will speak of a *failed solver invocation*, yielding the obviously unsatisfiable store $\blacksquare = \blacklozenge \Box \varepsilon$.

As defined above, a constraint store $\Pi \Box \sigma$ is said to be in *solved form* w.r.t. a set of critical variables $\mathcal{X}$ (or simply in solved form if $\mathcal{X} = \emptyset$) iff $solve^{\mathcal{D}}(\Pi, \mathcal{X}) = \Pi \Box \varepsilon$. In practice, solved forms can be recognized by syntactical criteria, and a solver invocation $solve^{\mathcal{D}}(\Pi, \mathcal{X})$ is performed only in the case that $\Pi \Box \sigma$ is not yet solved w.r.t. $\mathcal{X}$. Whenever a solver is invoked, the *soundness* condition requires that no new spurious solution (whether well-typed or not) is introduced, while the *completeness* condition requires that no *well-typed* solution is lost. In practice, any solver can be expected to be sound, but completeness may hold only for some choices of the constraint set Π to be solved. Demanding completeness for arbitrary (rather than well-typed) solutions would be still less realistic. The solvers of interest for this paper suffer some limitations regarding completeness, as explained in Subsections 2.4.2, 2.4.3, and 2.4.4.

From a user's viewpoint, a solver can behave as a *black-box* or as a *glass-box*. Black-box solvers can just be invoked to compute disjunctions of solved forms, but users cannot observe their inner workings, in contrast to the case of glass-box solvers. Users can define glass-box solvers by means of appropriate tools, such as *Constraint Handling Rules* (Frühwirth 1998). In this paper, we propose to use store transformation systems (*sts*s) as a convenient abstract technique for specifying the behavior of glass-box solvers. An *sts* over the constraint domain $\mathcal{D}$ is specified as a set of store transformation rules (*str*s) **RL** that describe different ways to transform a given store $\Pi \Box \sigma$ w.r.t. a given set $\mathcal{X}$ of critical variables. The notions and notations defined below are useful for working with *sts*s. Some of them refer to a selected set of *str*s noted as $\mathcal{RS}$.

- $\Pi \Box \sigma \Vdash_{\mathcal{D},\mathcal{X}} \Pi' \Box \sigma'$ indicates that the store $\Pi \Box \sigma$ can be transformed into $\Pi' \Box \sigma'$ in one step, using one of the available *str*s. This notation can also be used to indicate a failing transformation step, writing the inconsistent store $\blacksquare = \blacklozenge \Box \varepsilon$ in place of $\Pi' \Box \sigma'$.
- $\Pi \Box \sigma \Vdash^*_{\mathcal{D},\mathcal{X}} \Pi' \Box \sigma'$ indicates that $\Pi \Box \sigma$ can be transformed into $\Pi' \Box \sigma'$ in finitely many steps.
- The store $\Pi \Box \sigma$ is called *$\mathcal{RS}$-irreducible* iff there is no *str* **RL** $\in \mathcal{RS}$ that can be applied to transform $\Pi \Box \sigma$. Note that this is trivially true if $\mathcal{RS}$ is the empty set. If $\mathcal{RS}$ is the set of all the available *str*s, the store $\Pi \Box \sigma$ is called simply irreducible (or also a $\mathcal{X}$-solved form).
- $\Pi \Box \sigma \Vdash^*_{\mathcal{D},\mathcal{X}} ! \, \Pi' \Box \sigma'$ indicates that $\Pi \Box \sigma \Vdash^*_{\mathcal{D},\mathcal{X}} \Pi' \Box \sigma'$ holds, and moreover, the final store $\Pi' \Box \sigma'$ is irreducible.

Assume a given *sts* over $\mathcal{D}$ such that for any finite $\Pi \subseteq APCon_{\mathcal{D}}$ and any $\mathcal{X} \subseteq cvar_{\mathcal{D}}(\Pi)$, the set $\mathcal{SF}_{\mathcal{D}}(\Pi, \mathcal{X}) = \{\Pi' \Box \sigma' \mid \Pi \Box \varepsilon \Vdash^*_{\mathcal{D},\mathcal{X}} ! \, \Pi' \Box \sigma'\}$ is finite. Then, the solver defined by the *sts* can be specified to behave as follows:

$$solve^{\mathcal{D}}(\Pi, \mathcal{X}) = \bigvee \{\exists \overline{Y'}(\Pi' \Box \sigma') \mid \Pi' \Box \sigma' \in \mathcal{SF}_{\mathcal{D}}(\Pi, \mathcal{X}), \overline{Y'} = var(\Pi' \Box \sigma') \setminus var(\Pi)\}$$

Once $solve^{\mathcal{D}}$ has been so defined, the notation $\Pi \Vdash_{solve^{\mathcal{D}}_{\mathcal{X}}} \exists \overline{Y'}(\Pi' \Box \sigma')$ actually happens to mean that $\Pi \Box \varepsilon \Vdash^*_{\mathcal{D},\mathcal{X}} ! \, \Pi' \Box \sigma'$ and $\overline{Y'} = var(\Pi' \Box \sigma') \setminus var(\Pi)$. Therefore, the symbols $\Vdash_{solve^{\mathcal{D}}_{\mathcal{X}}}$ and $\Vdash^*_{\mathcal{D},\mathcal{X}}!$ should not be confused, but have related meanings. The following definition specifies different properties of *sts*s that are useful to check that the corresponding solvers satisfy the conditions stated in Definition 6.

Definition 7 (*Properties of sts*s)
Assume an *sts* over $\mathcal{D}$ whose transition relation is $\Vdash_{\mathcal{D},\mathcal{X}}$, and a selected set $\mathcal{RS}$ of *str*s. Then the *sts* is said to satisfy:

(1) The **Fresh Local Variables Property** iff $\Pi \Box \sigma \Vdash_{\mathcal{D},\mathcal{X}} \Pi' \Box \sigma'$ implies that $\Pi' \Box \sigma'$ is a store, $\overline{Y'} = var(\Pi' \Box \sigma') \setminus var(\Pi \Box \sigma)$ are fresh local variables, and $\sigma' = \sigma\sigma_1$

for some substitution σ_1 (responsible for the variable bindings created at this step) such that $vdom(\sigma_1) \cup vran(\sigma_1) \subseteq var(\Pi) \cup \overline{Y'}$.

(2) The **Safe Bindings Property** iff $\Pi \,\Box\, \sigma \Vdash_{\mathcal{D},\mathcal{X}} \Pi' \,\Box\, \sigma'$ implies that $\sigma_1(X)$ is a constant for all $X \in \mathcal{X} \cap vdom(\sigma_1)$, where $\sigma' = \sigma\sigma_1$ as in the previous item.

(3) The **Finitely Branching Property** iff for any fixed $\Pi \,\Box\, \sigma$ there are finitely many $\Pi' \,\Box\, \sigma'$ such that $\Pi \,\Box\, \sigma \Vdash_{\mathcal{D},\mathcal{X}} \Pi' \,\Box\, \sigma'$.

(4) The **Termination Property** iff there is no infinite sequence $\{\Pi_i \,\Box\, \sigma_i \mid i \in \mathbb{N}\}$ such that $\Pi_i \,\Box\, \sigma_i \Vdash_{\mathcal{D},\mathcal{X}} \Pi_{i+1} \,\Box\, \sigma_{i+1}$ for all $i \in \mathbb{N}$.

(5) The **Local Soundness Property** iff for any $\mathcal{D}$ store $\Pi \,\Box\, \sigma$, the union

$$\bigcup\{Sol_{\mathcal{D}}(\exists\overline{Y'}(\Pi' \,\Box\, \sigma')) \mid \Pi \,\Box\, \sigma \Vdash_{\mathcal{D},\mathcal{X}} \Pi' \,\Box\, \sigma', \overline{Y'} = var(\Pi' \,\Box\, \sigma') \setminus var(\Pi \,\Box\, \sigma)\}$$

is a subset of $Sol_{\mathcal{D}}(\Pi \,\Box\, \sigma)$.

(6) The **Local Completeness Property** for $\mathcal{RS}$-free steps iff for any $\mathcal{D}$ store $\Pi \,\Box\, \sigma$ which is $\mathcal{RS}$-irreducible but not in $\mathcal{X}$-solved form, $WTSol_{\mathcal{D}}(\Pi \,\Box\, \sigma)$ is a subset of the union

$$\bigcup\{WTSol_{\mathcal{D}}(\exists\overline{Y'}(\Pi' \,\Box\, \sigma')) \mid \Pi\Box\sigma \Vdash_{\mathcal{D},\mathcal{X}} \Pi'\Box\sigma', \overline{Y'} = var(\Pi'\Box\sigma') \setminus var(\Pi\Box\sigma)\}$$

If $\mathcal{RS}$ is the empty set (in which case all the stores are trivially $\mathcal{RS}$-irreducible) this property is called simply *local completeness.*

In the case of an extensible solver, the six conditions listed in this definition must be checked for any conservative extension $\mathcal{D}'$ of $\mathcal{D}$ and any set Π of SPF-restricted atomic primitive constraints over $\mathcal{D}'$.

Assume a solver $solve^{\mathcal{D}}$ defined by means of a given *sts* with transition relation $\Vdash_{\mathcal{D},\mathcal{X}}$ and a selected set $\mathcal{RS}$ of *strs*. If the *sts* is terminating, the following recursive definition makes sense: a given store $\Pi \,\Box\, \sigma$ is *hereditarily $\mathcal{RS}$-irreducible* iff $\Pi \,\Box\, \sigma$ is *$\mathcal{RS}$-irreducible* and all the stores $\Pi' \,\Box\, \sigma'$ such that $\Pi \,\Box\, \sigma \Vdash_{\mathcal{D},\mathcal{X}} \Pi' \,\Box\, \sigma'$ (if any) are also *hereditarily $\mathcal{RS}$-irreducible*. A solver invocation $solve^{\mathcal{D}}(\Pi, \mathcal{X})$ is called *$\mathcal{RS}$-free* iff the store $\Pi \,\Box\, \varepsilon$ is hereditarily $\mathcal{RS}$-irreducible. This notion occurs in the following technical lemma (proved in Appendix A.1), which can be applied to ensure that $solve^{\mathcal{D}}$ satisfies the requirements for solvers listed in Definition 6.

Lemma 5 (*Solvers Defined by Means of stss*)
Any finitely branching and terminating $\mathcal{D}$-*sts* verifies:

(1) $\mathcal{SF}_{\mathcal{D}}(\Pi, \mathcal{X})$ is always finite, and hence $solve^{\mathcal{D}}$ is well defined and trivially satisfies the solved forms property.

(2) $solve^{\mathcal{D}}$ has the fresh local variables resp. safe bindings property if the *sts* has the corresponding property.

(3) $solve^{\mathcal{D}}$ is sound if the *sts* is locally sound.

(4) $solve^{\mathcal{D}}$ is complete for $\mathcal{RS}$-free invocations if the *sts* is locally complete for $\mathcal{RS}$-free steps. In the case that $\mathcal{RS}$ is empty, this amounts to say that $solve^{\mathcal{D}}$ is complete if the *sts* is locally complete.

Note that this lemma can be used for proving global properties of extensible solvers, provided that the *sts* can work with constraint stores $\Pi \,\Box\, \sigma$, where Π is a finite set of SPF-restricted atomic primitive constraints over some arbitrary

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

conservative extension $\mathcal{D}'$ of $\mathcal{D}$, and the local properties required by the lemma hold for any such $\mathcal{D}'$.

In the rest of this paper, we will work with the three pure domains $\mathcal{H}$, $\mathcal{FD}$, and $\mathcal{R}$ introduced in the following sections. We will rely on black-box solvers for $\mathcal{R}$ and $\mathcal{FD}$ provided by SICStus Prolog, and we will define an extensible glass-box solver for $\mathcal{H}$ using the store transformation technique just explained.

2.4.2 The pure constraint domain $\mathcal{H}$

The *Herbrand domain* $\mathcal{H}$ supports computations with symbolic equality and disequality constraints over values of any type. Formally, it is defined as follows:

- Specific signature $\Sigma = \langle TC, SBT, DC, DF, SPF \rangle$, where SBT is empty and SPF includes just the strict equality operator `== :: A -> A -> bool`.
- Interpretation $==^{\mathcal{H}}$, defined as for any domain whose specific signature includes ==.

Recall Definition 2 and note that a conservative extension of $\mathcal{H}$ is any domain $\mathcal{D}$ whose specific signature includes the primitive ==. Such a $\mathcal{D}$ will be called a *domain with equality* in the sequel. The $\{==\}$-restricted constraints over a given domain with equality are also called *extended* Herbrand constraints. As already explained in Subsection 2.3, atomic Herbrand constraints have the form $e_1 == e_2 \to! t$, *strict equality constraints* $e_1 == e_2$ abbreviate $e_1 == e_2 \to!$ `true`, and *strict disequality constraints* abbreviate $e_1 == e_2 \to!$ `false`.

Obviously demanded variables (and thus critical variables) for primitive extended Herbrand constraints are computed as explained in Subsection 2.3. An extensible Herbrand solver must be able to solve any finite set $\Pi \subseteq APCon_{\mathcal{D}} \upharpoonright \{==\}$ of atomic primitive extended Herbrand constraints, w.r.t. any $\mathcal{X} \subseteq cvar_{\mathcal{D}}(\Pi)$ of critical variables. Roughly speaking, the solver proceeds by symbolic decomposition and binding propagation transformations. More precisely, we define an extensible glass-box solver for $\mathcal{H}$ by means of the store transformation technique explained in Subsection 2.4.1, using the transformation rules for $\mathcal{H}$ stores shown in Table 1. Each of these rules has the form $\pi, \Pi \,\Box\, \sigma \Vdash_{\mathcal{H},\mathcal{X}} \Pi' \,\Box\, \sigma'$ and indicates the transformation of any store $\pi, \Pi \,\Box\, \sigma$, which includes the atomic constraint π plus other constraints Π; no sequential ordering is intended. We say that π is the *selected atomic constraint* for this transformation step. The notation $\overline{t_m{==}s_m}$ in transformation **H3** abbreviates $t_1 == s_1, \ldots, t_m == s_m$ and will be used at some other places. All the *stss* make sense for arbitrary extended Herbrand constraints, which ensures extensibility of the $\mathcal{H}$ solver. Note that transformations **H3** and **H7** involve decompositions. An application of **H3** or **H7** is called *opaque* iff h is m-opaque in the sense explained in Subsection 2.2, in which case the new constraints resulting from the decomposition may become ill-typed. Note also that an application of transformation **H13** may obviously lose solutions. An invocation $solve^{\mathcal{H}}(\Pi, \mathcal{X})$ of the $\mathcal{H}$ solver is called *safe* iff it has been computed without any opaque application of the store transformation

Table 1. *Store transformations for $solve^{\mathcal{H}}$*

H1 $(t == s) \to! R,\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{H},\mathcal{X}} (t == s,\ \Pi)\sigma_1 \,\Box\, \sigma\sigma_1$ where $\sigma_1 = \{R \mapsto true\}$.
H2 $(t == s) \to! R,\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{H},\mathcal{X}} (t \mathrel{/=} s,\ \Pi)\sigma_1 \,\Box\, \sigma\sigma_1$ where $\sigma_1 = \{R \mapsto false\}$.

H3 $h\,\overline{t}_m == h\,\overline{s}_m,\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{H},\mathcal{X}} \overline{t_m == s_m},\ \Pi \,\Box\, \sigma$

H4 $t == X,\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{H},\mathcal{X}} X == t,\ \Pi \,\Box\, \sigma$ if t is not a variable.

H5 $X == t,\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{H},\mathcal{X}} tot(t),\ \Pi\sigma_1 \,\Box\, \sigma\sigma_1$ if $X \notin \mathcal{X}$, $X \notin var(t)$, $X \neq t$, where $\sigma_1 = \{X \mapsto t\}$, $tot(t)$ abbreviates $\bigwedge_{Y \in var(t)}(Y == Y)$.

H6 $X == t,\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{H},\mathcal{X}} \blacksquare$ if $X \in var(t)$, $X \neq t$.

H7 $h\,\overline{t}_m \mathrel{/=} h\,\overline{s}_m,\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{H},\mathcal{X}} (t_i \mathrel{/=} s_i,\ \Pi \,\Box\, \sigma)$ for each $1 \leqslant i \leqslant m$.

H8 $h\,\overline{t}_n \mathrel{/=} h'\,\overline{s}_m,\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{H},\mathcal{X}} \Pi \,\Box\, \sigma$ if $h \neq h'$ or $n \neq m$.

H9 $t \mathrel{/=} t,\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{H},\mathcal{X}} \blacksquare$ if $t \in \mathcal{V}ar \cup DC \cup DF \cup SPF$.

H10 $t \mathrel{/=} X,\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{H},\mathcal{X}} X \mathrel{/=} t,\ \Pi \,\Box\, \sigma$ if t is not a variable.

H11 $X \mathrel{/=} c\,\overline{t}_n, \Pi\Box\sigma \Vdash_{\mathcal{H},\mathcal{X}} (Z_i \mathrel{/=} t_i, \Pi)\sigma_1\Box\sigma\sigma_1$ if $X \notin \mathcal{X}$, $c \in DC^n$ and $\mathcal{X} \cap var(c\,\overline{t}_n) \neq \emptyset$ where $1 \leqslant i \leqslant n$ (nondeterministic choice), $\sigma_1 = \{X \mapsto c\,\overline{Z}_n\}$, $\overline{Z}_n$ fresh variables.

H12 $X \mathrel{/=} c\,\overline{t}_n,\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{H},\mathcal{X}} \Pi\sigma_1 \,\Box\, \sigma\sigma_1$ if $X \notin \mathcal{X}$, $c \in DC^n$ and $\mathcal{X} \cap var(c\,\overline{t}_n) \neq \emptyset$ where $\sigma_1 = \{X \mapsto d\,\overline{Z}_m\}$, $c \in DC^n$, $d \in DC^m$, $d \neq c$, d belongs to the same datatype as c, $\overline{Z}_m$ fresh variables.

H13 $X \mathrel{/=} h\,\overline{t}_m,\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{H},\mathcal{X}} \blacksquare$ if $X \notin \mathcal{X}$, $\mathcal{X} \cap var(h\,\overline{t}_m) \neq \emptyset$ and $h \notin DC^m$.

rules **H3** and **H7** and without any application of the store transformation rule **H13**. More formally, $solve^{\mathcal{H}}(\Pi, \mathcal{X})$ is a safe invocation of the $\mathcal{H}$ solver iff it is $\mathcal{URS}$-free, where $\mathcal{URS}$ is the set $\{$**OH3**, **OH7**, **H13**$\}$ consisting of **H13** and the unsafe instances **OH3** and **OH7** corresponding to opaque applications of **H3** and **H7**, respectively.

The idea of using equality and disequality constraints in Logic Programming stems from Colmerauer (1984, 1990). The problem of solving these constraints, as well as related decision problems for theories involving equations and disequations, has been widely investigated in works such as Lassez *et al.* (1988), Maher (1988), Comon and Lescanne (1989), Comon (1991), Fernandez (1992), and Buntine and Bürckert (1994), among others. These papers assume the classical algebraic semantics for the equality relation, and propose methods for solving so-called *unification and disunification problems* bearing some analogies to the transformation rules shown in Table 1. However, there are also some differences, because strict equality in *CFLP* is designed to work with lazy and possibly nondeterministic functions, whose behavior does not correspond to the semantics of equality in classical algebra and equational logic, as argued in Rodríguez-Artalejo (2001). Note, in particular, transformation **H5**, which introduces constraints of the form $Y == Y$ in $\mathcal{H}$-solved forms. These are called *totality constraints*, because a valuation η is a solution of $Y == Y$ iff $\eta(Y)$ is a total pattern. An approach to disequality constraints close to our semantic framework can be found in Arenas *et al.* (1994), but no formalization of a Herbrand solver is provided.

The following theorem ensures that the *sts* for $\mathscr{H}$ stores can be accepted as a correct specification of an extensible glass-box solver for the domain $\mathscr{H}$, which is complete for safe solver invocations.

Theorem 1 (*Formal Properties of solve*$^{\mathscr{H}}$)
The *sts* with transition relation $\Vdash_{\mathscr{H},\mathscr{X}}$ is finitely branching and terminating, and therefore

$$solve^{\mathscr{H}}(\Pi, \mathscr{X}) = \bigvee\{\exists \overline{Y'}(\Pi' \Box \sigma') \mid \Pi' \Box \sigma' \in \mathscr{SF}_{\mathscr{H}}(\Pi, \mathscr{X}),\ \overline{Y'} = var(\Pi' \Box \sigma') \backslash var(\Pi)\}$$

is well defined for any domain with equality $\mathscr{D}$, any finite $\Pi \subseteq APCon(\mathscr{D}) \upharpoonright \{==\}$ and any $\mathscr{X} \subseteq cvar_{\mathscr{D}}(\Pi)$. Moreover, for any arbitrary choice of a domain $\mathscr{D}$ with equality, $solve^{\mathscr{H}}$ satisfies all the requirements for solvers enumerated in Definition 6, except that the *completeness* property may fail for some choices of the constraint set $\Pi \subseteq APCon(\mathscr{D}) \upharpoonright \{==\}$ to be solved, and is guaranteed to hold only if the solver invocation $solve^{\mathscr{H}}(\Pi, \mathscr{X})$ is safe (i.e., $\{$**OH3**, **OH7**, **H13**$\}$-free).

The proof of the previous theorem is rather technical and can be found in Appendix A.1. At this point, we just make a few remarks related to the discrimination and completeness properties that may help to understand some differences between our $\mathscr{H}$ solver and more classical methods for solving unification and disunification problems. On the one hand, transformations **H11** and **H12** are designed to ensure the discrimination property while preserving completeness w.r.t. well-typed solutions. On the other hand, transformation **H13** trivially ensures discrimination, but it sacrifices completeness because it fails without making sure that no well-typed solutions exist. This corresponds to situations unlikely to occur in practice and such that no practical way of preserving completeness is at hand. The other two failing transformations given in Table 1 (namely **H6** and **H9**) respect completeness, because they are applied to unsatisfiable stores. Finally, the other cases where completeness may be lost correspond to unsafe decomposition steps performed with the opaque instances **OH3** and **OH7** of the *str*s **H3** and **H7**. Because of the termination property of the $\mathscr{H}$-*sts*, it is decidable wether a given $\mathscr{H}$ store $\Pi \Box \sigma$ is hereditarily $\mathscr{URS}$-irreducible, in which case no opaque decompositions will occur when solving the store. However, computations in the cooperative goal-solving calculus presented in Section 3 can sometimes give rise to $\mathscr{H}$ stores whose resolution involves opaque decomposition steps. Because of theoretical results proved in González-Moreno *et al.* (2001), the eventual occurrence of opaque decomposition steps during goal solving is an undecidable problem. In case that opaque decompositions occur, they should be signaled as warnings to the user.

Example 4 (*Behavior of solve*$^{\mathscr{H}}$)
In order to illustrate the behavior of $solve^{\mathscr{H}}$, consider the disequality constraint `L /= X:Xs` discussed in item (4) of Example 3. Remember that variable `L` is obviously demanded, while variables `X` and `Xs` are both critical. Therefore, there are four possible choices for the set $\mathscr{X}$ of critical variables to be used within the solver invocation, namely: $\emptyset$, {`X`}, {`Xs`} and {`X, Xs`}. Let us discuss these cases one by one.

- Choosing $\mathcal{X} = \emptyset$ means that the solver is not asked to discriminate w.r.t. any critical variable. In this case, $solve^{\mathcal{H}}$(`L/=X:Xs`,$\emptyset$) returns `L/=X:Xs` $\Box\,\varepsilon$, showing that `L/=X:Xs` is seen as a solved form w.r.t. the empty set of critical variables.
- Choosing $\mathcal{X} = \{$`X`$\}$ asks the solver to discriminate w.r.t. the critical variable `X`. $solve^{\mathcal{H}}$(`L/=X:Xs`,$\{$`X`$\}$) returns a disjunction of alternatives

$$(\Diamond \,\Box \{\mathtt{L} \mapsto [\,]\}) \vee (\mathtt{X'/=\ X} \,\Box \{\mathtt{L} \mapsto \mathtt{X'} : \mathtt{Xs'}\}) \vee (\mathtt{Xs'/=\ Xs} \,\Box \{\mathtt{L} \mapsto \mathtt{X'} : \mathtt{Xs'}\})$$

  whose members correspond to the three different stores $\Pi' \Box \sigma'$ such that the step `L/=X:Xs` $\Box\,\varepsilon \Vdash_{\mathcal{H},\{\mathtt{X}\}} \Pi' \Box \sigma'$ can be performed with transformation **H12**. Since these stores are solved w.r.t. $\{$`X`$\}$, no further transformations are required. Note that `X` does not occur in the first and third alternatives, while it has become obviously demanded in the second one. In this way, the discrimination property required for solvers is fulfilled.
- For each of the two choices $\mathcal{X} = \{$`Xs`$\}$ and $\mathcal{X} = \{$`X`, `Xs`$\}$, it is easy to check that the solver invocation $solve^{\mathcal{H}}$(`L/=X:Xs`$, \mathcal{X}$) returns the same disjunction of three alternatives as in the previous item, and the discrimination property is also fulfilled w.r.t. the chosen set $\mathcal{X}$ in both cases.

2.4.3 The pure constraint domain $\mathcal{R}$

The $\mathcal{R}$ domain supporting computation with arithmetic constraints over real numbers is a familiar idea, used in the well-known instance $CLP(\mathcal{R})$ of the CLP scheme (Jaffar *et al.* 1992). In the context of our $CFLP$ framework, a convenient formal definition of the domain $\mathcal{R}$ is as follows:

- Specific signature $\Sigma = \langle TC,\ SBT,\ DC,\ DF,\ SPF\rangle$, where $SBT = \{$`real`$\}$ includes just one base type whose values represent real numbers, and SPF includes the following binary primitive symbols, all of them intended to be used in infix notation:
  - — The strict equality operator `== :: A -> A -> bool`.
  - — The arithmetical operators `+, -, *, / :: real -> real -> real`.
  - — The inequality operator `<= :: real -> real -> bool`.
- Set of base values $\mathcal{B}^{\mathcal{R}}_{\mathtt{real}} = \mathbb{R}$.
- Interpretation $\mathtt{==}^{\mathcal{R}}$, defined as for any domain whose specific signature includes `==`.
- Interpretation $\mathtt{+}^{\mathcal{R}}$, defined so that for all $t_1, t_2, t \in \mathcal{U}_{\mathcal{R}}$:
  $t_1\ \mathtt{+}^{\mathcal{R}} t_2 \to t$ is defined to hold iff some of the following cases hold:
  either t_1, t_2, and t are real numbers, t being equal to the addition of t_1 and t_2, or else $t = \bot$. The interpretations of `-, *,` and `/` are defined analogously.
- Interpretation $\mathtt{<=}^{\mathcal{R}}$, defined so that for all $t_1, t_2, t \in \mathcal{U}_{\mathcal{R}}$:
  $t_1\ \mathtt{<=}^{\mathcal{R}} t_2 \to t$ is defined to hold iff some of the following cases hold:
  either t_1, t_2 are real numbers such that t_1 is less than or equal to t_2, and $t =$ `true`; or else t_1, t_2 are real numbers such that t_1 is greater than t_2, and $t =$ `false`; or else $t = \bot$.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Atomic $\mathcal{R}$ constraints have the form $e_1 \odot e_2 \rightarrow! t$, where $\odot$ is the strict equality operator or the inequality operator or an arithmetical operator. An atomic $\mathcal{R}$ constraint is called *proper* iff $\odot$ is not the strict equality operator, and an extended Herbrand constraint otherwise. As already explained in previous sections, *strict equality constraints* e_1 `==` e_2 and *strict disequality constraints* e_1 `/=` e_2 can be understood as abbreviations of extended Herbrand constraints. Moreover, various kinds of *inequality constraints* can also be defined as abbreviations as follows:

- e_1 `<` $e_2 =_{def} e_2$ `<=` $e_1 \rightarrow!$ `false` e_1 `<=` $e_2 =_{def} e_1$ `<=` $e_2 \rightarrow!$ `true`
- e_1 `>` $e_2 =_{def} e_1$ `<=` $e_2 \rightarrow!$ `false` e_1 `>=` $e_2 =_{def} e_2$ `<=` $e_1 \rightarrow!$ `true`

Concerning the solver $solve^{\mathcal{R}}$, we expect that it is able to deal with $\mathcal{R}$-specific constraint sets $\Pi \subseteq APCon_{\mathcal{R}}$ consisting of atomic primitive constraints π of the following two kinds:

- Proper $\mathcal{R}$ constraints $t_1 \odot t_2 \rightarrow! t$, where $\odot$ is either the inequality operator or an arithmetical operator.
- $\mathcal{R}$-specific Herbrand constraints having the form t_1 `==` t_2 or t_1 `/=` t_2, where each of the two patterns t_1 and t_2 is either a real constant value or a variable whose type is known to be `real` prior to the solver invocation.

For any finite $\mathcal{R}$-specific $\Pi \subseteq APCon_{\mathcal{R}}$, it is clear that $dvar_{\mathcal{R}}(\Pi) = var(\Pi)$. Therefore, it is safe to define $odvar_{\mathcal{R}}(\Pi) = var(\Pi)$ and thus $cvar_{\mathcal{R}}(\Pi) = \emptyset$. Consequently, invocations to $solve^{\mathcal{R}}$ can be assumed to be always of the form $solve^{\mathcal{R}}(\Pi, \emptyset)$ (abbreviated as $solve^{\mathcal{R}}(\Pi)$), and the discrimination requirements for critical variables become trivial. Assuming that $solve^{\mathcal{R}}$ is used under the restrictions described above and implemented as a black-box solver on top of SICStus Prolog, we are confident that the postulate stated below is a reasonable one. In particular, we assume that SICStus Prolog solves $\mathcal{R}$-specific Herbrand constraints in a way compatible with the behavior of the extensible $\mathcal{H}$ solver described in the previous subsection.

Postulate 1 (*Assumptions on the $\mathcal{R}$ Solver*)
$solve^{\mathcal{R}}$ satisfies five of the six properties required for solvers in Definition 6 (namely *Fresh Local Variables*, *Solved Forms*, *Safe Bindings*, *Discrimination*, and *Soundness*), although the *Completeness* property may fail for some choices of the $\mathcal{R}$-specific $\Pi \subseteq APCon_{\mathcal{R}}$ to be solved. Moreover, whenever $\Pi \subseteq APCon_{\mathcal{R}}$ is $\mathcal{R}$-specific and $\Pi \Vdash_{solve^{\mathcal{R}}} \exists \overline{Y'}(\Pi' \Box \sigma')$, the constraint set Π' is also $\mathcal{R}$-specific, and for all $X \in vdom(\sigma')$: either $\sigma'(X)$ is a real value, or else X and $\sigma'(X)$ belong to $var(\Pi)$.

Example 5 (*Behavior of the $\mathcal{R}$ Solver*)
Let us now illustrate the behavior of $solve^{\mathcal{R}}$ by considering the set of primitive atomic constraints `RY >= d-0.5, RY-RX <= 0.5, RY+RX <= n+0.5` occurring in item (2) of Example 3. The solver invocation $solve^{\mathcal{R}}(\Pi_R)$ returns one single alternative $\Pi'_R \Box \varepsilon$ with $\Pi'_R = \Pi_R \cup \{$`RY <= d+0.5`$\}$. In this case, the new constraint `RY <= d+0.5` has been inferred by adding the two former constraints `RY-RX <= 0.5` and `RY+RX <= n+0.5` and taking into account that `n = 2*d`. In other cases, the $\mathcal{R}$ solver can perform other inferences by means of arithmetical reasoning valid in the mathematical theory

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

of the real number field. In general, solved forms computed by solvers help to make more explicit the requirements on variable values already "hidden" in the constraints prior to solving (as the upper bound `RY <= d+0.5` in this example).

2.4.4 *The pure constraint domain $\mathcal{FD}$*

The idea of an $\mathcal{FD}$ domain supporting computation with arithmetic and finite domain constraints over the integers is a familiar one within the *CLP* community (see, e.g., van Hentenryck *et al.* 1994, 1998). In the context of our *CFLP* framework, a convenient formal definition of the domain $\mathcal{FD}$ is as follows:

- Specific signature $\Sigma = \langle TC, SBT, DC, DF, SPF \rangle$, where $SBT = \{\texttt{int}\}$ includes just one base type whose values represent integer numbers, and SPF includes the following primitive symbols:
  - — The strict equality operator `== :: A -> A -> bool`.
  - — The arithmetical operators `#+, #-, #*, #/ :: int -> int -> int`.
  - — The following primitive symbols related to computation with finite domains:
    - `domain:: [int] -> int -> int -> bool`
    - `belongs:: int -> [int] -> bool`
    - `labeling:: [labelType] -> [int] -> bool`, where `labelType` is an enumerated datatype used to represent labeling strategies.
  - — The inequality operator `#<= :: int -> int -> bool`.
- Set of base values $\mathcal{B}^{\mathcal{FD}}_{\texttt{int}} = \mathbb{Z}$.
- Interpretation $==^{\mathcal{FD}}$, defined as for any domain whose specific signature includes ==.
- Interpretation $\texttt{#+}^{\mathcal{FD}}$, defined so that for all $t_1, t_2, t \in \mathcal{U}_{\mathcal{FD}}$: t_1 $\texttt{#+}^{\mathcal{FD}} t_2 \to t$ is defined to hold iff some of the following cases hold: either t_1, t_2, and t are integer numbers, t being equal to the addition of t_1 and t_2, or else $t = \bot$. The interpretations of `#-`, `#*`, and `#/` are defined analogously.
- Interpretation $\texttt{domain}^{\mathcal{FD}}$, defined so that for all $t_1, t_2, t_3, t \in \mathcal{U}_{\mathcal{FD}}$: $\texttt{domain}^{\mathcal{FD}}\ t_1\, t_2\, t_3 \to t$ is defined to hold iff some of the following cases hold: either t_2 and t_3 are integer numbers a and b such that $a \leqslant b$, t_1 is a nonempty finite list of integers belonging to the interval $a..b$ and $t = \texttt{true}$; or else t_2 and t_3 are integer numbers a and b such that $a \leqslant b$, t_1 is a nonempty list of integers some of which do not belong to the interval $a..b$ and $t = \texttt{false}$; or else t_2 and t_3 are integer numbers a and b such that $a > b$ and $t = \texttt{false}$; or else $t = \bot$.
- Interpretation $\texttt{belongs}^{\mathcal{FD}}$, defined so that for all $t_1, t_2, t \in \mathcal{U}_{\mathcal{FD}}$: $\texttt{belongs}^{\mathcal{FD}}\ t_1\, t_2 \to t$ is defined to hold iff some of the following cases hold: either t_1 is an integer, t_2 is a finite list of integers including t_1 as element, and $t = \texttt{true}$; or else t_1 is an integer, t_2 is a finite list of integers not including t_1 as element, and $t = \texttt{false}$; or else $t = \bot$.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

- Interpretation $\texttt{labeling}^{\mathscr{F}\mathscr{D}}$, defined so that for all $t_1, t_2, t \in \mathscr{U}_{\mathscr{F}\mathscr{D}}$:
  $\texttt{labeling}^{\mathscr{F}\mathscr{D}}\ t_1\, t_2 \rightarrow t$ is defined to hold iff some of the following cases hold: either t_1 is a defined value of type `labelType`, t_2 is a finite list of integers, and $t = \texttt{true}$; or else $t = \bot$.
- Interpretation $\texttt{\#<=}^{\mathscr{F}\mathscr{D}}$, defined so that for all $t_1, t_2, t \in \mathscr{U}_{\mathscr{F}\mathscr{D}}$:
  $t_1\ \texttt{\#<=}^{\mathscr{F}\mathscr{D}} t_2 \rightarrow t$ is defined to hold iff some of the following cases hold: either t_1, t_2 are integer numbers such that t_1 is less than or equal to t_2, and $t = \texttt{true}$; or else t_1, t_2 are integer numbers such that t_1 is greater than t_2, and $t = \texttt{false}$; or else $t = \bot$.

Atomic $\mathscr{F}\mathscr{D}$ constraints include those of the form $e_1 \odot e_2 \rightarrow! t$, where $\odot$ is either the strict equality operator or the inequality operator or an arithmetical operator, as well as *domain constraints* `domain` $e_1\, e_2\, e_3 \rightarrow! t$, *membership constraints* `belongs` $e_1\, e_2 \rightarrow! t$ and *labeling constraints* `labeling` $e_1\, e_2 \rightarrow! t$. Atomic $\mathscr{F}\mathscr{D}$ constraints are called extended Herbrand if they have the form $e_1 == e_2 \rightarrow! t$, and *proper* $\mathscr{F}\mathscr{D}$ constraints otherwise. As already explained in previous sections, *strict equality constraints* e_1 `==` e_2 and *strict disequality constraints* e_1 `/=` e_2 can be understood as abbreviations of extended Herbrand constraints. Moreover, various kinds of *inequality constraints* can also be defined as abbreviations as follows:

- e_1 `#<` $e_2 =_{def} e_2$ `#<=` $e_1 \rightarrow!$ `false` e_1 `#<=` $e_2 =_{def} e_1$ `#<=` $e_2 \rightarrow!$ `true`
- e_1 `#>` $e_2 =_{def} e_1$ `#<=` $e_2 \rightarrow!$ `false` e_1 `#>=` $e_2 =_{def} e_2$ `#<=` $e_1 \rightarrow!$ `true`

Concerning the solver $solve^{\mathscr{F}\mathscr{D}}$, we expect that it is able to deal with $\mathscr{F}\mathscr{D}$-specific constraint sets $\Pi \subseteq APCon_{\mathscr{F}\mathscr{D}}$ consisting of atomic primitive constraints π of the following two kinds:

- Proper $\mathscr{F}\mathscr{D}$ atomic primitive constraints (which may be $t_1 \odot t_2 \rightarrow! t$, where $\odot$ is either an integer arithmetical primitive or an inequality primitive, or primitive domain, membership, and labeling constraints).
- $\mathscr{F}\mathscr{D}$-specific Herbrand constraints having the form t_1 `==` t_2 or t_1 `/=` t_2, where each of the two patterns t_1 and t_2 is either an integer constant value or a variable whose type is known to be `int` prior to the solver invocation.

For any finite $\mathscr{F}\mathscr{D}$-specific $\Pi \subseteq APCon_{\mathscr{F}\mathscr{D}}$, it is clear that $dvar_{\mathscr{F}\mathscr{D}}(\Pi) = var(\Pi)$. Therefore, it is safe to define $odvar_{\mathscr{F}\mathscr{D}}(\Pi) = var(\Pi)$ and thus $cvar_{\mathscr{F}\mathscr{D}}(\Pi) = \emptyset$. Consequently, invocations to $solve^{\mathscr{F}\mathscr{D}}$ can be assumed to be always of the form $solve^{\mathscr{F}\mathscr{D}}(\Pi, \emptyset)$ (abbreviated as $solve^{\mathscr{F}\mathscr{D}}(\Pi)$), and the discrimination requirements for critical variables become trivial. Assuming that $solve^{\mathscr{F}\mathscr{D}}$ is used under the restrictions described above and implemented as a black-box solver on top of SICStus Prolog, we are confident that the postulate stated below is a reasonable one. In particular, we assume that SICStus Prolog solves $\mathscr{F}\mathscr{D}$-specific Herbrand constraints in a way compatible with the behavior of the extensible $\mathscr{H}$ solver described in the previous subsection.

Postulate 2 (*Assumptions on the $\mathscr{F}\mathscr{D}$ Solver*)
$solve^{\mathscr{F}\mathscr{D}}$ satisfies five of the six properties required for solvers in Definition 6 (namely *Fresh Local Variables*, *Solved Forms*, *Safe Bindings*, *Discrimination*, and *Soundness*),

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

although the *Completeness* property may fail for some choices of the $\mathscr{FD}$-specific $\Pi \subseteq APCon_{\mathscr{FD}}$ to be solved. Moreover, whenever $\Pi \subseteq APCon_{\mathscr{FD}}$ is $\mathscr{FD}$-specific and $\Pi \Vdash_{solve^{\mathscr{FD}}} \exists \overline{Y'}(\Pi' \Box \sigma')$, the constraint set Π' is also $\mathscr{FD}$-specific, and for all $X \in vdom(\sigma')$: either $\sigma'(X)$ is an integer value, or else X and $\sigma'(X)$ belong to $var(\Pi)$.

In particular, labeling constraints are solved by a systematic enumeration of possible values for certain integer variables. Therefore, $solve^{\mathscr{FD}}$ is unable to solve a labeling constraint π unless the current constraint store includes domain or membership constraints for all the variables occurring in π. The next example shows a typical situation.

Example 6 (*Behavior of the $\mathscr{FD}$ Solver*)
In order to illustrate the behavior of $solve^{\mathscr{FD}}$, let us consider the set of primitive atomic constraints $\Pi_F = \{$`domain [X,Y] 0 n, labeling [] [X,Y]`$\}$ occurring also in item (2) of Example 3. The solver invocation $solve^{\mathscr{FD}}(\Pi_F)$ must solve the conjunction of a domain constraint and a labeling constraint, both involving the integer variables `X` and `Y`. The solver proceeds by enumerating all the possible values of both variables `X` and `Y` within their respective domains (determined in this case by the domain constraint `domain [X,Y] 0 n`) and returns a disjunction of $(\mathtt{n+1})^2$ alternatives, each of which describes one single solution:

$$(\Diamond \Box \{\mathtt{X} \mapsto 0,\ \mathtt{Y} \mapsto 0\}) \vee \cdots \vee (\Diamond \Box \{\mathtt{X} \mapsto n,\ \mathtt{Y} \mapsto n\})$$

In general, solving labeling constraints can give rise to very expensive enumerations of solutions, unless the finite domains of the integer variables involved have been pruned by some precedent computation. As already discussed in Subsection 1.2, the efficiency of solving the constraint system occurring in item (2) of Example 3 can be greatly improved by cooperation among the the domains $\mathscr{H}$, $\mathscr{R}$, and $\mathscr{FD}$. We propose to use the *coordination domains* defined in the next subsection as a vehicle for domain cooperation in *CFLP* programming.

2.5 Coordination domains

Coordination domains $\mathscr{C}$ are a kind of "hybrid" domains built from various component domains $\mathscr{D}_i$, intended to cooperate. The construction of coordination domains also involves a so-called *mediatorial domain* $\mathscr{M}$, whose purpose is to supply *bridge constraints* for communication among the component domains. In practice, the component domains will be chosen as pure domains equipped with solvers, and the communication provided by the mediatorial domain will also benefit the solvers.

Mathematically, the construction of coordination domains relies on a joinability condition. Two given constraint domains $\mathscr{D}_1$ and $\mathscr{D}_2$ with specific signatures $\Sigma_1 = \langle TC, SBT_1, DC, DF, SPF_1 \rangle$ and $\Sigma_2 = \langle TC, SBT_2, DC, DF, SPF_2 \rangle$, respectively, are called *joinable* iff the following two conditions hold:

- $SPF_1 \cap SPF_2 \subseteq \{==\}$; i.e., the only primitive function symbol p allowed to belong both to SPF_1 and SPF_2 is the strict equality operator `==`.
- For every common base type $d \in SBT_1 \cap SBT_2$, one has $\mathscr{B}_d^{\mathscr{D}_1} = \mathscr{B}_d^{\mathscr{D}_2}$.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

The *amalgamated sum* $\mathscr{S} = \mathscr{D}_1 \oplus \mathscr{D}_2$ of two joinable domains $\mathscr{D}_1$ and $\mathscr{D}_2$ is defined as a new domain with signature $\Sigma = \langle TC,\ SBT_1 \cup SBT_2,\ DC,\ DF,\ SPF_1 \cup SPF_2 \rangle$, constructed as follows:

- For $i = 1, 2$ and for all $d \in SBT_i$: $\mathscr{B}_d^{\mathscr{S}} = \mathscr{B}_d^{\mathscr{D}_i}$.
  (no conflict will arise for those $d \in SBT_1 \cap SBT_2$, because of joinability)
- For $i = 1, 2$, for all $p \in SPF_i$, p other than ==, and for all $\overline{t}_n$, $t \in \mathscr{U}_{\mathscr{S}}$:
  $p^{\mathscr{S}}\overline{t}_n \rightarrow t$ is defined to hold iff one of the following two cases holds:
  either $t = \perp$ or else there exist $\overline{t'}_n$, $t' \in \mathscr{U}_{\mathscr{D}_i}$ such that $\overline{t'}_n \sqsubseteq \overline{t}_n$, $t' \sqsupseteq t$ and $p^{\mathscr{D}_i}\overline{t'}_n \rightarrow t'$.

Note that the value universe $\mathscr{U}_{\mathscr{S}}$ underlying an amalgamated sum $\mathscr{S} = \mathscr{D}_1 \oplus \mathscr{D}_2$ is a superset of $\mathscr{U}_{\mathscr{D}_i}$ for $i = 1, 2$. The interpretation of == in $\mathscr{S}$ will behave as defined for any constraint domain (see Subsection 2.3). For primitive functions $p \in SPF_i$ other than ==, the definition of $p^{\mathscr{S}}$ is designed to obtain an extension of $p^{\mathscr{D}_i}$ which satisfies the technical conditions required by Definition 1.

The amalgamated sum $\mathscr{D}_1 \oplus \cdots \oplus \mathscr{D}_n$ of n pairwise joinable domains $\mathscr{D}_i$ ($1 \leqslant i \leqslant n$) can be defined analogously. The following definition and theorem guarantee the expected behavior of amalgamated sums as conservative extensions of their components. The proof of the theorem is given in Appendix A.1.

Definition 8 (*Domain-specific Constraints and Truncation Operator*)
Assume $\mathscr{S} = \mathscr{D}_1 \oplus \cdots \oplus \mathscr{D}_n$ of signature Σ, constructed as the amalgamated sum of n pairwise joinable domains $\mathscr{D}_i$ of signatures Σ_i. Let any $1 \leqslant i \leqslant n$ be arbitrarily fixed.

(1) A set $\Pi \subseteq APCon_{\mathscr{D}_i}$ is called $\mathscr{D}_i$*-specific* iff every valuation $\eta \in Val_{\mathscr{S}}$ such that $\eta \in Sol_{\mathscr{S}}(\Pi)$ satisfies $\eta(X) \in \mathscr{U}_{\mathscr{D}_i}$ for all $X \in var(\Pi)$. Note that the $\mathscr{R}$- and $\mathscr{F}\mathscr{D}$-specific sets of constraints previously introduced in subsections 2.4.3 and 2.4.4 are also specific in the sense just defined.

(2) Consider the information ordering $\sqsubseteq$ over $\mathscr{U}_{\mathscr{S}}$. The $\mathscr{D}_i$*-truncation* of a given $\mathscr{S}$ value $t \in \mathscr{U}_{\mathscr{S}}$ is defined as the $\sqsubseteq$-greatest $\mathscr{D}_i$ value $\mid t \mid_{\mathscr{D}_i} \in \mathscr{U}_{\mathscr{D}_i}$ which satisfies $\mid t \mid_{\mathscr{D}_i} \sqsubseteq t$, so that any other $\mathscr{D}_i$ value $\hat{t} \in \mathscr{U}_{\mathscr{D}_i}$ such that $\hat{t} \sqsubseteq t$ must satisfy $\hat{t} \sqsubseteq \mid t \mid_{\mathscr{D}_i}$. An effective construction of $\mid t \mid_{\mathscr{D}_i}$ from t can be obtained by substituting $\perp$ in place of any subpattern of t which has any of the following two forms: a basic value u which does not belong to $\mathscr{U}_{\mathscr{D}_i}$, or a partial application of a primitive function which does not belong to $\mathscr{D}_i$-specific signature. Note that $\mid t \mid_{\mathscr{D}_i} = t$ if and only if $t \in \mathscr{U}_{\mathscr{D}_i}$.

(3) The $\mathscr{D}_i$*-truncation* of a given $\mathscr{S}$-valuation $\eta \in Val_{\mathscr{S}}$ is the $\mathscr{D}_i$ valuation $\mid \eta \mid_{\mathscr{D}_i}$ defined by the condition $\mid \eta \mid_{\mathscr{D}_i} (X) = \mid \eta(X) \mid_{\mathscr{D}_i}$, for all $X \in \mathscr{V}ar$. Note that $\mid \eta \mid_{\mathscr{D}_i} = \eta$ if and only if $\eta \in Val_{\mathscr{D}_i}$.

Theorem 2 (*Properties of Amalgamated Sums*)
For any $\mathscr{S} = \mathscr{D}_1 \oplus \cdots \oplus \mathscr{D}_n$ of signature Σ constructed as the amalgamated sum of n pairwise joinable domains $\mathscr{D}_i$ of signatures Σ_i ($1 \leqslant i \leqslant n$):

(1) $\mathscr{S}$ is well defined as a constraint domain; i.e., the interpretations of primitive function symbols in $\mathscr{S}$ satisfy the four conditions listed in Definition 1 from Subsection 2.3.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

(2) $\mathcal{S}$ is a conservative extension of $\mathcal{D}_i$ for all $(1 \leqslant i \leqslant n)$; i.e., for all $1 \leqslant i \leqslant n$, for any $p \in SPF_i^m$ other than ==, and for every $\overline{t}_m, t \in \mathcal{U}_{\mathcal{D}_i}$, one has $p^{\mathcal{D}_i}\,\overline{t}_m \to t$ iff $p^{\mathcal{S}}\,\overline{t}_m \to t$.
(3) For all $1 \leqslant i \leqslant n$, for any set of primitive constraints $\Pi \subseteq APCon_{\mathcal{D}_i}$ and for every valuation $\eta \in Val_{\mathcal{D}_i}$, one has $\eta \in (WT)Sol_{\mathcal{D}_i}(\Pi)$ iff $\eta \in (WT)Sol_{\mathcal{S}}(\Pi)$.
(4) For all $1 \leqslant i \leqslant n$, for any set of $\mathcal{D}_i$-specific primitive constraints $\Pi \subseteq APCon_{\mathcal{D}_i}$ and for every valuation $\eta \in Val_{\mathcal{S}}$, one has $\eta \in (WT)Sol_{\mathcal{S}}(\Pi)$ iff $\mid \eta \mid_{\mathcal{D}_i} \in (WT)Sol_{\mathcal{D}_i}(\Pi)$.

Note that amalgamated sums of the form $\mathcal{H} \oplus \mathcal{D}$ are always possible, and give rise to compound domains that can profit from the extensible Herbrand solver. However, in order to construct more interesting sums tailored to the communication among several pure domains, so-called *mediatorial domains* are needed. Given n pairwise joinable domains $\mathcal{D}_i$ with specific signatures $\Sigma_i = \langle TC,\ SBT_i,\ DC,\ DF,\ SPF_i \rangle$ $(1 \leqslant i \leqslant n)$, a *mediatorial domain* for the communication among $\mathcal{D}_1, \ldots, \mathcal{D}_n$ is defined as any domain $\mathcal{M}$ with specific signature $\Sigma_0 = \langle TC,\ SBT_0,\ DC,\ DF,\ SPF_0 \rangle$ constructed in such a way that the following conditions are satisfied:

- $SBT_0 \subseteq \bigcup_{i=1}^{n} SBT_i$, and $SPF_0 \cap SPF_i = \emptyset$ for all $1 \leqslant i \leqslant n$.
- For each $d \in SBT_0$ and for any $1 \leqslant i \leqslant n$ such that $d \in SBT_i$, $\mathcal{B}_d^{\mathcal{M}} = \mathcal{B}_d^{\mathcal{D}_i}$. (no confusion can arise, since the domains $\mathcal{D}_i$ are pairwise joinable).
- Each $p \in SPF_0$ is a so-called *equivalence primitive* $\#{==}_{d_i,d_j}$ with declared principal type $d_i \to d_j \to bool$, for some $1 \leqslant i, j \leqslant n$ and some $d_i \in SBT_i$, $d_j \in SBT_j$.
- Moreover, each equivalence primitive $\#{==}_{d_i,d_j}$ is used in infix notation and there is an injective partial mapping $inj_{d_i,d_j} : \mathcal{B}_{d_i}^{\mathcal{D}_i} \to \mathcal{B}_{d_j}^{\mathcal{D}_j}$ used to define the interpretation of $\#{==}_{d_i,d_j}$ in $\mathcal{M}$ as follows: for all $s, t, r \in \mathcal{U}_{\mathcal{M}}$, $s \ \#{==}_{d_i,d_j}^{\mathcal{M}}\ t \to r$ iff some of the three cases listed below hold:
  (1) $s \in dom(inj_{d_i,d_j})$, $t \in \mathcal{B}_{d_j}^{\mathcal{D}_j}$, $t = inj_{d_i,d_j}(s)$ and $true \sqsupseteq r$.
  (2) $s \in dom(inj_{d_i,d_j})$, $t \in \mathcal{B}_{d_j}^{\mathcal{D}_j}$, $t \neq inj_{d_i,d_j}(s)$ and $false \sqsupseteq r$.
  (3) $r = \perp$.

Equivalence primitives $\#{==}_{d_i,d_j}$ allow to write well-typed atomic *mediatorial constraints* of the form $a \ \#{==}_{d_i,d_j}\ b \ \to!\, c$, using expressions $a :: d_i$, $b :: d_j$ and $c :: bool$. Constraints of the form $a \ \#{==}_{d_i,d_j}\ b \ \to!$ `true` resp. $a \ \#{==}_{d_i,d_j}\ b \ \to!$ `false` are abbreviated as $a \ \#{==}_{d_i,d_j}\ b$ resp. $a \ \#/{==}_{d_i,d_j}\ b$ and called *bridges* and *antibridges*, respectively. The usefulness of bridges for cooperative goal solving in *CFLP* has been motivated in the introduction and will be elaborated when presenting the cooperative goal-solving calculus $CCLNC(\mathcal{C})$ in Section 3. Antibridges and mediatorial constraints $a \ \#{==}_{d_i,d_j}\ b \ \to!\, R$, where R is a variable, can also occur in $CCLNC(\mathcal{C})$ computations, but they are not so directly related to domain cooperation as bridges.

Each particular choice of injective partial mappings inj_{d_i,d_j} and their corresponding equivalence primitives $\#{==}_{d_i,d_j}$ gives rise to the construction of a particular mediatorial domain $\mathcal{M}$, suitable for communication among the $\mathcal{D}_i$. Moreover, it is clear by construction that the $n+1$ domains $\mathcal{M}$, $\mathcal{D}_1$, …, $\mathcal{D}_n$ are pairwise joinable, and it

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

is possible to build the amalgamated sum $\mathcal{C} = \mathcal{M} \oplus \mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_n$. This "hybrid" domain supports the communication among the domains $\mathcal{D}_i$ via bridge constraints provided by $\mathcal{M}$. Therefore, $\mathcal{M}$ is called a *coordination domain* for $\mathcal{D}_1, \ldots, \mathcal{D}_n$.

In practice, it is advisable to include the Herbrand domain $\mathcal{H}$ as one of the component domains $\mathcal{D}_i$ when building a coordination domain $\mathcal{C}$. In application programs over such a coordination domain, the $\mathcal{H}$ solver is typically helpful for solving symbolic equality and disequality constraints over user-defined datatypes, while the solvers of other component domains $\mathcal{D}_i$ whose specific signatures include the primitive `==` may be helpful for computing with equalities and disequalities related to $\mathcal{D}_i$'s specific base types.

2.6 The coordination domain $\mathcal{C} = \mathcal{M} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{R}$

In this subsection, we explain the construction of a coordination domain for cooperation among the three pure domains $\mathcal{H}$, $\mathcal{FD}$, and $\mathcal{R}$.

First, we define a mediatorial domain $\mathcal{M}$ suitable to this purpose. It is built with specific signature $\Sigma_0 = \langle TC, SBT_0, DC, DF, SPF_0\rangle$, where $SBT_0 = \{int, real\}$ and $SPF_0 = \{\texttt{\#==}_{int,real}\}$. The equivalence primitive $\texttt{\#==}_{int,real}$ is interpreted with respect to the total injective mapping $inj_{int,real} :: \mathbb{Z} \rightarrow \mathbb{R}$, which maps each integer value into the equivalent real value. In the sequel, we will write `#==` in place of $\texttt{\#==}_{int,real}$ when referring to this equivalence primitive. We will use the same abbreviation for writing mediatorial constraints.

Next, we use this mediatorial domain for building $\mathcal{C} = \mathcal{M} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{R}$. In the rest of the paper, $\mathcal{C}$ will always stand for this particular coordination domain, whose usefulness has been motivated in Section 1 and will become more evident in Section 3. Note that bridges `X #== RX` and antibridges `X #/== RX` can be useful just as constraints; in particular, `X #== RX` acts as an *integrality constraint* over the value of the real variable `RX`. More importantly, in Section 3 the mediatorial domain $\mathcal{C}$ will serve as a basis for useful cooperation facilities, including the projection of $\mathcal{R}$ constraints to the $\mathcal{FD}$ solver (and vice versa) using bridges, the specialization of $\mathcal{H}$ constraints to become $\mathcal{R}$- or $\mathcal{FD}$-specific in some computational contexts, and some other special mechanisms designed for processing the mediatorial constraints occurring in computations.

In particular, computation rules for simplifying mediatorial constraints will be needed. Although $\mathcal{M}$ is not a "pure" domain, simplifying $\mathcal{M}$ constraints is most conveniently thought of as the task of a $\mathcal{M}$ solver. This solver is expected to deal with $\mathcal{M}$-specific constraint sets $\Pi \subseteq APCon_{\mathcal{M}}$ consisting of atomic primitive constraints π of the form $t \texttt{ \#==} s \rightarrow! b$, where b is either a variable or a Boolean constant and each of the two patterns t and s is either a variable or a numeric value of the proper type (`int` for t and `real` for s). For any finite set $\Pi \subseteq APCon_{\mathcal{M}}$ of such $\mathcal{M}$-specific constraints, it is clear that $dvar_{\mathcal{M}}(\Pi) = var(\Pi)$. Therefore, it is safe to define $odvar_{\mathcal{M}}(\Pi) = var(\Pi)$ and thus $cvar_{\mathcal{M}}(\Pi) = \emptyset$. We define a glass-box solver $solve^{\mathcal{M}}$ by means of the store transformation technique explained in Subsection 2.4.1, using the *strs* for $\mathcal{M}$ stores shown in Table 2. Because of the absence of critical variables, one-step transformations of $\mathcal{M}$ stores do not

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Table 2. *Store transformations for $solve^{\mathcal{M}}$*

M1	$(t \;\texttt{\#==}\; s) \to!\, B,\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{M}} (t \;\texttt{\#==}\; s,\ \Pi)\sigma_1 \,\Box\, \sigma\sigma_1$	if $t \in \mathcal{V}ar \cup \mathbb{Z}$, $s \in \mathcal{V}ar \cup \mathbb{R}$, $B \in \mathcal{V}ar$, where $\sigma_1 = \{B \mapsto true\}$.
M2	$(t \;\texttt{\#==}\; s) \to!\, B,\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{M}} (t \;\texttt{\#/==}\; s,\ \Pi)\sigma_1 \,\Box\, \sigma\sigma_1$	if $t \in \mathcal{V}ar \cup \mathbb{Z}$, $s \in \mathcal{V}ar \cup \mathbb{R}$, $B \in \mathcal{V}ar$, where $\sigma_1 = \{B \mapsto false\}$.
M3	$X \;\texttt{\#==}\; u',\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{M}} \Pi\sigma_1 \,\Box\, \sigma\sigma_1$	if $u' \in \mathbb{R}$, and there is $u \in \mathbb{Z}$ such that $u \;\texttt{\#==}^{\mathcal{M}}\; u' \to true$, where $\sigma_1 = \{X \mapsto u\}$.
M4	$X \;\texttt{\#==}\; u',\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{M}} \blacksquare$	if $u' \in \mathbb{R}$, and there is no $u \in \mathbb{Z}$ such that $u \;\texttt{\#==}^{\mathcal{M}}\; u' \to true$.
M5	$u \;\texttt{\#==}\; RX,\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{M}} \Pi\sigma_1 \,\Box\, \sigma\sigma_1$	if $u \in \mathbb{Z}$ and $u' \in \mathbb{R}$ is so chosen that $u \;\texttt{\#==}^{\mathcal{M}}\; u' \to true$, where $\sigma_1 = \{RX \mapsto u'\}$.
M6	$u \;\texttt{\#==}\; u',\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{M}} \Pi \,\Box\, \sigma$	if $u \in \mathbb{Z}$, $u' \in \mathbb{R}$, and $u \;\texttt{\#==}^{\mathcal{M}}\; u' \to true$.
M7	$u \;\texttt{\#==}\; u',\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{M}} \blacksquare$	if $u \in \mathbb{Z}$, $u' \in \mathbb{R}$, and $u \;\texttt{\#==}^{\mathcal{M}}\; u' \to false$.
M8	$u \;\texttt{\#/==}\; u',\ \Pi \,\Box\, \sigma \Vdash_{\mathcal{M}} \Pi \,\Box\, \sigma$	if $u \in \mathbb{Z}$, $u' \in \mathbb{R}$, and $u \;\texttt{\#==}^{\mathcal{M}}\; u' \to false$
M9	$u \;\texttt{\#/==}\; u',\ M \,\Box\, \sigma \Vdash_{\mathcal{M}} \blacksquare$	if $u \in \mathbb{Z}$, $u' \in \mathbb{R}$, and $u \;\texttt{\#==}^{\mathcal{M}}\; u' \to true$.

depend on a parametrically given set $\mathcal{X}$ of critical variables and have the form $\pi, \Pi \,\Box\, \sigma \Vdash_{\mathcal{M}} \Pi' \,\Box\, \sigma'$, indicating the transformation of any store $\pi, \Pi \,\Box\, \sigma$, which includes the atomic constraint π plus other constraints Π; no sequential ordering is intended. We say that π is the *selected atomic constraint* for this transformation step.

The following theorem ensures that the *sts* for $\mathcal{M}$ stores can be accepted as a correct specification of a glass-box solver for the domain $\mathcal{M}$.

Theorem 3 (*Formal Properties of the $\mathcal{M}$ Solver*)
The *sts* with transition relation $\Vdash_{\mathcal{M}}$ is finitely branching and terminating, and therefore

$$solve^{\mathcal{M}}(\Pi) = \bigvee\{\exists \overline{Y'}(\Pi' \,\Box\, \sigma') \mid \Pi' \,\Box\, \sigma' \in \mathcal{SF}_{\mathcal{M}}(\Pi),\ \overline{Y'} = var(\Pi' \,\Box\, \sigma') \setminus var(\Pi)\}$$

is well defined for any finite $\Pi \subseteq APCon_{\mathcal{M}}$ of $\mathcal{M}$-specific constraints. The solver $solve^{\mathcal{M}}$ satisfies all the requirements for solvers enumerated in Definition 6. Moreover, whenever $\Pi \subseteq APCon_{\mathcal{M}}$ is $\mathcal{M}$-specific and $\Pi \Vdash_{solve^{\mathcal{M}}} \exists \overline{Y'}(\Pi' \,\Box\, \sigma')$, the constraint set Π' is also $\mathcal{M}$-specific and $\sigma'(X)$ is either a Boolean value, an integer value or a real value for all $X \in vdom(\sigma')$.

The proof is omitted, because it is completely similar to that of Theorem 1 but much easier. In fact, the *sts* for $\mathcal{M}$ stores involves no decompositions. Actually, this *sts* is finitely branching, terminating, locally sound, and locally complete. Therefore, Lemma 5 can be applied.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

The framework for cooperative programming and the cooperative goal-solving calculus $CCLNC(\mathscr{C})$ presented in Section 3 essentially rely on the coordination domain $\mathscr{C}$ just discussed, and the instance $CFLP(\mathscr{C})$ of the $CFLP$ scheme (López-Fraguas *et al.* 2007) provides a declarative semantics for proving the soundness and completeness of $CCLNC(\mathscr{C})$. As we will see, some cooperative goal-solving rules in $CCLNC(\mathscr{C})$ rely on the identification of certain atomic primitive Herbrand constraints π as $\mathscr{FD}$- or $\mathscr{R}$-specific, respectively, on the basis of the mediatorial constraints available in a given $\mathscr{M}$ store M. The notations $M \vdash \pi$ `in` $\mathscr{FD}$ and $M \vdash \pi$ `in` $\mathscr{R}$ defined below serve to this purpose.

Definition 9 (*Inference of Domain-specific Extended Herbrand Constraints*)
Assume a mediatorial store M and a well-typed atomic extended Herbrand constraint π having the form t_1 `==` t_2 or t_1 `/=` t_2, where each of the two patterns t_1 and t_2 is either a numeric constant v or a variable V. Then we define:

(1) $M \vdash \pi$ `in` $\mathscr{FD}$ (read as "M allows to infer that π is $\mathscr{FD}$-specific") iff some of the following three conditions hold:
   (a) t_1 or t_2 is an integer constant.
   (b) t_1 or t_2 is a variable that occurs as the left argument of the operator `#==` within some mediatorial constraint belonging to M.
   (c) t_1 or t_2 is a variable that has been recognized to have type `int` by some implementation-dependent device.

(2) $M \vdash \pi$ `in` $\mathscr{R}$ (read as "M allows to infer that π is $\mathscr{R}$-specific") iff some of the following three conditions hold:
   (a) t_1 or t_2 is a real constant.
   (b) t_1 or t_2 is a variable that occurs as the right argument of the operator `#==` within some mediatorial constraint belonging to M.
   (c) t_1 or t_2 is a variable that has been recognized to have type `real` by some implementation-dependent device.

3 Cooperative programming and goal solving in $CFLP(\mathscr{C})$

This section presents our cooperative computation model for goal solving. After introducing programs and goals in the first subsection, the subsequent subsections deal with goal-solving rules, illustrative computation examples, and results concerning the formal properties of the computation model.

Our goal-solving rules work by transforming initial goals into final goals in solved form which serve as computed answers, as in the previously published constrained lazy narrowing calculus $CLNC(\mathscr{D})$ (López-Fraguas *et al.* 2004), which works over any parametrically given domain $\mathscr{D}$ equipped with a solver. We have substantially extended $CLNC(\mathscr{D})$ with various mechanisms for domain cooperation via bridges, projections, and some more ad hoc operations. The result is a cooperative constrained lazy narrowing calculus $CCLNC(\mathscr{C})$ which is sound and complete (with some limitations) w.r.t. the instance $CFLP(\mathscr{C})$ of the generic $CFLP$ scheme (López-Fraguas *et al.* 2007). For the sake of typographic simplicity, we have restricted our

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

presentation of $CCLNC(\mathcal{C})$ to the case $\mathcal{C} = \mathcal{M} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{R}$, although it could be easily extended to other coordination domains, as sketched in our previous paper (Estévez-Martín *et al.* 2007b).

3.1 Programs and goals

$CFLP(\mathcal{C})$-programs are sets of constrained rewrite rules that define the behavior of possibly higher-order and/or nondeterministic lazy functions over $\mathcal{C}$, called *program rules*. More precisely, a program rule Rl for a defined function symbol $f \in DF^n_\Sigma$ with principal type $f :: \overline{\tau}_n \to \tau$ has the form $f\,\overline{t}_n \to r \Leftarrow \Delta$, where $\overline{t}_n$ is a linear sequence of patterns, r is an expression, and Δ is a finite conjunction $\delta_1, \ldots, \delta_m$ of atomic constraints $\delta_i \in ACon_{\mathcal{C}}$. Each program rule Rl is required to be well-typed, i.e., there must exist some type environment Γ for the variables occurring in Rl such that $\Sigma, \Gamma \vdash_{WT} t_i :: \tau_i$ for all $1 \leqslant i \leqslant n$, $\Sigma, \Gamma \vdash_{WT} r :: \tau$ and $\Sigma, \Gamma \vdash_{WT} \delta_i$ for all $1 \leqslant i \leqslant m$.

The left-linearity requirement is quite common in functional and functional logic programming. As in CLP, the conditional part of a program rule needs no explicit occurrences of existential quantifiers. A program rule Rl is said to include *free occurrences of higher-order logic variables* iff there is some variable X which does not occur in the left-hand side of Rl but has some occurrence in a context of the form $X\,\overline{e}_m$ (with $m > 0$) somewhere else in Rl. A program $\mathcal{P}$ includes free occurrences of higher-order logic variables iff some of the program rules in $\mathcal{P}$ do.

As in functional languages such as Haskell (Peyton-Jones 2002), our programs rules can deal with higher-order functions and are not expected to be always terminating. Moreover, in contrast to Haskell and most other functional languages, we do not require program rules to be confluent. Therefore, some program defined functions can be *nondeterministic* and return several values for a fixed choice of arguments in some cases. As a concrete example of typed $CFLP(\mathcal{C})$-program written in the concrete syntax of the $\mathcal{TOY}$ system, we refer to the program rules presented in Subsection 1.2.

Programs are used to solve *goals* using a cooperative goal-solving calculus which will be described in Subsections 3.2, 3.3, and 3.4. Goals over the coordination domain $\mathcal{C}$ have the general form $G \equiv \exists \overline{U}.\ P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R$, where the symbol $\Box$ is interpreted as conjunction and:

- $\overline{U}$ is a finite set of so-called *existential variables*, intended to represent local variables in the computation.
- P is a set of so-called *productions* of the form $e_1 \to t_1, \ldots, e_m \to t_m$, where $e_i \in Exp_{\mathcal{C}}$ and $t_i \in Pat_{\mathcal{C}}$ for all $1 \leqslant i \leqslant m$. The set of *produced variables* of G is defined as the set $pvar(P)$ of variables occurring in $t_1 \ldots t_m$. During goal solving, productions are used to compute values for the produced variables insofar as demanded, using the goal-solving rules for constrained lazy narrowing presented in Subsection 3.2. We consider a *production relation* between variables, such that $X \gg_P Y$ holds iff $X, Y \in pvar(P)$ and there is some $1 \leqslant i \leqslant m$ such that $X \in var(e_i)$ and $Y \in var(t_i)$.

- C is the so-called *constraint pool*, a finite set of constraints to be solved, possibly including active occurrences of defined functions symbols.
- $M = \Pi_M \,\square\, \sigma_M$ is the *mediatorial store*, including a finite set of atomic primitive constraints $\Pi_M \subseteq APCon_{\mathscr{M}}$ and a substitution σ_M. We will write $B_M \subseteq \Pi_M$ for the set of all $\pi \in \Pi_M$ which are *bridges* t `#==` s, where each of the two patterns t and s may be either a variable or a numeric constant.
- $H = \Pi_H \,\square\, \sigma_H$ is the *Herbrand store*, including a finite set of atomic primitive constraints $\Pi_H \subseteq APCon_{\mathscr{H}}$ and a substitution σ_H.
- $F = \Pi_F \,\square\, \sigma_F$ is the *finite domain store*, including a finite set of atomic primitive constraints $\Pi_F \subseteq APCon_{\mathscr{FD}}$ and a substitution σ_F.
- $R = \Pi_R \,\square\, \sigma_R$ is the *real arithmetic store*, including a finite set of atomic primitive constraints $\Pi_R \subseteq APCon_{\mathscr{R}}$ and a substitution σ_R.

A goal G is said to have *free occurrences of higher-order logic variables* iff there is some variable X occurring in G in some context of the form $X\,\overline{e}_m$, with $m > 0$. Two special kinds of goals are particularly interesting. *Initial goals* just consist of a well-typed constraint pool C. More precisely, the existential prefix $\overline{U}$, productions in P, and stores M, H, F, and R are empty. *Solved goals* (also called *solved forms*) have empty P and C parts and cannot be transformed by any goal-solving rule. Therefore, they are used to represent *computed answers*. We will also write $\blacksquare$ to denote an *inconsistent goal*.

Example 7 (*Initial and Solved Goals*)
Consider the initial goals **Goal 1**, **Goal 2,** and **Goal 3** presented in $\mathscr{TOY}$ syntax in Subsection 1.2, for the choice `d = 2`, `n = 4`. When written with the abstract syntax for general $CFLP(\mathscr{C})$-goals they become

(1) □ bothIn (triangle (2, 2.75) 4 0.5) (square 4) (X,Y) □□□□
(2) □ bothIn (triangle (2, 2.5) 2 1) (square 4) (X,Y) □□□□
(3) □ bothIn (triangle (2, 2.5) 8 1) (square 4) (X,Y) □□□□

The expected solutions for these goals have been explained in Subsection 1.2. A general notion of solution for goals will be defined in Subsection 3.6. The resolution of these example goals in our cooperative goal-solving calculus $CCLNC(\mathscr{C})$ will be discussed in detail in Subsection 3.5. The respective solved forms obtained as computed answers (restricted to the variables in the initial goal) will be:

(1) $\blacksquare$
(2) $\square\square\square\square\ (\diamond\ \square\ \{X \mapsto 2, Y \mapsto 2\})\ \square$
(3) $\square\square\square\square\ (\diamond\ \square\ \{X \mapsto 0, Y \mapsto 2\})\ \square$
$\square\square\square\square\ (\diamond\ \square\ \{X \mapsto 1, Y \mapsto 2\})\ \square$
$\square\square\square\square\ (\diamond\ \square\ \{X \mapsto 2, Y \mapsto 2\})\ \square$
$\square\square\square\square\ (\diamond\ \square\ \{X \mapsto 3, Y \mapsto 2\})\ \square$
$\square\square\square\square\ (\diamond\ \square\ \{X \mapsto 4, Y \mapsto 2\})\ \square$

The goal-solving rules of the $CCLNC(\mathscr{C})$ calculus presented in the rest of this section have been designed as an extension of an existing goal-solving calculus for the $CFLP$ scheme (López-Fraguas *et al.* 2004), adding the new features needed to

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

support solver coordination via bridge constraints. In contrast to previous related works such as Loogen *et al.* (1993), Antoy *et al.* (1994, 2000), and del Vado-Vírseda (2003, 2005, 2007), we have omitted the use of so-called *definitional trees* to ensure an optimal selection of needed narrowing steps. This feature could be easily added to $CCLNC(\mathcal{C})$ following the ideas from del Vado-Vírseda (2005), but we have decided not do so in order to avoid technical complications which do not contribute to a better understanding of domain cooperation. Moreover, the design of $CCLNC(\mathcal{C})$ is tailored to programs and goals having no free occurrences of higher-order logic variables. As shown in González-Moreno *et al.* (2001), goal-solving rules for dealing with free higher-order logic variables give rise to ill-typed solutions very often. If desired, they could be easily added to our present setting.

Let us finish this subsection with a brief discussion of some technical issues needed in the sequel. The set $odvar(G)$ of *obviously demanded variables* in a given goal G is defined as the least subset of $var(G)$ which satisfies the following two conditions:

(1) $odvar(G)$ includes $odvar_{\mathcal{M}}(\Pi_M)$, $odvar_{\mathcal{H}}(\Pi_H)$, $odvar_{\mathcal{FD}}(\Pi_F)$, and $odvar_{\mathcal{R}}(\Pi_R)$ which are defined as explained in Subsections 2.3 and 2.4.
(2) $X \in odvar(G)$ for any production $(X\overline{a}_k \to t) \in P$ such that $k > 0$ and either $t \notin \mathcal{V}ar$ or else $t \in odvar(G)$.

Note that $odvar(G)$ boils down to $odvar_{\mathcal{M}}(\Pi_M) \cup odvar_{\mathcal{H}}(\Pi_H) \cup odvar_{\mathcal{FD}}(\Pi_F) \cup odvar_{\mathcal{R}}(\Pi_R)$ in the case that G has no free occurrences of higher-order variables. Productions $e \to X$ such that e is an active expression and $X \notin odvar(G)$ is a not obviously demanded variable are called *suspensions*, and play an important role during goal solving.

Certain properties are trivially satisfied by initial goals and kept invariant through the application of goal transformations. Such *goal invariant properties* include those formalized in previous works as, e.g., López-Fraguas *et al.* (2004): each produced variable is produced only once, all the produced variables must be existential, the transitive closure $\gg_P^+$ of the relation between produced variables must be irreflexive, and no produced variable occurs in the answer substitutions. Other goal invariants are more specific of our current cooperative setting based on the coordination domain $\mathcal{C}$:

- The domains of the substitutions σ_M, σ_H, σ_F, and σ_R are pairwise disjoint.
- For any store S in G, the application of σ_S causes no modification to the goal.
- For any $X \in vdom(\sigma_M)$, $\sigma_M(X)$ is either a Boolean value, an integer value, or a real value.
- For any $X \in vdom(\sigma_F)$, $\sigma_F(X)$ is either an integer value or a variable occurring in Π_F.
- For any $X \in vdom(\sigma_R)$, $\sigma_R(X)$ is either a real value or a variable occurring in Π_R.

These properties remain invariant through goal transformations because of Theorem 3 and Postulates 2 and 1, and also because the bindings computed by each particular solver are properly propagated to the rest of the goal. In particular,

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

whenever a variable binding $\{X \mapsto t\}$ arises in one of the stores during goal solving, the propagation of this binding to the goal *applies* the binding everywhere, but *places* it only within the substitution of this particular store, so that the first item above is ensured.

At this point, we must introduce some auxiliary notations in order to make this idea more precise. Let $\mathscr{D}$ stand for any of the four domains $\mathscr{M}$, $\mathscr{H}$, $\mathscr{FD}$, or $\mathscr{R}$ and consider the store $S = \Pi_S \,\square\, \sigma_S$ corresponding to $\mathscr{D}$. We will note as $(P \,\square\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)@_{\mathscr{D}}\sigma'$ the result of *applying* σ' to $P \,\square\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R$ and *composing* σ_S with σ'. More formally, in the particular case that $\mathscr{D}$ is chosen as $\mathscr{FD}$, we define $(P \,\square\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)@_{\mathscr{FD}}\sigma'$ as $P\sigma' \,\square\, C\sigma' \,\square\, M \star \sigma' \,\square\, H \star \sigma' \,\square\, F@\sigma' \,\square\, R \star \sigma'$, where $F@\sigma'$ is defined as $\Pi_F\sigma' \,\square\, \sigma_F\sigma'$ and $S \star \sigma'$ is defined as $\Pi_S\sigma' \,\square\, \sigma_S \star \sigma'$ for S being M, H, or R. Recall that the *application* of σ' to σ_S has been defined as $\sigma_S \star \sigma' = \sigma_S\sigma' \upharpoonright vdom(\sigma_S)$ in Subsection 2.2, and note that $\sigma_S \star \sigma'$ retains the same domain as σ_S.

The notations explained in the previous paragraph will be used for presenting several goal transformation rules in the next subsections. The formal definition for the other three possible choices of $\mathscr{D}$ is completely analogous. In the rest of the paper, we will restrict our attention to so-called *admissible goals* G that arise from initial goals through the iterated application of goal transformation rules and enjoy the goal invariant properties just described.

3.2 Constrained lazy narrowing rules

The core of our cooperative goal-solving calculus $CLNC(\mathscr{C})$ consists of the goal-solving rules displayed in Table 3. Roughly speaking, these rules model the behavior of constrained lazy narrowing ignoring domain cooperation and solver invocation. They have been adapted from López-Fraguas *et al.* (2004) and can be classified as follows: the first four rules encode unification transformations similar to those found in the $\mathscr{H}$ *sts* (see Subsection 2.4.2) and other related formalisms; rule **EL** discharges unneeded suspensions, rule **DF** (presented in two cases in order to optimize the $k = 0$ case) applies program rules to deal with calls to program defined functions; rule **PC** transforms demanded calls to primitive functions into atomic constraints that are placed in the pool; and rule **FC**, working in interplay with **PC**, transforms the atomic constraints in the pool into a flattened form consisting of a conjunction of atomic primitive constraints with new existential variables.

The behavior of the main rules in Table 3 will be illustrated in Subsection 3.5. Example 8 focuses on the transformation rules **PC** and **FC**. Their iterated application flattens the atomic $\mathscr{R}$ constraint `(RX + 2*RY)*RZ <= 3.5` into a conjunction of four atomic primitive $\mathscr{R}$ constraints involving three new existential variables that are placed in the constraint pool. Note that López-Fraguas *et al.* (2004) and other previous related calculi also include rules that can be used to achieve constraint flattening, but the resulting atomic primitive constraints are placed in a constraint store. In our present setting, they are kept in the pool in order that the domain cooperation rules described in the next subsection can process them.

Table 3. *Rules for constrained lazy narrowing*

DC DeComposition

$\exists\overline{U}.\ h\ \overline{e_m} \to h\ \overline{t_m},\ P \ \Box\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R \Vdash_{\mathbf{DC}} \exists\overline{U}.\ \overline{e_m \to t_m},\ P \ \Box\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R$

CF Conflict Failure

$\exists\overline{U}.\ e \to t, P \ \Box\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R \Vdash_{\mathbf{CF}} \blacksquare$

If e is rigid and passive, $t \notin \mathcal{V}ar$, e and t have conflicting roots.

SP Simple Production

$\exists\overline{U}.\ s \to t,\ P \ \Box\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R \Vdash_{\mathbf{SP}} \exists\overline{U}'.\ (P \ \Box\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R)@_{\mathcal{H}}\sigma'$

If $s = X \in \mathcal{V}ar$, $t \notin \mathcal{V}ar$, $\sigma' = \{X \mapsto t\}$ and $\overline{U}' = \overline{U}$ or else $s \in Pat_{\mathcal{C}}$, $t = X \in \mathcal{V}ar$, $\sigma' = \{X \mapsto s\}$ and $\overline{U}' = \overline{U}\backslash\{X\}$.

IM IMitation

$\exists X, \overline{U}.\ h\ \overline{e_m} \to X,\ P \ \Box\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R \Vdash_{\mathbf{IM}} \exists\overline{X_m}, \overline{U}.\ (\overline{e_m \to X_m},\ P \ \Box\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R)\sigma'$

If $h\ \overline{e_m} \notin Pat_{\mathcal{C}}$ is passive, $X \in odvar(G)$ and $\sigma' = \{X \mapsto h\ \overline{X_m}\}$.

EL ELimination

$\exists X, \overline{U}.\ e \to X,\ P \ \Box\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R \Vdash_{\mathbf{EL}} \exists\overline{U}.\ P \ \Box\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R$

If X does not occur in the rest of the goal.

DF Defined Function

$\exists\overline{U}.\ f\ \overline{e_n} \to t,\ P \ \Box\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R \Vdash_{\mathbf{DF}_f}$
$\exists\overline{Y}, \overline{U}.\ \overline{e_n \to t_n},\ r \to t,\ P \ \Box\ C',\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R$

If $f \in DF^n$, $t \notin \mathcal{V}ar$ or $t \in odvar(G)$ and $Rl : f\ \overline{t_n} \to r \Leftarrow C'$ is a fresh variant of a rule in $\mathcal{P}$, with $\overline{Y} = var(Rl)$ new variables.

$\exists\overline{U}.\ f\ \overline{e_n a_k} \to t,\ P \ \Box\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R \Vdash_{\mathbf{DF}_f}$
$\exists X, \overline{Y}, \overline{U}.\ \overline{e_n \to t_n},\ r \to X,\ X\ \overline{a_k} \to t,\ P \ \Box\ C',\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R$

If $f \in DF^n$ $(k > 0)$, $t \notin \mathcal{V}ar$ or $t \in odvar(G)$ and $Rl : f\ \overline{t_n} \to r \Leftarrow C'$ is a fresh variant of a rule in $\mathcal{P}$, with $\overline{Y} = var(Rl)$ and X new variables.

PC Place Constraint

$\exists\overline{U}.\ p\ \overline{e_n} \to t,\ P \ \Box\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R \Vdash_{\mathbf{PC}} \exists\overline{U}.\ \ P \ \Box\ p\ \overline{e_n} \to!\ t,\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R$

If $p \in PF^n$ and $t \notin \mathcal{V}ar$ or $t \in odvar(G)$.

FC Flatten Constraint

$\exists\overline{U}.\ P \ \Box\ p\ \overline{e_n} \to!\ t,\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R \Vdash_{\mathbf{FC}}$
$\exists\overline{V}_m, \overline{U}.\ \overline{a_m \to V_m},\ P \ \Box\ p\ \overline{t}_n \to!\ t,\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ R$

If $p \in PF^n$, some $e_i \notin Pat_{\mathcal{C}}$, $\overline{a_m}$ $(m \leqslant n)$ are those e_i which are not patterns, $\overline{V_m}$ are new variables, and $p\ \overline{t_n}$ is obtained from $p\ \overline{e_n}$ by replacing each e_i which is not a pattern by V_i.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Example 8 (*Constraint Flattening*)

$\square\ \underline{(RX + 2 * RY) * RZ}$ <= 3.5 $\square\square\square\square \Vdash_{\mathbf{FC}}$

$\exists RA.\ \underline{(RX + 2 * RY) * RZ\ \rightarrow\ RA}\ \square\ RA$ <= 3.5 $\square\square\square\square \Vdash_{\mathbf{PC}}$

$\exists RA.\ \square\ \underline{(RX + 2 * RY) * RZ\ \rightarrow!\ RA},\ RA$ <= 3.5 $\square\square\square\square \Vdash_{\mathbf{FC}}$

$\exists RB, RA. \underline{RX + 2 * RY\ \rightarrow RB}\ \square\ RB * RZ \rightarrow!RA, RA$ <= 3.5 $\square\square\square\square \Vdash_{\mathbf{PC}}$

$\exists RB, RA.\ \square\ \underline{RX + 2 * RY\ \rightarrow!\ RB},\ RB * RZ\ \rightarrow!\ RA,\ RA$ <= 3.5 $\square\square\square\square \Vdash_{\mathbf{FC}}$

$\exists RC, RB, RA.\ \underline{2 * RY\ \rightarrow RC}\ \square\ RX + RC \rightarrow!RB,\ RB * RZ \rightarrow!RA,\ RA$ <= 3.5 $\square\square\square\square \Vdash_{\mathbf{PC}}$

$\exists RC, RB, RA.\ \square\ 2 * RY\ \rightarrow!RC,\ RX + RC \rightarrow!RB,\ RB * RZ \rightarrow!RA,\ RA$ <= 3.5 $\square\square\square\square$

Note that suspensions $e \rightarrow X$ can be discharged by rule **EL** in case that X does not occur in the rest of the goal. Otherwise, they must wait until X gets bound to a nonvariable pattern or becomes obviously demanded, and then they can be processed by using either rule **DF** or rule **PC**, according to the syntactic form of e. Moreover, all the substitutions produced by the transformations bind variables X to patterns t, standing for computed values that are shared by all the occurrences of t in the current goal. In this way, the goal transformation rules encode a lazy narrowing strategy.

3.3 Domain cooperation rules

This subsection presents the goal transformation rules in $CCLNC(\mathscr{C})$ which take care of domain cooperation. The core of the subsection deals with bridges and projections. A few more ad hoc cooperation rules are presented at the end of the subsection.

Given a goal G whose pool C includes an atomic primitive constraint $\pi \in APCon_{\mathscr{FD}}$ and whose mediatorial store M includes a set of bridges B_M, we will consider three possible goal transformations intended to convey useful information from π to the $\mathscr{R}$ solver:

- To compute new bridges $bridges^{\mathscr{FD}\rightarrow\mathscr{R}}(\pi, B_M)$ to add to M, by means of a *bridge generation* operation $bridges^{\mathscr{FD}\rightarrow\mathscr{R}}$ defined to this purpose.
- To compute projected $\mathscr{R}$ constraints $proj^{\mathscr{FD}\rightarrow\mathscr{R}}(\pi, B_M)$ to be added to R, by means of a *projection* operation $proj^{\mathscr{FD}\rightarrow\mathscr{R}}$ defined to this purpose.
- To place π into the $\mathscr{FD}$ store F.

Similar goal transformations based on two operations $bridges^{\mathscr{R}\rightarrow\mathscr{FD}}$ and $proj^{\mathscr{R}\rightarrow\mathscr{FD}}$ can be used to convey useful information from a primitive atomic constraint $\pi \in PCon_{\mathscr{R}}$ to the $\mathscr{FD}$ solver. Rules **SB**, **PP,** and **SC** in Table 4 formalize these transformations, while Tables 5 and 6 give an effective specification of the bridge generation and projection operations.

The formulation of **SB**, **PP,** and **SC** in Table 4 relies on the identification of certain atomic primitive Herbrand constraints π as $\mathscr{FD}$- or $\mathscr{R}$-specific, as indicated by the notations $M \vdash \pi$ `in` $\mathscr{FD}$ and $M \vdash \pi$ `in` $\mathscr{R}$, previously explained in Subsection 2.6.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Table 4. *Rules for bridges and projections*

SB Set Bridges

$\exists\overline{U}.\ P \,\square\, \pi,\ C \,\square\, M \,\square\, H \,\square\, F \,\square\, R \Vdash_{\mathbf{SB}} \exists\overline{V'},\overline{U}.\ P \,\square\, \pi,\ C \,\square\, M' \,\square\, H \,\square\, F \,\square\, R$

If π is a primitive atomic constraint and either (i) or (ii) holds, where

(i) π is a proper $\mathcal{FD}$ constraint or else an extended $\mathcal{H}$ constraint such that $M \vdash \pi$ `in` $\mathcal{FD}$, and $M' = B', M$, where $\exists\overline{V'}\, B' = bridges^{\mathcal{FD}\to\mathcal{R}}(\pi, B_M) \neq \emptyset$.
(ii) π is a proper $\mathcal{R}$ constraint or else an extended $\mathcal{H}$ constraint such that $M \vdash \pi$ `in` $\mathcal{R}$, and $M' = B', M$, where $\exists\overline{V'}\, B' = bridges^{\mathcal{R}\to\mathcal{FD}}(\pi, B_M) \neq \emptyset$.

PP Propagate Projections

$\exists\overline{U}.\ P \,\square\, \pi,\ C \,\square\, M \,\square\, H \,\square\, F \,\square\, R \Vdash_{\mathbf{PP}} \exists\overline{V'},\ \overline{U}.\ P \,\square\, \pi,\ C \,\square\, M \,\square\, H \,\square\, F' \,\square\, R'$

If π is a primitive atomic constraint and either (i) or (ii) holds, where

(i) π is a proper $\mathcal{FD}$ constraint or else an extended $\mathcal{H}$ constraint such that $M \vdash \pi$ `in` $\mathcal{FD}$, $\exists\overline{V'}\,\Pi' = proj^{\mathcal{FD}\to\mathcal{R}}(\pi, B_M) \neq \emptyset$, $F' = F$, and $R' = \Pi', R$, or else,
(ii) π is a proper $\mathcal{R}$ constraint or else an extended $\mathcal{H}$ constraint such that $M \vdash \pi$ `in` $\mathcal{R}$, $\exists\overline{V'}\,\Pi' = proj^{\mathcal{R}\to\mathcal{FD}}(\pi, B_M) \neq \emptyset$, $F' = \Pi', F$, and $R' = R$.

SC Submit Constraints

$\exists\overline{U}.\ P \,\square\, \pi,\ C \,\square\, M \,\square\, H \,\square\, F \,\square\, R \Vdash_{\mathbf{SC}} \exists\overline{U}.\ P \,\square\, C \,\square\, M' \,\square\, H' \,\square\, F' \,\square\, R'$

If π is a primitive atomic constraint and one of the following cases applies:

(i) π is a $\mathcal{M}$ constraint, $M' = \pi, M$, $H' = H$, $F' = F$, and $R' = R$.
(ii) π is an extended $\mathcal{H}$ constraint such that neither $M \vdash \pi$ `in` $\mathcal{FD}$ nor $M \vdash \pi$ `in` $\mathcal{R}$, $M' = M$, $H' = \pi, H$, $F' = F$, and $R' = R$.
(iii) π is a proper $\mathcal{FD}$ constraint or else an extended $\mathcal{H}$ constraint such that $M \vdash \pi$ `in` $\mathcal{FD}$, $M' = M$, $H' = H$, $F' = \pi, F$, and $R' = R$.
(iv) π is a proper $\mathcal{R}$ constraint or else an extended $\mathcal{H}$ constraint such that $M \vdash \pi$ `in` $\mathcal{R}$, $M' = M$, $H' = H$, $F' = F$, and $R' = \pi, R$.

The notation Π, S is used at several places to indicate the new store obtained by adding the set of constraints Π to the constraints within store S. The notation π, S (where π is a single constraint) must be understood similarly. In practice, **SB**, **PP,** and **SC** are best applied in this order. Note that **PP** places the projected constraints in their corresponding stores, while constraints in the pool that are not useful anymore for computing additional bridges or projections will be eventually placed into their stores by means of transformation **SC**.

The functions $bridges^{\mathcal{D}\to\mathcal{D}'}$ and $proj^{\mathcal{D}\to\mathcal{D}'}$ are specified in Table 5 for the case $\mathcal{D} = \mathcal{FD}$, $\mathcal{D}' = \mathcal{R}$ and in Table 6 for the case $\mathcal{D} = \mathcal{R}$, $\mathcal{D}' = \mathcal{FD}$. Note that the primitive `#/` is not considered in Table 5 because integer division constraints cannot be projected into real division constraints. The notations $\lceil a \rceil$ (resp. $\lfloor a \rfloor$) used in Table 6 stand for the least integer upper bound (resp. the greatest integer lower bound) of $a \in \mathbb{R}$. Constraints t_1 `>` t_2, t_1 `>=` t_2 are not explicitly considered in Table 6; they are treated as t_2 `<` t_1, t_2 `<=` t_1, respectively. In Tables 5 and 6, the existential quantification of the new variables $\overline{V'}$ is left implicit, and results displayed as an empty set of constraints must be read as an empty (and thus trivially true) conjunction.

The next result states some basic properties of $bridges^{\mathcal{D}\to\mathcal{D}'}$ and $proj^{\mathcal{D}\to\mathcal{D}'}$. The easy proof is omitted.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Table 5. *Computing bridges and projections from $\mathcal{FD}$ to $\mathcal{R}$*

π	$bridges^{\mathcal{FD}\to\mathcal{R}}(\pi, B)$	$proj^{\mathcal{FD}\to\mathcal{R}}(\pi, B)$
`domain` $[X_1,\ldots,X_n]$ a b	$\{X_i$ `#==` $RX_i \mid 1 \leqslant i \leqslant n$, X_i has no bridge in B and RX_i new$\}$	$\{a$ `<=` RX_i, RX_i `<=` $b \mid 1 \leqslant i \leqslant n$ and $(X_i$ `#==` $RX_i) \in B\}$
`belongs` X $[a_1,\ldots,a_n]$	$\{X$ `#==` $RX \mid X$ has no bridge in B and RX new$\}$	$\{min(a_1,\ldots,a_n)$ `<=` RX, RX `<=` $max(a_1,\ldots,a_n) \mid 1 \leqslant i \leqslant n$ and $(X$ `#==` $RX) \in B\}$
t_1 `#<` t_2 (resp. `#<=`, `#>`, `#=>`)	$\{X_i$ `#==` $RX_i \mid 1 \leqslant i \leqslant 2$, t_i is a variable X_i with no bridge in B, and RX_i new$\}$	$\{t_1^{\mathcal{R}}$ `<` $t_2^{\mathcal{R}} \mid$ For $1 \leqslant i \leqslant 2$: either t_i is an integer constant n and $t_i^{\mathcal{R}}$ is the integral real n, or else t_i is a variable X_i with $(X_i$ `#==` $RX_i) \in B$, and $t_i^{\mathcal{R}}$ is $RX_i\}$
t_1 `==` t_2	$\{X$ `#==` $RX \mid$ either t_1 is an integer constant and t_2 is a variable X with no bridges in B (or vice versa) and RX is new$\}$	$\{t_1^{\mathcal{R}}$ `==` $t_2^{\mathcal{R}} \mid$ For $1 \leqslant i \leqslant 2$: $t_i^{\mathcal{R}}$ is determined as in the `#<` case$\}$
t_1 `/=` t_2	$\{X$ `#==` $RX \mid$ either t_1 is an integer constant and t_2 is a variable X with no bridges in B (or vice versa) and RX is new$\}$	$\{t_1^{\mathcal{R}}$ `/=` $t_2^{\mathcal{R}} \mid$ For $1 \leqslant i \leqslant 2$: $t_i^{\mathcal{R}}$ is determined as in the `#<` case$\}$
t_1 `#+` $t_2 \to! t_3$ (resp. `#-`, `#*`)	$\{X_i$ `#==` $RX_i \mid 1 \leqslant i \leqslant 3$, t_i is a variable X_i with no bridge in B and RX_i new$\}$	$\{t_1^{\mathcal{R}}$ `+` $t_2^{\mathcal{R}} \to! t_3^{\mathcal{R}} \mid$ For $1 \leqslant i \leqslant 3$: $t_i^{\mathcal{R}}$ is determined as in the `#<` case$\}$

Proposition 1 (*Properties of Bridges and Projections Between $\mathcal{FD}$ and $\mathcal{R}$*)
Let $\mathcal{D}$ and $\mathcal{D}'$ be chosen as $\mathcal{FD}$ and $\mathcal{R}$, or vice versa. Then:

(1) $bridges^{\mathcal{D}\to\mathcal{D}'}(\pi, B)$ and $proj^{\mathcal{D}\to\mathcal{D}'}(\pi, B)$ make sense for any atomic primitive constraint π which is either $\mathcal{D}$-proper or extended Herbrand and $\mathcal{D}$-specific, and for any finite set B of bridges.

(2) $bridges^{\mathcal{D}\to\mathcal{D}'}(\pi, B)$ returns a possibly empty finite set B' of new bridges involving new variables $\overline{V'}$. In particular, $bridges^{\mathcal{D}\to\mathcal{D}'}(\pi, B) = \emptyset$ is assumed whenever Tables 5 and 6 do not include any row covering π. The *completeness condition* $WTSol_{\mathcal{C}}(\pi \wedge B) \subseteq WTSol_{\mathcal{C}}(\exists \overline{V'}(\pi \wedge B \wedge B'))$ holds, where B and B' are interpreted as conjunctions. Note that the *correctness condition* $Sol_{\mathcal{C}}(\pi \wedge B) \supseteq Sol_{\mathcal{C}}(\exists \overline{V'}(\pi \wedge B \wedge B'))$ also holds trivially.

(3) $proj^{\mathcal{D}\to\mathcal{D}'}(\pi, B)$ returns a finite set $\Pi' \subseteq APCon_{\mathcal{D}'}$ of atomic primitive $\mathcal{D}'$ constraints involving new variables $\overline{V'}$. In particular, $proj^{\mathcal{D}\to\mathcal{D}'}(\pi, B) = \emptyset$ is assumed whenever Tables 5 and 6 do not include any row covering π. The *completeness condition* $WTSol_{\mathcal{C}}(\pi \wedge B) \subseteq WTSol_{\mathcal{C}}(\exists \overline{V'}(\pi \wedge B \wedge \Pi'))$ holds,

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Table 6. *Computing bridges and projections from $\mathcal{R}$ to $\mathcal{FD}$*

π	$bridges^{\mathcal{R}\to\mathcal{FD}}(\pi, B)$	$proj^{\mathcal{R}\to\mathcal{FD}}(\pi, B)$
RX < RY	$\emptyset$ (*no bridges are created*)	$\{X$ #< $Y \mid (X$ #== $RX),(Y$ #== $RY) \in B\}$
RX < a	$\emptyset$ (*no bridges are created*)	$\{X$ #< $\lceil a \rceil \mid a \in \mathbb{R}, (X$ #== $RX) \in B\}$
a < RY	$\emptyset$ (*no bridges are created*)	$\{\lfloor a \rfloor$ #< $Y \mid a \in \mathbb{R}, (Y$ #== $RY) \in B\}$
RX <= RY	$\emptyset$ (*no bridges are created*)	$\{X$ #<= $Y \mid (X$ #== $RX),(Y$ #== $RY) \in B\}$
RX <= a	$\emptyset$ (*no bridges are created*)	$\{X$ #<= $\lfloor a \rfloor \mid a \in \mathbb{R}, (X$ #== $RX) \in B\}$
a <= RY	$\emptyset$ (*no bridges are created*)	$\{\lceil a \rceil$ #<= $Y \mid a \in \mathbb{R}, (Y$ #== $RY) \in B\}$
t_1 == t_2	$\{X$ #== $RX \mid$ either t_1 is an integral real constant and t_2 is a variable RX with no bridges in B (or vice versa) and X is new$\}$	$\{t_1^{\mathcal{FD}}$ == $t_2^{\mathcal{FD}} \mid$ For $1 \leqslant i \leqslant 2$: either t_i is an integral real constant n and $t_i^{\mathcal{FD}}$ is the integer n, or else t_i is a variable RX_i with $(X_i$ #== $RX_i) \in B$, and $t_i^{\mathcal{FD}}$ is $X_i\}$
t_1 /= t_2	$\emptyset$ (*no bridges are created*)	$\{t_1^{\mathcal{FD}}$ /= $t_2^{\mathcal{FD}} \mid$ For $1 \leqslant i \leqslant 2$: either t_i is an integral real constant n and $t_i^{\mathcal{FD}}$ is the integer n, or else t_i is a variable RX_i with $(X_i$ #== $RX_i) \in B$, and $t_i^{\mathcal{FD}}$ is $X_i\}$
t_1 + t_2 $\to!$ t_3 (resp. $-$, $*$)	$\{X$ #== $RX \mid t_3$ is a variable RX with no bridge in B, X new, for $1 \leqslant i \leqslant 2$, t_i is either an integral real constant or a variable RX_i with bridge $(X_i$ #== $RX_i) \in B\}$	$\{t_1^{\mathcal{FD}}$ #+ $t_2^{\mathcal{FD}} \to! t_3^{\mathcal{FD}} \mid$ For $1 \leqslant i \leqslant 3$: $t_i^{\mathcal{FD}}$ is determined as in the previous case$\}$
t_1 / t_2 $\to!$ t_3	$\emptyset$ (*no bridges are created*)	$\{t_2^{\mathcal{FD}}$ #* $t_3^{\mathcal{FD}} \to! t_1^{\mathcal{FD}} \mid$ For $1 \leqslant i \leqslant 3$ is determined as in the previous case$\}$

where B and Π' are interpreted as conjunctions. Note that the *correctness condition* $Sol_{\mathcal{C}}(\pi \wedge B) \supseteq Sol_{\mathcal{C}}(\exists \overline{V'}(\pi \wedge B \wedge \Pi'))$ also holds trivially.

Example 9 illustrates the operation of the goal transformation rules from Table 4 for computing bridges and projections with the help of the functions specified in Tables 5 and 6.

Example 9 (*Computation of Bridges and Projections*)

□ $\underline{(RX + 2 * RY) * RZ \text{ <= } 3.5}$ □ X #== RX, Y #== RY, Z #== RZ □□□ $\Vdash_{\mathbf{FC^3,PC^3}}$

$\exists RC, RB, RA.$ □ $\underline{2 * RY \ \to! \ RC}$, $\underline{RX + RC \ \to! \ RB}$, $\underline{RB * RZ \ \to! \ RA}$, RA <= 3.5 □

X #== RX, Y #== RY, Z #== RZ □□□ $\Vdash_{\mathbf{SB^3}}$

$\exists C, B, A, RC, RB, RA.$ □ $\underline{2 * RY \ \to! \ RC}$, $\underline{RX + RC \ \to! \ RB}$, $\underline{RB * RZ \ \to! \ RA}$, $\underline{RA \text{ <= } 3.5}$ □

C #== RC, B #== RB, A #== RA, X #== RX, Y #== RY, Z #== RZ □□□ $\Vdash_{\mathbf{PP^4}}$

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

$\exists C, B, A, RC, RB, RA.\ \Box\ \underline{2 * RY\ \ \rightarrow!\ RC},\ \underline{RX + RC\ \ \rightarrow!\ RB},\ \underline{RB * RZ\ \ \rightarrow!\ RA},\ \underline{RA <= 3.5}\ \Box$
C #== RC, B #== RB, A #== RA, X #== RX, Y #== RY, Z #== RZ $\Box\Box$
2 #* $Y \rightarrow!\ C$, X #+ $C \rightarrow!\ B$, B #* $Z \rightarrow!\ A$, A #<= 3 $\Box \Vdash_{\mathbf{SC4}}$
$\exists C, B, A, RC, RB, RA.\ \Box\Box$ C #== RC, B #== RB, A #== RA, X #== RX, Y #== RY, Z #== RZ $\Box\Box$
2 #* $Y \rightarrow!\ C$, X #+ $C \rightarrow!\ B$, B #* $Z \rightarrow!\ A$, A #<= 3 $\Box$
$2 * RY \rightarrow!\ RC$, $RX + RC \rightarrow!\ RB$, $RB * RZ \rightarrow!\ RA$, $RA <= 3.5$

Note that the initial goal in this current example is an extension of the initial goal in Example 8. The first six steps of the current computation are similar to those in Example 8, taking care of flattening the $\mathcal{R}$ constraint `(RX+2*RY)*RZ <= 3.5`. The subsequent steps use the transformation rules from Table 4 until no further bridges and projections can be computed and no constraints remain in the constraint pool.

We have borrowed the projection idea from Hofstedt's work (see, e.g., Hofstedt 2001; Hofstedt and Pepper 2007), but our proposal of using bridges to compute projections is a novelty. In Hofstedt's approach, projecting constraints from one domain into another depends on common variables present in both stores. In our approach, well-typing requirements generally prevent one and the same variable to occur in constraints from different domains. In order to improve the opportunities for computing projections, our cooperative goal-solving calculus $CCLNC(\mathcal{C})$ provides the goal transformation rule **SB** for creating new bridges during the computations. Some other differences between $CCLNC(\mathcal{C})$ and the cooperative computation model proposed by Hofstedt *et al.* are as follows:

- All the projections presented in this paper return just one $\exists\overline{V'}\,\Pi'$. In Hofstedt's terminology, such projections are called *weak*, while projections returning disjunctions $\bigvee_{k=1}^{l} \exists\overline{V'}_k\Pi'_k$ with $l > 1$ are called *strong*. The use of strong projections is illustrated in Hofstedt and Pepper (2007) by means of a problem dealing with the computation of resistors that have a certain capacity. The strong projection used in this example is a finite disjunction of conjunctions of the form $X == x \wedge Y == y$ for various numeric values x and y. Solving this disjunction gives rise to an enumeration of solutions. In Estévez-Martín *et al.* (2007b), we have presented a solution of the resistors problem where an equivalent enumeration of solutions can be computed by the $\mathcal{FD}$ solver via backtracking, without building any strong projection. This is possible in our framework due to the presence of labeling constraints that are not used in the resistor example as presented in Hofstedt and Pepper (2007). Therefore, strong projections are not necessary for this particular example of cooperation between $\mathcal{FD}$ and $\mathcal{R}$. Theoretically, strong projections could be useful in other problems, and rule **PP** in our $CCLNC(\mathcal{C})$ calculus could be very straightforwardly adapted to work with strong projections. However, we decided not to do so because we are not aware of any useful extension to extend Tables 5 and 6 for computing strong projections. We could find no formulation of practical procedures for computing projections in Hofstedt and Pepper (2007) and related works, where all projections used in examples are presented in an *ad hoc* manner.

Table 7. *Rules for inferring $\mathcal{H}$ constraints from $\mathcal{M}$ constraints*

IE Infer Equalities

$\exists \overline{U}.\ P \Box C \ \Box\ X \,\texttt{\#==}\, RX,\ X' \,\texttt{\#==}\, RX,\ M \ \Box\ H \ \Box\ F \Box R \Vdash_{\mathbf{UB}}$
$\exists \overline{U}.\ P \ \Box\ C \ \Box\ X \,\texttt{\#==}\, RX,\ M \ \Box\ H \ \Box\ X \texttt{ == } X', F \Box R.$

$\exists \overline{U}.\ P \Box C \ \Box\ X \,\texttt{\#==}\, RX,\ X \,\texttt{\#==}\, RX',\ M \ \Box\ H \ \Box\ F \Box R \Vdash_{\mathbf{UB}}$
$\exists \overline{U}.\ P \ \Box\ C \ \Box\ X \,\texttt{\#==}\, RX,\ M \ \Box\ H \ \Box\ F \Box RX \texttt{==} RX', R.$

ID Infer Disequalities

$\exists \overline{U}.\ P \Box C \ \Box\ X \texttt{\#/==} u',\ M \ \Box\ H \ \Box\ F \Box R \Vdash_{\mathbf{ID}} \exists \overline{U}.\ P \ \Box\ C \ \Box\ M \ \Box\ H \ \Box\ X \texttt{/=} u, F \Box R$
if $u \in \mathbb{Z}$, $u' \in \mathbb{R}$ and $u \,\texttt{\#==}^{\mathcal{M}}\, u' \to true$.

$\exists \overline{U}.\ P \Box C \ \Box\ u \texttt{\#/==} RX,\ M \ \Box\ H \ \Box\ F \Box R \Vdash_{\mathbf{ID}} \exists \overline{U}.\ P \ \Box\ C \ \Box\ M \ \Box\ H \ \Box\ F \ \Box\ RX \texttt{/=} u', R$
if $u \in \mathbb{Z}$, $u' \in \mathbb{R}$ and $u \,\texttt{\#==}^{\mathcal{M}}\, u' \to true$.

- Currently, our $CCLNC(\mathcal{C})$ calculus projects $\mathcal{FD}$ (resp. $\mathcal{R}$) constraints from the pool C into the $\mathcal{R}$ store R (resp. $\mathcal{FD}$ store F). Hofstedt's proposal also allows to compute projections from constraints placed into the stores. In our previous paper (Estévez-Martín *et al.* 2007b), we have sketched a cooperative goal-solving calculus where an arbitrary coordination domain was assumed and projections could act over the constraints within constraint stores. In fact, the resistor problem mentioned in the previous item was solved in Estévez-Martín *et al.* (2007b) by making essential use of projections that acted over constraints within the $\mathcal{FD}$ and $\mathcal{R}$ stores. In the current paper, goal solving is restricted to the coordination domain $\mathcal{C} = \mathcal{M} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{R}$ and projections can be applied only to the constraints placed in the constraint pool. These two limitations correspond to the state of the current $\mathcal{TOY}$ implementation. In particular, projections acting over stored constraints are not yet handled because the current $\mathcal{TOY}$ system has no convenient mechanisms for processing the constraint stores handled by the underlying SICStus Prolog.
- Goal solving in $CCLNC(\mathcal{C})$ enjoys the soundness and completeness properties presented in Subsection 3.6. In our opinion, these are more elaborate than the soundness and completeness results provided in Hofstedt's work.

To finish this subsection, we present the goal transformation rules in Table 7, which can be used to infer $\mathcal{H}$ constraints from the $\mathcal{M}$ constraints placed in the store M. The inferred $\mathcal{H}$ constraints happen to be $\mathcal{FD}$- or $\mathcal{R}$-specific, according to the case, and can be placed in the corresponding store. Therefore, the rules in this group model domain cooperation mechanisms other than bridges and projections.

3.4 Constraint-solving rules

The presentation of $CCLNC(\mathcal{C})$ finishes with the constraint-solving rules displayed in Table 8. Rule **SF** models the detection of failure by a solver, and the other

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Table 8. *Rules for $\mathcal{M}$, $\mathcal{H}$, $\mathcal{FD}$, and $\mathcal{R}$ constraint solving*

MS $\mathcal{M}$-Constraint Solver (*glass-box*)

$$\exists \overline{U}.\ P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R \Vdash_{\mathbf{MS}} \exists \overline{Y'}, \overline{U}.\ (P \,\Box\, C \,\Box\, (\Pi' \,\Box\, \sigma_M) \,\Box\, H \,\Box\, F \,\Box\, R)@_{\mathcal{M}}\sigma'$$

If $pvar(P) \cap var(\Pi_M) = \emptyset$, $(\Pi_M \,\Box\, \sigma_M)$ is not solved, $\Pi_M \Vdash_{solve^{\mathcal{M}}} \exists \overline{Y'}(\Pi' \,\Box\, \sigma')$.

HS $\mathcal{H}$-Constraint Solver (*glass-box*)

$$\exists \overline{U}.\ P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R \Vdash_{\mathbf{HS}} \exists \overline{Y'}, \overline{U}.\ (P \,\Box\, C \,\Box\, M \,\Box\, (\Pi' \,\Box\, \sigma_H) \,\Box\, F \,\Box\, R)@_{\mathcal{H}}\sigma'$$

If $pvar(P) \cap odvar_{\mathcal{H}}(\Pi_H) = \emptyset$, $\mathcal{X} =_{def} pvar(P) \cap var(\Pi_H)$, $(\Pi_H \,\Box\, \sigma_H)$ is not χ-solved, $\Pi_H \Vdash_{solve^{\mathcal{H}}_{\mathcal{X}}} \exists \overline{Y'}(\Pi' \,\Box\, \sigma')$.

FS $\mathcal{FD}$-Constraint Solver (*black-box*)

$$\exists \overline{U}.\ P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R \Vdash_{\mathbf{FS}} \exists \overline{Y'}, \overline{U}.\ (P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, (\Pi' \,\Box\, \sigma_F) \,\Box\, R)@_{\mathcal{FD}}\sigma'$$

If $pvar(P) \cap var(\Pi_F) = \emptyset$, $(\Pi_F \,\Box\, \sigma_F)$ is not solved, $\Pi_F \Vdash_{solve^{\mathcal{FD}}} \exists \overline{Y'}(\Pi' \,\Box\, \sigma')$.

RS $\mathcal{R}$-Constraint Solver (*black-box*)

$$\exists \overline{U}.\ P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R \Vdash_{\mathbf{RS}} \exists \overline{Y'}, \overline{U}.\ (P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, (\Pi' \,\Box\, \sigma_R))@_{\mathcal{R}}\sigma'$$

If $pvar(P) \cap var(\Pi_R) = \emptyset$, $(\Pi_R \,\Box\, \sigma_R)$ is not solved, $\Pi_R \Vdash_{solve^{\mathcal{R}}} \exists \overline{Y'}(\Pi' \,\Box\, \sigma')$.

SF Solving Failure

$$\exists \overline{U}.\ P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R \Vdash_{\mathbf{SF}} \blacksquare$$

If S is the $\mathcal{D}$ store ($\mathcal{D}$ being $\mathcal{M}$, $\mathcal{H}$, $\mathcal{FD}$ or $\mathcal{R}$), $pvar(P) \cap odvar_{\mathcal{D}}(\Pi_S) = \emptyset$, $\mathcal{X} =_{def} pvar(P) \cap var(\Pi_S)$, $(\Pi_S \,\Box\, \sigma_S)$ is not χ-solved and $\Pi_S \Vdash_{solve^{\mathcal{D}}_{\mathcal{X}}} \blacksquare$. Note that $\mathcal{X} \neq \emptyset$ is possible only in the case $\mathcal{D} = \mathcal{H}$.

rules describe the possible transformation of a goal by a solver's invocation. Each time a new constraint from the pool is placed into its store by means of transformation **SC**, it is pragmatically convenient to invoke the corresponding solver by means of the rules in this table. The solvers for the four domains $\mathcal{M}$, $\mathcal{H}$, $\mathcal{FD}$, and $\mathcal{R}$ involved in the coordination domain $\mathcal{C}$ are considered. The availability of the $\mathcal{M}$ solver means that solving mediatorial constraints contributes to the cooperative goal-solving process, in addition to the role of bridges for guiding projections.

Let $\mathcal{D}$ be any of the four domains, and let Π be the set of constraints included in the $\mathcal{D}$ store in a given goal G with productions P. As explained in Subsection 2.4.1, each invocation $solve^{\mathcal{D}}(\Pi, \mathcal{X})$ depends on a set of critical variables $\mathcal{X} \subseteq cvar_{\mathcal{D}}(\Pi)$ which must be properly chosen. On the other hand, the goal invariants explained in Subsection 3.1 require that no produced variable is bound to a nonlinear pattern, and the *safe binding* condition satisfied by any solver ensures that a solver invocation never binds any variable $\mathtt{X} \in \mathcal{X}$, except to a constant.

Because of these reasons, the rules in Table 8 allow a solver invocation $solve^{\mathcal{D}}(\Pi, \mathcal{X})$ only if the following two conditions are satisfied:

(a) $pvar(P) \cap odvar_{\mathcal{D}}(\Pi) = \emptyset$.
Motivation: If this condition does not hold, for any choice of $\mathcal{X} \subseteq cvar_{\mathcal{D}}(\Pi)$ there is some variable $\mathtt{X} \in pvar(P) \setminus \mathcal{X}$, and the solver invocation could bind X to a nonlinear pattern.

(b) $\mathcal{X} = pvar(P) \cap var(\Pi)$.
Motivation: Because of condition (a), this $\mathcal{X}$ is a subset of $cvar_{\mathcal{D}}(\Pi)$, and the safe binding condition of solvers ensures that the invocation $solve^{\mathcal{D}}(\Pi, \mathcal{X})$ will bind no produced variable, except to a constant.

When $\mathcal{D}$ is not $\mathcal{H}$, we know from Section 2 that all the variables in Π can be assumed to be obviously demanded. Then $odvar_{\mathcal{D}}(\Pi) = var(\Pi)$, condition (a) becomes $pvar(P) \cap var(\Pi) = \emptyset$, (b) becomes $\mathcal{X} = \emptyset$, and $solve^{\mathcal{D}}(\Pi, \emptyset)$ can be abbreviated as $solve^{\mathcal{D}}(\Pi)$. The rules related to $\mathcal{M}$, $\mathcal{FD}$, and $\mathcal{R}$ in Table 8 assume the simplified form of condition (a), (b). The notations $\Pi \Vdash_{solve^{\mathcal{D}}_{\mathcal{X}}} \exists \overline{Y'}(\Pi' \Box \sigma')$ and $\Pi \Vdash_{solve^{\mathcal{D}}_{\mathcal{X}}} \blacksquare$ introduced in Subsection 2.4.1 are used to indicate the nondeterministic choice of an alternative returned by a successful $\mathcal{D}$ solver invocation and a failed $\mathcal{D}$ solver invocation, respectively. Note also the use of the notation $(\ldots)@_{\mathcal{D}}\sigma'$ explained near the end of Subsection 3.1.

At this point, we can precise the notion of *solved goal* as follows: a goal G is solved iff it has the form $\exists \overline{U}.\ \Box\ \Box\ M\ \Box\ H\ \Box\ F\ \Box\ R$ (with empty P and C) and the $CLNC(\mathcal{C})$-transformations in Tables 7 and 8 cannot be applied to G. The $CLNC(\mathcal{C})$-transformations in Tables 3 and 4 are obviously not applicable to solved goals, since they refer to P and C.

3.5 One example of cooperative goal solving

In order to illustrate the overall behavior of our cooperative goal-solving calculus, we present a $CCLNC(\mathcal{C})$ computation solving the goal **Goal 2** discussed in Subsection 1.2. The reader is referred to Figure 2 for a graphical representation of the problem and to Subsection 3.1 for a formulation of the goal and the expected solution in the particular case `d = 2`, `n = 4`. However, the solution is the same for any choice of positive integer values `d` and `n` such that `n = 2*d`, and here we will discuss the general case.

The $CCLNC(\mathcal{C})$ calculus leaves ample room for choosing a particular goal transformation at each step, so that many different computations are possible in principle. However, the $\mathcal{TOY}$ implementation follows a particular strategy. The part $P \Box C$ of the current goal is treated as a sequence and processed from left to right, with the only exception of suspensions $e \rightarrow X$ that are delayed until they can be safely eliminated by means of rule **EL** or the goal is so transformed that they cease to be suspensions. As long as the current goal is not in solved form, a subgoal is selected and processing according to a strategy which can be roughly described as follows:

(1) If P includes some production which can be handled by the constrained lazy narrowing rules in Table 3, the leftmost such production is selected and processed. Note that the selected production must be either a suspension $e \to X$ that can be discharged by rule **EL**, or else a production that is not a suspension. The applications of rule **DF** are performed in an optimized way by using definitional trees (del Vado-Vírseda 2005, 2007).

(2) If P is empty or consists only of productions $e \to X$ that cannot be processed by means of the constrained lazy narrowing rules in Table 3, and moreover some of the stores M, H, F, or R is not in solved form and its constraints include no obviously demanded produced variables, then the solvers for such stores are invoked, choosing the set $\mathcal{X}$ of critical variables as explained in Table 8.

(3) If neither of the two previous items applies and C is not empty, the leftmost atomic constraint δ in C is selected. In case it is not primitive, the flattening rule **FC** from Table 3 is applied. Otherwise, δ is a primitive atomic constraint π, and exactly one of the following cases applies:

   (a) If π is a proper $\mathcal{FD}$ constraint or else an extended $\mathcal{H}$ constraint such that $M \vdash \pi$ `in` $\mathcal{FD}$, then π is processed by means of the rules **SB**, **PP,** and **SC** from Table 4. This generates bridges and projected constraints π', if possible, and submits π to the store F. Then, the rules from Table 8 are used for invoking the $\mathcal{FD}$ solver (in case that the constraints in F include no produced variables) and the $\mathcal{R}$ solver (in case that the constraints in R include no produced variables).

   (b) If π is a proper $\mathcal{R}$ constraint or else an extended $\mathcal{H}$ constraint such that $M \vdash \pi$ `in` $\mathcal{R}$, then π is processed by means of the rules **SB**, **PP,** and **SC** from Table 4. This generates bridges and projected constraints π', if possible, and submits π to the store R. Then, the rules from Table 8 are used for invoking the $\mathcal{R}$ solver (in case that the constraints in R include no produced variables) and the $\mathcal{FD}$ solver (in case that the constraints in F include no produced variables).

   (c) If π is an extended $\mathcal{H}$ constraint such that neither $M \vdash \pi$ `in` $\mathcal{FD}$ nor $M \vdash \pi$ `in` $\mathcal{R}$, then π is submitted to the store H by means of rule **SC**, and the $\mathcal{H}$ solver is invoked in case that the constraints in H include no obviously demanded produced variables.

   (d) If π is a $\mathcal{M}$ constraint, then π is submitted to the store M by means of rule **SC**, the rules of Table 7 are applied if possible, and the $\mathcal{M}$ solver is invoked in case that the constraints in M include no produced variables.

The series of goals G_0 up to G_{12} displayed below correspond to the initial goal, the final solved goal, and a selection of intermediate goals in a computation which roughly models the strategy of the $\mathcal{TOY}$ implementation, working with the projection functionality activated. In the initial goal, `d` and `n` are arbitrary positive integers such that `n = 2*d` and `d' = d+0.5`.

$G_0 : \Box\ \underline{bothIn\,(triangle\,(d,d')\,2\,1)\,(square\,n)\,(X,Y) == \texttt{true}}\ \Box\Box\Box\Box \Vdash_{\mathbf{FC}}$

$G_1 : \exists\overline{U_1}.\ bothIn\,(triangle\,(d,d')\,2\,1)\,(square\,n)\,(X,Y) \to A\ \Box\ \underline{A == \texttt{true}}\ \Box\Box\Box\Box \Vdash_{\mathbf{SC(ii)}}$

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

$G_2 : \exists\overline{U_2}.\ \underline{bothIn\,(triangle\,(d,d')\,2\,1)\,(square\,n)\,(X,Y)} \rightarrow A \ \Box\ \Box\Box\ A \texttt{ == true}\Box\Box \Vdash_{\mathbf{DF}_{bothIn}}$

$G_3 : \exists\overline{U_3}.\underline{triangle\,(d,d')\,2\,1 \rightarrow R},\ \underline{square\,n \rightarrow G},\ \underline{(X,Y) \rightarrow (X',Y')},\ \underline{\texttt{true} \rightarrow A}\ \Box$
$\quad \underline{X' \texttt{ #== } RX,\ Y' \texttt{ #== } RY,\ isIn\,R\,(RX,RY) \texttt{ == true},\ isIn\,G\,(X',Y') \texttt{ == true}},$
$\quad labeling\,[\,]\,[X',Y']\ \Box\Box\ \underline{A \texttt{ == true}}\ \Box\Box \Vdash^{*}_{\mathbf{SP^2,DC,SP^3,HS}}$

$G_4 : \exists\overline{U_4}.\ \Box\underline{X \texttt{ #== } RX},\ \underline{Y \texttt{ #== } RY},\ isIn\,(triangle\,(d,d')\,2\,1)\,(RX,RY) \texttt{ == true},$
$\quad isIn\,(square\,n)\,(X,Y) \texttt{ == true},\ labeling\,[\,]\,[X,Y]\ \Box\Box\ \sigma_H\ \Box\Box \Vdash^{*}_{\mathbf{SC(i)^2,MS}}$

$G_5 : \exists\overline{U_5}.\ \Box\ \underline{isIn\,(triangle\,(d,d')\,2\,1)\,(RX,RY) \texttt{== true}, isIn\,(square\,n)\,(X,Y) \texttt{== true}},$
$\quad labeling\,[\,]\,[X,Y]\ \Box X \texttt{ #== } RX,\ Y \texttt{ #== } RY \Box\ \sigma_H\ \Box\Box \Vdash^{*}_{\mathbf{CLN}}$

$G_6 : \exists\overline{U_6}.\ \Box\ \underline{RY >= d' - 1},\ 2 * RY - 2 * 1 * RX <= 2 * d' - 2 * 1 * d,$
$\quad 2 * RY + 2 * 1 * RX <= 2 * d' + 2 * 1 * d,\ domain\,[X,Y]\,0\,n,$
$\quad labeling\,[\,]\,[X,Y]\ \Box\ X \texttt{ #== } RX,\ Y \texttt{ #== } RY \Box\ \sigma'_H\ \Box\Box \Vdash^{*}_{\mathbf{FC,PC}}$

$G_7 : \exists\overline{U_7}.\ \Box\ \underline{d' - 1\ \rightarrow !RA},\ RY >= RA,\ 2 * RY - 2 * 1 * RX <= 2 * d' - 2 * 1 * d,$
$\quad 2 * RY + 2 * 1 * RX <= 2 * d' + 2 * 1 * d,\ domain\,[X,Y]\,0\,n,$
$\quad labeling\,[\,]\,[X,Y]\ \Box\ X \texttt{ #== } RX,\ Y \texttt{ #== } RY \Box\ \sigma'_H\ \Box\Box \Vdash^{*}_{\mathbf{SC(iv),RS}}$

$G_8 : \exists\overline{U_8}.\ \Box\ \underline{RY >= d''},\ 2 * RY - 2 * 1 * RX <= 2 * d' - 2 * 1 * d,$
$\quad 2 * RY + 2 * 1 * RX <= 2 * d' + 2 * 1 * d,\ domain\,[X,Y]\,0\,n,$
$\quad labeling\,[\,]\,[X,Y]\ \Box\ X \texttt{ #== } RX,\ Y \texttt{ #== } RY \Box\ \sigma'_H\ \Box\Box\ S_R \Vdash^{*}_{\mathbf{BP,CS}}$

$G_9 : \exists\overline{U_9}.\ \Box\ \underline{2 * RY - 2 * 1 * RX <= 2 * d' - 2 * 1 * d,}$
$\quad \underline{2 * RY + 2 * 1 * RX <= 2 * d' + 2 * 1 * d},\ domain\,[X,Y]\,0\,n,\ labeling\,[\,]\,[X,Y]\ \Box$
$\quad X \texttt{ #== } RX,\ Y \texttt{ #== } RY\ \Box\ \sigma'_H\ \Box\ Y\# >= d\ \Box\ RY >= d'', S_R \Vdash^{*}_{\mathbf{FR,BP}}$

$G_{10} : \exists\overline{U_{10}}.\ \Box\ domain\,[X,Y]\,0\,n,\ labeling\,[\,]\,[X,Y]\ \Box$
$\quad \underline{X \texttt{ #== } RX,\ Y \texttt{ #== } RY\ \ B \texttt{ #== } RB,\ C \texttt{ #== } RC,\ S'_M}\ \Box\ \sigma'_H\ \Box$
$\quad \underline{Y\# >= d, 2\# * Y\# - 2\# * X \rightarrow !B, B\# <= 1, 2\# * Y\# + 2\# * X \rightarrow !C, C\# <= n', S'_F}\ \Box$
$\quad \underline{RY >= d'', 2 * RY - 2 * RX \rightarrow !RB, RB <= 1, 2 * RY + 2 * RX \rightarrow !RC, RC <= n', S'_R} \Vdash^{*}_{\mathbf{CS}}$

$G_{11} : \exists\overline{U_{11}}.\ \Box\ \underline{domain\,[d,d]\,0\,n},\ \underline{labeling\,[\,]\,[d,d]}\ \Box\ S''_M\ \Box\ \sigma'_H\ \Box\ S''_F\ \Box\ S''_R \Vdash^{*}_{\mathbf{SC(iii),FS,SC(iii),FS}}$

$G_{12} : \exists\overline{U_{12}}.\ \Box\ \Box\ S''_M\ \Box\ \sigma'_H\ \Box\ S''_F\ \Box\ S''_R$

The local existential variables $\exists\overline{U_i}$ of each goal G_i are not explicitly displayed, and the notation $G_{i-1} \Vdash^{*}_{\mathcal{RS}} G_i$ is used to indicate the transformation of G_{i-1} into G_i using the goal-solving rules indicated by $\mathcal{RS}$. At some steps, $\mathcal{RS}$ indicates a particular sequence of individual rules, named as explained in the previous subsections. In other cases, namely for $i = 6$ and $9 \leqslant i \leqslant 11$, $\mathcal{RS}$ indicates sets of goal transformation rules, named according to the following conventions:

- **CLN** names the set of constrained lazy narrowing rules presented in Table 3.
- **FR** names the set consisting of the two rules **FC** and **PC** displayed at the end of Table 3, used for constraint flattening.
- **BP** names the set of rules for bridges and projections presented in Table 4.
- **CS** names the set of constraint-solving rules presented in Table 8.

We finish with some comments on the computation steps:

- Transition from G_0 to G_1: The only constraint in C is flattened, giving rise to one suspension and one flat constraint in the new goal. The produced variable `A` is not obviously demanded because the constraint `A == true` is not yet placed in the $\mathcal{H}$ store.
- Transition from G_1 to G_2: The only suspension is delayed, and the only constraint in the pool is processed by submitting it to the $\mathcal{H}$ store. However,

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

the $\mathcal{H}$ solver cannot be invoked at this point, because A has become an obviously demanded variable that is also produced.

- Transition from G_2 to G_3: The former suspension has become a production which is processed by applying the program rule defining the function `bothIn`, which introduces new productions in P and new constraints in C.
- Transition from G_3 to G_4: The four productions in P are processed by binding propagations and decompositions (rules **SP** and **DC**), until P becomes empty. Then the $\mathcal{H}$ solver can be invoked. At this point, the $\mathcal{H}$ store just contains a substitution σ_H resulting from the previous binding steps.
- Transition from G_4 to G_5: P is empty, and the two first constraints in C are bridges. They are submitted to the $\mathcal{M}$ store and the $\mathcal{M}$ solver is invoked, which has no effect in this case.
- Transition from G_5 to G_6: There are no productions, and the two first constraints in the pool are processed by steps similar to those used in the transition going from G_0 to G_4. Upon completing this process, the new pool includes a number of new constraints coming from the conditions in the program rules defining the functions `isIn`, `triangle` and `square`, and the substitution stored in H has changed. At this point, P is empty again and the constraints in C plus the bridges in M amount to a system equivalent to the one used in Subsection 1.2 for an informal discussion of the resolution of **Goal 2**.
- Transition from G_6 to G_7 and from G_7 to G_8: There are no productions, and flattening the first constraint in C gives rise to the primitive constraint `d'-1` $\to!$ `RA`. This is submitted to the $\mathcal{R}$ store and the $\mathcal{R}$ solver is invoked, which computes `d''` as the numeric value of `d'-1` and propagates the variable binding `RA` $\mapsto$ `d''` to the whole goal, possibly causing some other internal changes in the $\mathcal{R}$ store.
- Transition from G_8 to G_9: There are no productions, and the first constraint in C is now `RY >= d''`. Since `d'' = d'-1 = d+0.5-1 = d-0.5`, we have $\lceil$`d''`$\rceil$ `= d`. Therefore, projecting `RY >= d''` with the help of the available bridges (including `Y #== RY`) allows to compute `Y #>= d` as a projected $\mathcal{FD}$ constraint. Both `RY >= d''` and `Y #>= d` are submitted to their respective stores and the two solvers are invoked, having no effect in this case.
- Transition from G_9 to G_{10}: There are no productions, and the two first atomic constraints in the pool of G_9 (two $\mathcal{R}$ constraints δ_1 and δ_2) are processed by steps similar to those used in the transition going from G_6 to G_9, except that the solver invocations are delayed to the transition from G_{10} to G_{11} and commented in the next item. (Actually, the $\mathcal{TOY}$ implementation would invoke the solvers two times: the first time when processing δ_1 and the second time when processing δ_2. Here we explain the overall effect of the two invocations for the sake of simplicity.) Upon completing this process, G_{10} stays as follows: P is empty, C includes the two other constraints which were there in G_9, and the stores M, F, and R have changed because of new bridges and projections. In fact, the constraints within the stores F and R in G_{10} would be equivalent but not identical to the ones shown in this presentation due to

intermediate flattening steps that we have not shown explicitly. In particular, the $\mathcal{R}$ constraint `2*RY-2*RX` $\rightarrow!$ `RB` and its $\mathcal{FD}$-projection `2#*Y#-2#*X` $\rightarrow!$ `B` would really not occur in this form, but a conjunction of primitive constraints obtained by flattening them would occur at their place.

- Transition from G_{10} to G_{11}: At this point, the $\mathcal{FD}$ solver is able to infer that the constraints in the $\mathcal{FD}$ store imply one single solution for the variables `X` and `Y`, namely $\{$`X` $\mapsto$ `d, Y` $\mapsto$ `d`$\}$. Therefore, the $\mathcal{FD}$ solver propagates these bindings to the whole goal, affecting in particular to the bridges in M. Then, the $\mathcal{M}$ solver propagates the corresponding bindings $\{$`RX` $\mapsto$ `rd, RY` $\mapsto$ `rd`$\}$. (`rd` being the representation of `d` as an integral real number), and the $\mathcal{R}$ solver succeeds.
- Transition from G_{11} to G_{12}: The two constraints in C have now become trivial. Submitting them to their stores and invoking the respective solvers leads to a solved goal, whose restriction to the variables in the initial goal is the computed answer $\Box\Box\Box\Box$ ($\diamond$ $\Box$ $\{$`X` $\mapsto$ `d, Y` $\mapsto$ `d`$\}$) $\Box$. Note that no labeling whatsoever has been performed, independently of the size of `n`.

3.6 Properties of the cooperative goal-solving calculus $CCLNC(\mathscr{C})$

This final subsection presents the main semantic results of the paper, namely *soundness* and *limited completeness* of the cooperative goal-solving calculus $CCLNC(\mathscr{C})$ w.r.t. the declarative semantics of $CFLP(\mathscr{C})$ given in López-Fraguas *et al.* (2007). To start with, we define the notion of solution for a given goal.

Definition 10 (*Solutions of Goals and Their Witnesses*)

(1) Let $G \equiv \exists\overline{U}.\ P \Box C \Box M \Box H \Box F \Box R$ be an admissible goal for a given $CFLP(\mathscr{C})$-program $\mathscr{P}$. The *set of solutions* $Sol_{\mathscr{P}}(G)$ of G w.r.t. $\mathscr{P}$ includes all those $\mu \in Val_{\mathscr{C}}$ such that there is some $\mu' \in Val_{\mathscr{C}}$ verifying $\mu' =_{\setminus\overline{U}} \mu$ and $\mu' \in Sol_{\mathscr{P}}(P \Box C \Box M \Box H \Box F \Box R)$, which holds iff the following two conditions are satisfied:

  (a) $\mu' \in Sol_{\mathscr{P}}(P \Box C)$. By definition, this means $\mathscr{P} \vdash_{CRWL(\mathscr{C})} (P \Box C)\mu'$, which is equivalent to $\mathscr{P} \vdash_{CRWL(\mathscr{C})} P\mu'$ and $\mathscr{P} \vdash_{CRWL(\mathscr{C})} C\mu'$. This notation refers to the existence of proofs in the instance $CRWL(\mathscr{C})$ of the constrained rewriting logic $CRWL$, whose inference rules can be found in López-Fraguas *et al.* (2007).

  (b) $\mu' \in Sol_{\mathscr{C}}(M \Box H \Box F \Box R)$, which is equivalent to $\mu' \in Sol_{\mathscr{C}}(M) \cap Sol_{\mathscr{C}}(H) \cap Sol_{\mathscr{C}}(F) \cap Sol_{\mathscr{C}}(R)$.

(2) If $\mathscr{M}$ is a multiset having as its members the $CRWL(\mathscr{C})$-proofs mentioned in item (1)(a) above, we will say that $\mathscr{M}$ is a *witness* for the fact that $\mu \in Sol_{\mathscr{P}}(G)$, and we will write $\mathscr{M} : \mu \in Sol_{\mathscr{P}}(G)$.

(3) A solution $\mu \in Sol_{\mathscr{P}}(G)$ is called *well-typed* iff the valuation $\mu' =_{\setminus\overline{U}} \mu$ mentioned in item (1) can be so chosen that $(P \Box C \Box M \Box H \Box F \Box R)\mu'$ is well-typed, which is noted as $\mu' \in WTSol_{\mathscr{P}}(P \Box C \Box M \Box H \Box F \Box R)$. The set of all well-typed solutions of G w.r.t. $\mathscr{P}$ is written as $WTSol_{\mathscr{P}}(G)$. In case that $\mathscr{M}$ is a witness for $\mu \in Sol_{\mathscr{P}}(G)$, we also say that $\mathscr{M}$ is a witness for $\mu \in WTSol_{\mathscr{P}}(G)$ and we write $\mathscr{M} : \mu \in WTSol_{\mathscr{P}}(G)$.

In case that G is a solved goal S, we write $Sol_{\mathscr{C}}(S)$ (resp. $WTSol_{\mathscr{C}}(S)$) in place of $Sol_{\mathscr{P}}(S)$ (resp. $WTSol_{\mathscr{P}}(S)$).

Concerning item (1)(b) in the previous definition, note that the equivalence $\eta \in Sol_{\mathscr{C}}(M) \cap Sol_{\mathscr{C}}(H) \cap Sol_{\mathscr{C}}(F) \cap Sol_{\mathscr{C}}(R) \Leftrightarrow \eta \in Sol_{\mathscr{M}}(M) \cap Sol_{\mathscr{H}}(H) \cap Sol_{\mathscr{F}}(F) \cap Sol_{\mathscr{R}}(R)$ does not make sense in general, because a given valuation $\eta \in Val_{\mathscr{C}}$ is not always a $\mathscr{D}$ valuation when $\mathscr{D}$ is chosen as one of the four components of $\mathscr{C}$. However, Theorem 2 from Subsection 2.5 allows to reason with solutions known for $\mathscr{C}$ in terms of solutions known for the four components, as we will see in the mathematical proofs of Appendix A.2.

Before presenting our soundness and completeness results for $CCLNC(\mathscr{C})$ let us comment on some limitations concerning completeness:

- As already said in Subsection 3.1, the design of $CCLNC(\mathscr{C})$ is tailored to programs and goals having no free occurrences of higher-order logic variables. Therefore, the completeness results of this subsection are limited to this kind of programs and goals.
- The completeness of $CCLNC(\mathscr{C})$ is obviously conditioned by the completeness of the solvers invoked by the goal transformation rules in Table 8. On the other hand, the completeness requirement for solvers in Definition 6 is limited to well-typed solutions. Therefore, the completeness results of this subsection refer only to well-typed solutions of the initial goal.
- As discussed in Subsections 2.4.2, 2.4.3, and 2.4.4, certain invocations of constraint solvers can be incomplete even w.r.t. well-typed solutions. Therefore, the completeness results of this subsection are also limited by the assumption that no incomplete solver invocations occur during goal solving.
- Finally, the goal transformation rule **DC** from Table 3 can give rise to *opaque decompositions*. Similarly to the opaque decompositions caused by the transformation rules **H3** and **H7** for $\mathscr{H}$ stores (see Subsection 2.4.2), the opaque decompositions caused by **DC** can lose well-typed solutions. In what follows, we will say that an application of the goal transformation rule **DC** is *transparent* iff the expression $h\,\overline{e}_m$ involved in the rule application is such that h is m-transparent (or equivalently, h is not m-opaque). Only transparent applications of the rule **DC** can be trusted to preserve well-typed solutions. For this reason, the completeness results of this subsection are limited by the assumption that no opaque applications of rule **DC** occur during goal solving. Unfortunately, the eventual occurrence of opaque decomposition steps during goal solving (be they due to rule **DC** from Table 3 or to the *sts*s **H3** and **H7** of the $\mathscr{H}$ solver) is an undecidable problem, because of theoretical results proved in González-Moreno *et al.* (2001).

In the sequel, we will use the notation $G \Vdash_{\mathbf{RL},\gamma,\mathscr{P}} G'$ to indicate that the admissible goal G for the $CFLP(\mathscr{C})$-program $\mathscr{P}$ is transformed into the new goal G' by an application of the *selected rule* **RL** applied to the *selected part* γ of G. It is important to note that the selected part γ of G must have the form expected by the selected rule **RL**. More precisely, γ must be selected as one of the stores in case that **RL** is some transformations in Table 8, as a pair of bridges in case that **RL** is the

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

transformation **IE** from Table 7, and as an atom in any other case. We will use also the notation $G \Vdash^{+}_{\mathbf{RL},\gamma,\mathcal{P}} G'$ to indicate the existence of some computation of the form $G \Vdash_{\mathbf{RL},\gamma,\mathcal{P}} G_1 \Vdash^{*}_{\mathcal{P}} G'$ transforming G in G' in n steps for some $n \geqslant 1$.

We are now in a position to present the main results of this subsection. First, we state a theorem which guarantees *local* soundness and completeness for the *one-step* transformation of a given goal. A proof is given in Appendix A.2.

Theorem 4 (*Local Soundness and Limited Local Completeness*)
Assume a given program $\mathcal{P}$ and an admissible goal G for $\mathcal{P}$ which is not in solved form. Choose any rule **RL** applicable to G and select a part γ of G suitable for applying **RL**. Then there are finitely many possible transformations $G \Vdash_{\mathbf{RL},\gamma,\mathcal{P}} G'_j$ $(1 \leqslant j \leqslant k)$, and moreover:

(1) **Local Soundness:** $Sol_{\mathcal{P}}(G) \supseteq \bigcup_{j=1}^{k} Sol_{\mathcal{P}}(G'_j)$.
(2) **Limited Local Completeness:** $WTSol_{\mathcal{P}}(G) \subseteq \bigcup_{j=1}^{k} WTSol_{\mathcal{P}}(G'_j)$, provided that the application of **RL** to the selected part γ of G is *safe* in the following sense: it is neither an opaque application of **DC** nor an application of a rule from Table 8 involving an incomplete solver invocation.

A global soundness result for $CCLNC(\mathcal{C})$ follows easily from the first item of Theorem 4. In particular, 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.

Theorem 5 (*Soundness Theorem*)
Assume a $CFLP(\mathcal{C})$-program $\mathcal{P}$, an admissible goal G for $\mathcal{P}$, and a solved goal S such that $G \Vdash^{*}_{\mathcal{P}} S$. Then, $Sol_{\mathcal{C}}(S) \subseteq Sol_{\mathcal{P}}(G)$.

Proof
As an obvious consequence of Theorem 4 (item (1)), one gets $Sol_{\mathcal{P}}(G') \subseteq Sol_{\mathcal{P}}(G)$ for any G' such that $G \Vdash_{\mathcal{P}} G'$. From this, an easy induction shows that $Sol_{\mathcal{P}}(S) \subseteq Sol_{\mathcal{P}}(G)$ holds for each solved form S such that $G \Vdash^{*}_{\mathcal{P}} S$. Since $Sol_{\mathcal{P}}(S) = Sol_{\mathcal{C}}(S)$, the soundness result is proved. □

Note that the local completeness part (item (2)) of Theorem 4 also implies that failing goals have no solution; i.e., from a failing transformation step $G \Vdash_{\mathbf{RL},\mathcal{P}} \blacksquare$ we can conclude $WTSol_{\mathcal{P}}(G) = \emptyset$, provided that the application of **RL** is safe. However, a global completeness result for $CCLNC(\mathcal{C})$ does not immediately follow from item (2) of Theorem 4. For an arbitrarily given $\mu \in WTSol_{\mathcal{P}}(G)$, completeness needs to ensure a terminating $CCLNC(\mathcal{C})$ computation ending up with a solved form S such that $\mu \in WTSol_{\mathcal{C}}(S)$. According to Definition 10, $\mu \in WTSol_{\mathcal{P}}(G)$ implies the existence of a witness $\mathcal{M} : \mu \in WTSol_{\mathcal{P}}(G)$. In Appendix A.2, we have defined a *well-founded progress ordering* $\rhd$ between pairs $(G, \mathcal{M})$ formed by a goal G and a witness, and we have proved the following result:

Lemma 6 (*Progress Lemma*)
Consider an admissible goal G for a $CFLP(\mathcal{C})$-program $\mathcal{P}$ and a witness $\mathcal{M} : \mu \in WTSol_{\mathcal{P}}(G)$. Assume that neither $\mathcal{P}$ nor G have free occurrences of higher-order

variables, and that G is not in solved form. Then:

(1) There is some **RL** applicable to G which is not a failing rule.
(2) Assume any choice of a rule **RL** (not a failure rule) and a part γ of G, such that **RL** can be applied to γ in a safe manner. Then there is some finite computation $G \Vdash^{+}_{\mathbf{RL},\gamma,\mathscr{P}} G'$ such that:
   - $\mu \in WTSol_{\mathscr{P}}(G')$.
   - There is a witness $\mathscr{M}' : \mu \in WTSol_{\mathscr{P}}(G')$ verifying $(G, \mathscr{M}) \rhd (G', \mathscr{M}')$.

Using the former lemma, we can prove the following completeness result:

Theorem 6 (*Limited Completeness Theorem*)
Let an admissible goal G for a program $\mathscr{P}$ and a well-typed solution $\mu \in WTSol_{\mathscr{P}}(G)$ be given. Assume that neither $\mathscr{P}$ nor G have free occurrences of higher-order variables. Then, unless prevented by some unsafe rule application, one can find a $CCLNC(\mathscr{C})$-computation $G \Vdash^{*}_{\mathscr{P}} S$ ending with a goal in solved form S such that $\mu \in WTSol_{\mathscr{C}}(S)$.

Proof
The thesis of the theorem can be rephrased by writing $\mu \in WTSol_{\mathscr{P}}(S)$ in place of the equivalent condition $\mu \in WTSol_{\mathscr{C}}(S)$. The hypothesis allow us to choose a witness $\mathscr{M} : \mu \in WTSol_{\mathscr{P}}(G)$. In order to prove the rephrased thesis we reason by induction on the well-founded ordering $\rhd$ (see, e.g., Baader and Nipkow 1998 for an explanation of this proof technique). In case that G is a solved goal, the rephrased thesis holds trivially with S taken as G itself. In case that G is not solved, we apply the Progress Lemma 6 to $\mathscr{P}$ and $\mathscr{M} : \mu \in WTSol_{\mathscr{P}}(G)$ and we obtain a rule **RL** and a part γ of G such that **RL** can be applied to γ. Assuming that this rule application is a safe one, Lemma 6 also provides a finite computation $G \Vdash^{+}_{\mathbf{RL},\gamma,\mathscr{P}} G'$ such that there is a witness $\mathscr{M}' : \mu \in WTSol_{\mathscr{P}}(G')$ fulfilling $(G, \mathscr{M}) \rhd (G', \mathscr{M}')$. Since neither $\mathscr{P}$ nor G have free occurrences of higher-order variables, the same must be true for G'. By well-founded induction hypothesis we can then conclude that, unless prevented by some unsafe goal transformation step, one can find a computation $G' \Vdash^{*}_{\mathscr{P}} S$ ending with a goal in solved form S such that $\mu \in WTSol_{\mathscr{P}}(S)$. The desired computation is then $G \Vdash^{+}_{\mathbf{RL},\gamma,\mathscr{P}} G' \Vdash^{*}_{\mathscr{P}} S$. □

4 Implementation

This section sketches the implementation of the $CCLNC(\mathscr{C})$ computational model on top of the $\mathscr{TOY}$ system. The current implementation has evolved from older versions that supported the domains $\mathscr{H}$ and $\mathscr{R}$, but not yet $\mathscr{FD}$ and its cooperation with $\mathscr{H}$ and $\mathscr{R}$. We describe the architectural components of the current $\mathscr{TOY}$ system and briefly discuss the implementation of the main cooperation mechanisms provided by $CCLNC(\mathscr{C})$, namely bridges and projections. The reader is referred to Arenas *et al.* (2007) and Estévez-Martín *et al.* (2006, 2007c, 2008b) for more details.

Instead of using an abstract machine for running byte-code or intermediate code, $\mathscr{TOY}$ programs are compiled to and executed in Prolog, as in other related systems

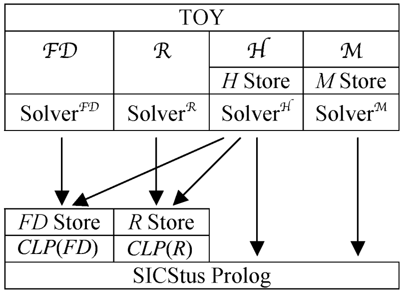

Fig. 4. Architectural components of the cooperation schema in $\mathcal{TOY}$.

(Antoy and Hanus 2000). The compilation generates Prolog code that implements goal solving by constrained lazy narrowing guided by *definitional trees*, a well-known device for ensuring an optimal behavior of lazy narrowing (Loogen et al. 1993; Antoy et al. 1994, 2000; del Vado-Vírseda 2003, 2005, 2007). $\mathcal{TOY}$ relies on an efficient Prolog system (SICStus Prolog 2007), which provides many libraries, including constraint solvers for the domains $\mathcal{FD}$ and $\mathcal{R}$.

$\mathcal{TOY}$ is distributed (`http://toy.sourceforge.net`) as a free open-source Sourceforge project and runs on several platforms. Installation is quite simple. Console and windows executables are provided, no further software is required. In addition, $\mathcal{TOY}$ can be used inside ACIDE (Sáenz-Pérez 2007), an emerging multiplatform and configurable integrated development environment (alpha development status).

4.1 Architectural components of the cooperation schema

Figure 4 shows the architectural components of the cooperation schema in $\mathcal{TOY}$. As explained in Subsection 2.6, the three pure constraint domains $\mathcal{H}$, $\mathcal{R}$, and $\mathcal{FD}$ are combined with a mediatorial domain $\mathcal{M}$ to yield the coordination domain $\mathcal{C} = \mathcal{M} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{R}$ which supports our cooperation model. Therefore, these four domains are supported by the implementation. Moreover, the set of primitives supported by the domains $\mathcal{R}$ and $\mathcal{FD}$ in the $\mathcal{TOY}$ implementation is wider than the simplified description given in Subsections 2.4.3 and 2.4.4.

The solvers and constraint stores for the domains $\mathcal{FD}$ and $\mathcal{R}$ are provided by the SICStus Prolog constraint libraries. The impedance mismatch problem among the host language constraint primitives and these solvers is tackled by glue code (see Subsection 4.2). Proper $\mathcal{FD}$ and $\mathcal{R}$ constraints, as well as Herbrand constraints specific to $\mathcal{FD}$ and $\mathcal{R}$ (see Subsections 2.4.4 and 2.4.3) are posted to the respective stores and handled by the respective SICStus Prolog solvers. On the other hand, the stores and solvers for the domains $\mathcal{H}$ and $\mathcal{M}$ are built into the code of the $\mathcal{TOY}$ implementation, rather than being provided by the underlying SICStus Prolog system.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

4.2 Implementing domain cooperation

This subsection explains the implementation of the fundamental mechanisms for domain cooperation: bridges and projections. The constraints provided by the mediatorial domain $\mathcal{M}$ and their semantics have been explained in Subsections 2.5 and 2.6. *Mediatorial constraints* have the general form a `#==` $b \rightarrow! c$, with a ::`int`, b :: `real` and c :: `bool`, while *bridges* a `#==` b and *antibridges* a `#/==` b abbreviate a `#==` $b \rightarrow!$ `true` and a `#==` $b \rightarrow!$ `false`, respectively.

In order to deal with $\mathcal{H}$ and $\mathcal{M}$ constraints, the $\mathcal{TOY}$ system uses a so-called *mixed store* which keeps a representation of the $\mathcal{H}$ and $\mathcal{M}$ stores as one single Prolog structure. It includes encodings of $\mathcal{H}$ constraints in solved form (i.e., totality constraints `X == X` and disequality constraints `X /= t`), as well as encodings of bridges and antibridges. The implementation of the $\mathcal{H}$ and $\mathcal{M}$ solvers in $\mathcal{TOY}$ is plugged into the Prolog code of various predicates which control the transformation of the mixed store (passed as argument) by means of two auxiliary arguments `Cin` and `Cout`.

In the next three subsections, we discuss the implementation of mediatorial constraints and projections. We will show and comment selected fragments of Prolog code, involving various predicates with auxiliary arguments `Cin` and `Cout`, as explained above. Regarding projections, the $\mathcal{TOY}$ implementation has been designed to support two modes of use: a "*disabled projections*" mode which allows to solve mediatorial constraints, but computes no projections; and an "*enabled projections*" mode which also computes projections. For each particular problem, the user can analyze the trade-off between communication flow and performance gain and decide the best option to execute a goal in the context of a given program.

4.2.1 *The equivalence primitive* `#==`

The equivalence primitive `#== :: int` $\rightarrow$ `real` $\rightarrow$ `bool` used for building mediatorial constraints is implemented as a Prolog predicate (also named `#==`) with five arguments, whose explanation follows. Arguments `L` and `R` stand for the left (integer) and right (real) parameters of the primitive `#==`. Argument `Out` stands for its result. Finally, arguments `Cin` and `Cout` stand for the state of the mixed store before and after performing a call to the primitive `#==`, respectively. Figure 5 shows the Prolog code for the predicate `#==`, and the comments below explain why this code implements the $\mathcal{M}$ solver described in Table 2 of Subsection 2.6 and the special cooperation rules **IE** and **ID** of the $CCLNC(\mathcal{C})$ calculus specified in Table 7 from Subsection 3.3.

Lines `(2)` and `(3)` compute the head normal forms (hnfs) of `L` and `R` into `HL` and `HR`, respectively. This process may generate new Herbrand constraints that will be added to the mixed store. The value of `HL` resp. `HR` will be either a variable or a number, ensuring that no suspensions will occur in the Prolog code from line `(4)` on. This code is intended to process the constraint `HL #== HR` $\rightarrow!$ `Out` according to the behavior of the $\mathcal{M}$ solver specified in Table 2, Subsection 2.6. Because of rules **M1** and **M2** in Table 2, the constraint is handled as a bridge `HL #== HR` when

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

```
(1) #==(L, R, Out, Cin, Cout):-
(2)     hnf(L, HL, Cin, Cout1),
(3)     hnf(R, HR, Cout1, Cout2),
(4)     tolerance(Epsilon),
(5)     ( (Out=true,
(6)           Cout3 = ['#=='(HL,HR)|Cout2],
(7)           freeze(HL, {HL - Epsilon =< HR, HR =< HL + Epsilon} ),
(8)           freeze(HR, (HL is integer(round(HR)))));
(9)       (Out=false,
(10)          Cout3 = ['#/=='(HL,HR)|Cout2],
(11)          freeze(HL, (F is float(HL), {HR =\= F})),
(12)          freeze(HR, (0.0 is float_fractional_part(HR) ->
(13)                   (I is integer(HR), HL #\= I); true)))),
(14)    cleanBridgeStore(Cout3,Cout).
```

Fig. 5. Implementation of mediatorial constraints (`#== / 2`).

`Out` equals `true`, and as an antibridge `HL #/== HR` when `Out` equals `false`. For this reason, one can say that the `#==` primitive accepts *reification*. Indeed, in Figure 5 we find that a bridge `HL #== HR` is posted to the mixed store if the value for `Out` can be unified with `true` (line `(6)`), whereas an antibridge `HL #/== HR` is posted if the value for `Out` can be unified with `false` (line `(10)`).

Solving both bridges and antibridges is accomplished by using the concurrent predicate `freeze`, which suspends the evaluation of its second argument until the first one becomes ground. Solving a bridge `HL #== HR` amounts to impose the equivalence of its two arguments (variables or constants), which are of different type (integer and real), so that type casting is needed. Variable `HL` is assigned to the integer version of `HR` (line `(8)`) via the Prolog functions `round` and `integer`, implementing rule **M3** in Table 2. Similarly, line `(7)` is roughly intended to assign the float version of `HL` to `HR` in order to implement rule **M5** in Table 2. However, due to the imprecise nature of real solvers, occasionally `HR`'s value will be an approximation to an integer value. Therefore, line `(7)` actually constrains the real variable `HR` to take a value between `HL - Epsilon` and `HL + Epsilon`, where `Epsilon` (line `(4)`) is a user-defined parameter (zero by default) which introduces a tolerance and avoids undesirable failures due to inexact computations of integer values. Lines `(7)` and `(8)` also cover the implementation of rule **M6** in Table 2. On the other hand, solving an antibridge `HL #/== HR` amounts to impose that both arguments are not equivalent. Therefore, as soon as `HL` or `HR` becomes bound to one numeric value, a disequality constraint between the (suitably type-casted) value of the bound variable and its mate argument is posted to the proper SICStus Prolog solver (lines `11-13`). The code in these lines implements rule **M8** in Table 2 and rule **ID** in Table 7.

Moreover, the failure rules in Table 2 (namely **M4**, **M7**, and **M9**) are also implemented by the frozen goals in lines `(7)-(8)` and `(11)-(13)` of Figure 5. Indeed, whenever `HL` and `HR` become bound, the corresponding frozen goal is triggered and the equivalence (resp. nonequivalence) is checked, which may yield to success or failure, thus implementing rules **M7** and **M9**; and wherever `HR` becomes bound to a nonintegral real value, the frozen goal in line `(8)` yields failure, thus

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

implementing rule **M4**. Finally, line (14)) invokes a predicate that simplifies the mixed store by implementing the effect of rule **IE** in Table 7 applied as much as possible to all the available (encodings of) bridges between variables.

4.2.2 Projection: $\mathcal{FD}$ to $\mathcal{R}$

If the user has enabled projections with the command `/proj`, the $\mathcal{TOY}$ system can process a given atomic primitive $\mathcal{FD}$ constraint by computing bridges and projected $\mathcal{R}$ constraints as explained in Subsection 3.3. The Prolog implementation has a different piece of code (Prolog clause) for each $\mathcal{FD}$ primitive which can be used to build projectable constraints. The information included in Table 5 for computing bridges and projections from different kinds of $\mathcal{FD}$ constraints, as well as the effect of the goal transformation rules in Table 4, is plugged into these pieces of Prolog code. The code excerpt below shows the basic behavior of the implementation for the case of $\mathcal{FD}$ constraints built with the inequality primitive `#<`, without considering optimizations:

```
(1) #<(L, R, Out, Cin, Cout):-
(2)     hnf(L, HL, Cin, Cout1), hnf(R, HR, Cout1, Cout2),
(3)     ((Out=true,  HL #<  HR); (Out=false, HL #>= HR)),
(4)     (proj_active ->
(5)        (searchVarsR(HL, Cout2, Cout3, RHL),
(6)         searchVarsR(HR, Cout3, Cout, RHR),
(7)        ((Out==true,  { RHL < RHR });
(8)         (Out==false, { RHL >= RHR })));
(9)       Cout=Cout2).
```

Following a technique similar to that explained for `#==` above, the primitive `#<` is implemented by a Prolog predicate with five arguments (line (1)). Its two input arguments (`L` and `R`) are reduced to hnf (line (2)), and a primitive constraint is posted to the SICStus $\mathcal{FD}$ solver, depending on the Boolean result (`Out`) returned by `#<` (line (3)). Moreover, if projection is active (indicated by the dynamic predicate `proj_active` in line (4)), then, the predicate `searchVarsR` (lines (5-6)) inspects the mixed store looking for bridges relating the $\mathcal{FD}$ variable `HL` and `HR` to the $\mathcal{R}$ variables `RHL` and `RHR`, respectively. In case that some of these variables are bound to a numeric variable, the relation to the mate variable just means that their numeric values are equivalent. Predicate `searchVarsR` also creates new bridges if necessary, according to the specifications in Table 5, and returns the modified state of the mixed store in its third argument. Finally, the projected constraints computed as specified in Table 5 (in this case, a single constraint, which is either `RHL < RHR` or `RHL >= RHR` depending on the value of `Out`) are sent to the SICStus $\mathcal{R}$ solver.

4.2.3 Projection: $\mathcal{R}$ to $\mathcal{FD}$

If the user has enabled projections, the $\mathcal{TOY}$ system can also process a given atomic primitive $\mathcal{R}$ constraint by computing bridges and projected $\mathcal{FD}$ constraints as explained in Subsection 3.3. The Prolog implementation is similar to that discussed in the previous subsection, with a different piece of code (Prolog clause) for each $\mathcal{R}$ primitive which can be used to build projectable constraints, and encoding the information from Table 6. A comparison between Tables 5 and 6 shows that there are less opportunities for building bridges from $\mathcal{R}$ to $\mathcal{FD}$ than the other way round, but more opportunities for building projections. The code excerpt below shows the basic behavior of the implementation for the case of $\mathcal{R}$ constraints built with the inequality primitive >, ignoring optimisations:

```
(1) >(L, R, Out, Cin, Cout):-
(2)  hnf(L, HL, Cin, Cout1), hnf(R, HR, Cout1, Cout),
(3)  (Out = true, {HR > HL} ; Out = false, {HL =< HR}),
(4)   (proj_active ->
(5)    (searchVarsFD(HL, Cout, BL, FDHL),
(6)     searchVarsFD(HR, Cout, BR, FDHR),
(7)     ((BL == true,  BR == true,  Out == true,  FDHL #>  FDHR);
(8)     (BL == true,  BR == true,  Out == false, FDHL #=< FDHR);
(9)     (BL == true,  BR == false, Out == true,  FDHL #>  FDHR);
(10)     (BL == true,  BR == false, Out == false, FDHL #=< FDHR);
(11)     (BL == false, BR == true,  Out == true,  FDHL #> FDHR);
(12)     (BL == false, BR == true,  Out == false, FDHL #=<  FDHR);
(13)      true); true).
```

As in the previous subsection, the primitive > is implemented by a Prolog predicate with five arguments (line `(1)`). Its two input arguments (`L` and `R`) are reduced to hnf (line `(2)`), and a primitive constraint is posted to the SICStus $\mathcal{R}$ solver, depending on the Boolean result (`Out`) returned by `>` (line `(3)`). Moreover, if projection is active (line `(4)`), then predicate `searchVarsFD` (lines `(5-6)`) inspects the mixed store looking for bridges relating the $\mathcal{R}$-variables `HL` and `HR` to $\mathcal{FD}$-variables. As shown in Table 6, no new bridges can be created during this process. Therefore, in contrast to the predicate `searchVarsR` presented in the previous subsection, the third argument of predicate `searchVarsFD` does not represent a modified state of the mixed store. Instead, it is a Boolean value that indicates whether a bridge has been found or not. More precisely, in line `(5)` there are two possibilities: either `BL` is `true` and `HL` is a nonbound $\mathcal{R}$-variable related to the $\mathcal{FD}$-variable `FDHL` by means of some bridge in the mixed store `Cout`; or else `BL` is `false`, `HL` is bound to a real value u, and `FDHL` is computed as $\lceil u \rceil$. Analogously, in line `(6)` there are two possibilities: either `BR` is `true` and `HR` is a nonbound $\mathcal{R}$-variable related to the $\mathcal{FD}$-variable `FDHR` by means of some bridge in the mixed store `Cout`; or else `BR` is `false`, `HR` is bound to a real value u, and `FDHR` is computed as $\lfloor u \rfloor$. Finally, lines `(7-12)` perform a distinction of cases corresponding to all the possiblities for projecting the constraint `HL > HR` $\rightarrow!$ `Out` according to Table 6 and the various values of `BL`, `BR`, and `Out`. In each case, the projected $\mathcal{FD}$ constraint is posted to the SICStus $\mathcal{FD}$ solver.

As a concrete example, when solving the conjunctive goal `X #== RX, RX > 4.3`, line `(11)` in the Prolog code for > just explained will eventually work for solving the right subgoal. In this case, viewing `RX` as `HL` and `4.3` as `HR`, the value computed for `BL` will be `true` because the bridge `X #== RX` will be available in the mixed store, and `FDHL` will be `X`. On the other hand, the value computed for `BR` will be `false`,

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

and the value of FDHR will be computed as ⌊4.3⌋, i.e., 4. Applying the proper case in Table 6, the projected constraint `X #> 4` will be posted to the SICStus $\mathcal{FD}$ solver.

5 Performance results

In this section, we study the performance of the systems $\mathcal{TOY}$ (Arenas *et al.* 2007; Estévez-Martín *et al.* 2006, 2007c, 2008b) and META-S (Frank *et al.* 2003a, 2003b, 2005), i.e., the closest related approach we are aware of when solving various problems requiring domain cooperation. After presenting a set of benchmarks in the first subsections, the following three subsections deal with an analysis of the benchmarks in each of the two systems and a comparison between both.

5.1 The benchmarks

We have selected a reasonably wide set of benchmarks which allows to analyze what happens when the set of constraints involved in the formulation of a programming problem is solved differently depending on the combination of domains that are involved in their solving. A concise description of the benchmarks is presented below.

- **Donald (donald)**: A cryptoarithmethic problem with 10 $\mathcal{FD}$ variables, 1 linear equation, and 1 *alldifferent* constraint. It consists of solving the equation DONALD + GERALD = ROBERT.
- **Send More Money (smm)**: Another cryptoarithmethic problem with 8 $\mathcal{FD}$ variables ranging over [0,9], 1 linear equation, 2 disequations, and 1 *alldifferent* constraint. It consists of solving the equation SEND + MORE = MONEY.
- **Nonlinear Crypto-Arithmetic (nl-csp)**: A problem with 9 $\mathcal{FD}$ variables and nonlinear equations.
- **Wrong-Wright (wwr)**: Another cryptoarithmethic problem with 8 $\mathcal{FD}$ variables ranging over [1,9], 1 linear equation, and 1 *alldifferent* constraint. It consists of solving the equation WRONG + WRONG = RIGHT.
- **3 × 3 Magic Square (mag.sq.)**: A problem that involves 9 $\mathcal{FD}$ variables and 7 linear equations.
- **Equation 10 (eq.10)**: A system of 10 linear equations with 7 $\mathcal{FD}$ variables ranging over [0,10].
- **Equation 20 (eq.20)**: A system of 20 linear equations with 7 $\mathcal{FD}$ variables ranging over [0,10].
- **Knapsack (knapsack)**: A classical knapsack problem taken from Hooker (2000). We considered two versions: one as a constraint satisfaction problem (labeled as **csp**) and another one as an optimisation one (labeled as **opt**).
- **Electrical Circuit (circuit)**: A problem taken from Hofstedt (2000a), in which one has an electric circuit with some connected resistors (i.e., $\mathcal{R}$ variables) and a set of capacitors (i.e., $\mathcal{FD}$ variables). The goal consists of knowing which capacitor has to be used so that the voltage reaches the 99% of the final voltage between a given time range.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

- **bothIn (goal2)**: The problem of solving the goal presented as **Goal 2** in Subsection 1.2 for several values of `n`. Instances `goal2(n)` of this benchmark correspond to solving an instance of **Goal 2** for the corresponding `n`.
- **bothIn (goal3)**: The problem of solving the goal presented as **Goal 3** in Subsection 1.2 for several values of `n`. Instances `goal3(n)` of this benchmark correspond to solving an instance of **Goal 3** for the corresponding `n`.
- **Distribution (distrib)**: An optimized distribution problem involving the cooperation of the domains $\mathcal{R}$ and $\mathcal{FD}$. The problem deals with a communication network where `NR` continuous and `ND` discrete suppliers of raw material have an attached cost to be minimized (see Appendix 8 in Arenas *et al.* 2007). The global optimum is computed. The various instances `distrib(ND,NR)` of this benchmark correspond to different choices of values for `ND` and `NR`.

All the benchmarks were coded using $\mathcal{FD}$ variables and most of them demand the solving of (non)linear equations. Only the last four of them strictly require cooperation between $\mathcal{FD}$ and $\mathcal{R}$ and cannot be solved by using just one of these domains. However, the rest of the benchmarks are also useful to evaluate the overhead introduced when the different solvers are enabled. The formulation of benchmarks **nl-csp**, **mag.sq**, **circuit,** and **smm** was taken from the distribution of META-S. Full details and code of the benchmarks (written in both $\mathcal{TOY}$ and META-S) are available at `http://www.lcc.uma.es/~afdez/cflpfdr/`.

All the benchmarking process was done using the same Linux machine (under the professional version of Suse Linux 9.3) with an Intel Pentium M processor running at 1.70 GHz and with an RAM memory of 1 GB. In the rest of this section, performance numbers, in milliseconds, have been computed as the average result of 10 runs for each benchmark. In all tables, the best result obtained for each benchmark among those computed under the various configurations has been highlighted in *boldface*.

5.2 Benchmark analysis in $\mathcal{TOY}$

In this section, we briefly present empirical support for two claims: (a) that the activation of the cooperation mechanism between $\mathcal{FD}$ and $\mathcal{R}$ does not penalize the execution time in problems which can be solved by using the domain $\mathcal{FD}$; and (b) that the cooperation mechanism using projections helps to speed up the execution time in problems where both the domain $\mathcal{FD}$ and the domain $\mathcal{R}$ are needed.

Tables 9–12 show the performance of each benchmark for several *configurations* of the $\mathcal{TOY}$ system, as explained below. The first column in each table displays the name of the benchmark to be solved, and the next columns correspond to different *activation modes* of the $\mathcal{TOY}$ system, namely:

- $\mathcal{TOY}(\mathcal{FD})$, an activation mode where the $\mathcal{FD}$ solver (but not the $\mathcal{R}$ solver) is enabled. Actually, this corresponds to an older version of the $\mathcal{TOY}$ system which did not provide simultaneous support for $\mathcal{R}$ constraints.
- $\mathcal{TOY}(\mathcal{FD} + \mathcal{R})$, an activation mode where both the $\mathcal{FD}$ solver and the $\mathcal{R}$ solver are enabled, but the projection mechanism is disabled.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Table 9. *Solving $\mathcal{FD}$ benchmarks in $\mathcal{TOY}$ (first solution search) (overload evaluation)*

	$\mathcal{FD}$ constraint solving					
	$\mathcal{TOY}(\mathcal{FD})$		$\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$		$\mathcal{TOY}(\mathcal{FD}+\mathcal{R})_p$	
Benchmark	naïve	ff	naïve	ff	naïve	ff
donald	1078	195	1040	**188**	7476	678
smm	16	15	**14**	16	47	49
nl-csp	**15**	20	**15**	18	39	86
wwr	**18**	19	**18**	19	58	52
maq.sq.	92	91	89	89	**87**	91
eq.10	**74**	90	**74**	81	284	261
eq.20	138	134	139	**131**	431	421
knapsack (csp)	**5**	**5**	**5**	**5**	**5**	**5**
knapsack (opt)	40	**15**	35	**15**	70	40

Table 10. *Solving $\mathcal{FD} \sim \mathcal{R}$ benchmarks in $\mathcal{TOY}$ (first solution search) (evaluation of the constraint projection mechanism)*

	$\mathcal{FD} \sim \mathcal{R}$ constraint solving			
	$\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$		$\mathcal{TOY}(\mathcal{FD}+\mathcal{R})_p$	
Benchmark	naïve	ff	naïve	ff
donald	304970	288700	8305	**727**
smm	22528	22627	41	**40**
nl-csp	411	383	**44**	87
wwr	411	420	**54**	58
maq.sq.	166	168	**158**	163
eq.10	**266**	271	290	269
eq.20	402	408	433	**397**
knapsack (csp)	**5**	**5**	**5**	**5**
knapsack (opt)	16	15	**11**	14

- $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})_p$, an activation mode where the $\mathcal{FD}$ solver, the $\mathcal{R}$ solver, and the projection mechanism are all enabled.

The heading "$\mathcal{FD}$ constraint solving" in Table 9 indicates that all the benchmarks have been formulated in such a way that all the constraints needed to solve them are submitted to the $\mathcal{FD}$ solver and the $\mathcal{R}$ solver is not invoked. Note that although the activation mode $\mathcal{TOY}(\mathcal{FD})$ is sufficient to execute all the benchmarks presented in this table, the benchmarks have also been executed in the modes $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$ and

Table 11. *Solving $\mathcal{FD} \sim \mathcal{R}$ benchmarks in $\mathcal{TOY}$ (first solution search) (evaluation of the constraint projection mechanism on benchmarks necessarily demanding solver cooperation)*

	$\mathcal{FD} \sim \mathcal{R}$ constraint solving			
	$\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$		$\mathcal{TOY}(\mathcal{FD}+\mathcal{R})_p$	
Benchmark	naïve	ff	naïve	ff
circuit	14	**13**	14	20
distrib (2,5.0)	662	506	**144**	504
distrib (3,3.0)	1486	810	**132**	814
distrib (3,4.0)	2098	1290	**156**	1178
distrib (4,5.0)	20444	12670	**240**	12744
distrib (5,2.0)	29108	5162	**198**	7340
distrib (5,5.0)	141734	85856	**272**	86497
distrib (5,10.0)	568665	464230	**474**	462980
goal2 (100)	25	28	**14**	**14**
goal2 (200)	40	44	**13**	15
goal2 (400)	70	72	**12**	13
goal2 (800)	131	135	**12**	15
goal2 (10000)	704	713	**14**	16
goal2 (20000)	1271	1270	**12**	16
goal2 (40000)	2325	2333	**11**	16
goal2 (80000)	4452	4472	**13**	16
goal2 (200000)	10725	10781	**13**	15
goal3 (100)	18	20	**15**	16
goal3 (200)	26	28	**13**	**13**
goal3 (400)	41	44	**15**	16
goal3 (800)	75	77	**16**	17
goal3 (5000)	354	360	**14**	16

Table 12. *Solving* **goal3(n)** *benchmarks in $\mathcal{TOY}$ (all solutions search)*

	$\mathcal{FD} \sim \mathcal{R}$ constraint solving			
	$\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$		$\mathcal{TOY}(\mathcal{FD}+\mathcal{R})_p$	
Benchmark	naïve	ff	naïve	ff
goal3 (100)	673	625	265	**242**
goal3 (200)	1867	1844	**329**	352
goal3 (400)	6527	6573	583	**579**
goal3 (800)	24460	24727	**976**	994
goal3 (5000)	911880	920670	**5365**	6135

$\mathcal{TOY}(\mathcal{FD}+\mathcal{R})_p$ with the aim of analyzing the overhead caused by the activation of these more complex modes when solving problems that do not need them.

The heading "$\mathcal{FD} \sim \mathcal{R}$ constraint solving" in Tables 10–12 indicates that the formulations of the benchmarks require both the $\mathcal{FD}$ solver and the $\mathcal{R}$ solver to be enabled; more precisely, although the benchmarks shown in Table 10 admit a natural formulation that can be totally solved by the $\mathcal{FD}$ solver, we have used an alternative formulation in which the (non)linear constraints were submitted to the $\mathcal{R}$ solver, whereas the rest of the constraints were sent to the $\mathcal{FD}$ solver; also solving the benchmarks shown in Tables 11 and 12 strictly requires cooperation between $\mathcal{FD}$ and $\mathcal{R}$. These tables only consider the two activation modes of the $\mathcal{TOY}$ system which make sense for such benchmarks, namely $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$ and $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})_p$.

Tables 9–12 also include two columns corresponding to two different *labeling strategies*: *naïve*, in which $\mathcal{FD}$ variables are labeled in a prefix order (i.e., the leftmost variable is selected); and *first fail (ff)*, in which the $\mathcal{FD}$ variable with the smallest domain is chosen first for enumerating. Combined with the distinct activation modes, this yields a number of configurations (i.e., six in Table 9 and four in the rest).

Inspection of Table 9 reveals that the performance of all the benchmarks does not get worse when moving from $\mathcal{TOY}(\mathcal{FD})$ to $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$ and $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})_p$, and it even improves in some cases. For those benchmarks that are most naturally coded in the domain $\mathcal{FD}$ (as, for instance, **smm**, **wwr,** and **mag.sq**) the best results are not those obtained in $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})_p$, but even in such cases the appreciable overload is not a great one.

Inspection of Tables 10 and 11 reveals that the projection mechanism causes a significant speedup of the solving process in most cases. Note that this mechanism behaves specially well in solving the **goal2(n)** and **goal3(n)** benchmarks, where the running time is stabilized in the range between 11 and 17 ms when projections are enabled. Significant speedups (i.e., at least two or more magnitude orders) are also detected in **donald** and **smm** benchmarks as well as in the different **distrib** benchmark instances.

Finally, Table 12 presents the results corresponding to computing all the results for the last five benchmarks in Table 11. The execution times are naturally higher than those shown in Table 11, where only first solutions were computed. However, the significant speedup caused by the activation of projections remains clearly observable.

5.3 Benchmark analysis in META-S

In this subsection, we present the results of executing benchmarks in META-S, a flexible meta-solver framework that implements the ideas proposed in Hofstedt (2001) and Hofstedt and Pepper (2007) for the dynamic integration of external stand-alone solvers to enable the collaborative processing of constraints. As already mentioned in Sections 1 and 3, the cooperative framework underlying META-S bears some analogies with the approach described in this paper. Both META-S and $\mathcal{TOY}$ provide means for different numeric constraint domains to cooperate. $\mathcal{TOY}$ supports cooperation between the domains $\mathcal{H}$, $\mathcal{FD}$, and $\mathcal{R}$, while META-S connects several kind of solvers, such as:

- An $\mathcal{FD}$ solver (for floats, strings, and rationals) that was implemented in Common Lisp using as reference a library of routines for solving binary constraint satisfaction problems provided by Peter van Beek and available from `http://www.ai.uwaterloo.ca/∼vanbeek/software/csplib.tar.gz`.
- A solver for linear arithmetic, i.e., the constraint solver LINAR described in Krzikalla (1997). This solver is based on the Simplex algorithm and was implemented in the language C. It handles linear equations, inequalities, and disequations over rational numbers.
- An interval arithmetic solver that uses the sound math library (available at `http://interval.sourceforge.net/interval/index.html`), an ANSI C library implemented on the basis of the solver for interval arithmetic of Timothy J. Hickey from Brandeis University (available from `http://www.cs.brandeis.edu/∼tim/`).

The interested reader is referred to Frank and Mai (2002) for more details on the META-S solvers. There are also some other significant differences between both systems. META-S is implemented in Common Lisp whereas $\mathcal{TOY}$ is implemented in Prolog. In contrast to $\mathcal{TOY}$, META-S does not support different activation modes (corresponding to $\mathcal{TOY}(\mathcal{FD})$, $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$, and $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})_p$ in $\mathcal{TOY}$), neither explicit labeling strategies, nor facilities for optimization. On the other hand, META-S supports the choice of different constraint-solving strategies (Frank *et al.* 2007), which is not the case in $\mathcal{TOY}$. More details regarding the comparison between $\mathcal{TOY}$ and META-S can be found in Subsection 5.4.

We have investigated the performance of META-S in solving the benchmarks already considered for $\mathcal{TOY}$ in the previous section and the performance results are shown in Tables 13–14. The organization of rows and columns is also similar to the $\mathcal{TOY}$ tables (but considering the two different strategies explained below). The occurrences of the symbol "–" indicate that the corresponding benchmark (namely, the knapsack optimization and the distribution problem) could not be executed because the META-S system provides no optimization facilities; the term "error" corresponds with a failure returned by the system that was not able to solve the goal. We have used the version 1.0 of META-S (kindly provided by its implementors on our request) compiled using SuSE Linux version 9.3 (professional version), based on CMU Common Lisp 18d.

For the META-S benchmarks, we have utilized the combination of the $\mathcal{FD}$ solver (usually for rationals) and an arithmetic solver which was found analogous to the $\mathcal{FD}$ plus $\mathcal{R}$ combination used in the corresponding $\mathcal{TOY}$ benchmark. In fact, for META-S, we have selected the linear arithmetic solver since the interval arithmetic solver yielded poorer results in all cases. In addition, we have considered the best problem formulation (in terms of the target solver for each constraint) that yielded

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Table 13. *Solving the benchmarks in META-S (first solution search)*

	META-S			
	Eager		Heuristic	
Benchmark	Standard	Ordered	Standard	Ordered
donald	268510	469370	**5290**	6140
smm	950	620	590	**580**
nl-csp	344800	1230	302314	**970**
wwr	10930	650	**620**	**620**
maq.sq.	1160	1220	**520**	540
eq.10	**60**	**60**	70	70
eq.20	**60**	**60**	70	70
knapsack (csp)	**60**	**60**	70	70
knapsack (opt)	–	–	–	–
distrib (2,5.0)	–	–	–	–
distrib (3,3.0)	–	–	–	–
distrib (3,4.0)	–	–	–	–
distrib (4,5.0)	–	–	–	–
distrib (5,2.0)	–	–	–	–
distrib (5,5.0)	–	–	–	–
distrib (5,10.0)	–	–	–	–
circuit	**70**	**70**	**70**	**70**
goal2 (100)	**330**	**330**	**330**	**330**
goal2 (200)	**730**	740	740	740
goal2 (400)	**2340**	**2340**	**2340**	2350
goal2 (800)	8550	**8540**	8560	8560
goal3 (100)	**410**	**410**	460	460
goal3 (200)	**900**	**900**	1080	1080
goal3 (400)	**2870**	2880	3520	3540
goal3 (800)	**10630**	10720	13140	13370

Table 14. *Solving* **goal3(n)** *benchmarks in META-S (all solutions search)*

	META-S			
	Eager		Heuristic	
Benchmark	Standard	Ordered	Standard	Ordered
goal3 (100)	8930	8880	**6940**	**6940**
goal3 (200)	**60700**	60870	47190	**46880**
goal3 (400)	**453330**	459980	**346930**	348900
goal3 (800)	error	error	error	error

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

the best running time. Moreover, we have executed each META-S benchmark under four different constraint-solving strategies:

- *Standard eager*, in which all constraint information is propagated as early as possible.
- *Ordered eager*, working as the previous one complemented with user-given information for determining the order of projection operations.
- *Standard heuristic*, working as the standard eager strategy complemented with an heuristic for giving priority to those variable bindings more likely to lead to failure.
- *Ordered heuristic*, working as the previous one complemented with user-given information for determining the order of projection operations.

In certain form, naïve and ff labeling in $\mathcal{TOY}$ are similar, respectively, to eager and heuristic strategies in META-S. For the sake of a fair comparison, whenever possible we have encoded the META-S benchmarks using exactly the same problem formulations as well as the same constraints that were used in the corresponding $\mathcal{TOY}$ benchmarks. Benchmarks were coded using the functional logic language FCLL of META-S. Also, we took care that the variable orders were identical for the different resolution/labeling strategies in both systems.

Note that the META-S benchmarks shown in Table 13 (resp. Table 14) correspond to the $\mathcal{TOY}$ benchmarks in Tables 9–11 (resp. Table 12), all of which refer to first solution search (resp. all solutions search).

5.4 $\mathcal{TOY}$ versus META-S

The tables displayed in this subsection are intended to compare the performance of $\mathcal{TOY}$ and META-S. Table 15 compares the behavior of both systems when computing the first solution of various benchmarks, while the results in Table 16 correspond to the computation of all the solutions for a few instances of the benchmark **goal3(n)**. More precisely, the execution times and META-S/$\mathcal{TOY}$ rates displayed in Table 15 correspond to the best results for each benchmark under those obtained for the various configurations in Tables 10–11 and 13, respectively; while Table 16 has been built from the information displayed in Tables 12 and 14 in a similar way.

The analogies and differences between the domain cooperation mechanisms supported by $\mathcal{TOY}$ and META-S have been discussed at the end of Subsection 3.3. In both cases, projections play a key role, and the information displayed in Tables 15 and 16 allows mainly to draw conclusions on the computational performance of both systems. META-S seems to behave particularly well in the solving of linear equations, especially when the problem requires no global constraints (such as an *alldifferent* constraint used in benchmarks **eq10** and **eq20**). The reason maybe twofold: first, that the linear arithmetic solver of META-S performs better than its $\mathcal{FD}$ solver, and, second, that flattening a nested constraint in $\mathcal{TOY}$ generates as many flat constraints as the number of operators it includes.

Table 15. *Solving benchmarks in 𝒯𝒪𝒴 versus META-S (first solution search)*

System	𝒯𝒪𝒴	META-S	META-S/𝒯𝒪𝒴
donald	188	5290	28.13
smm	14	580	41.42
nl-csp	15	970	64.66
wwr	18	620	34.44
maq.sq.	87	520	5.97
eq.10	74	60	0.81
eq.20	131	60	0.45
knapsack (csp)	5	60	12
knapsack (opt)	11	–	–
circuit	13	70	5.38
goal2 (100)	14	330	23.57
goal2 (200)	13	730	56.15
goal2 (400)	12	2340	195.00
goal2 (800)	12	8540	711.66
goal3 (100)	15	410	27.33
goal3 (200)	13	900	69.23
goal3 (400)	15	2870	191.33
goal3 (800)	16	10630	664.375

Table 16. *Solving* **goal3(n)** *in 𝒯𝒪𝒴 versus META-S (all solutions search)*

System	𝒯𝒪𝒴	META-S	META-S/𝒯𝒪𝒴
goal3 (100)	242	6940	28.67
goal3 (200)	329	46880	142.49
goal3 (400)	579	346930	599.18
goal3 (800)	976	error	–

However, in general, 𝒯𝒪𝒴 shows an improvement of about one order of magnitude with respect to the META-S system, for the benchmarks used in our comparison. As an extreme case, the computation time for obtaining the first solution of the benchmark **goal3(800)** increases more than three orders of magnitude with respect to 𝒯𝒪𝒴, and computing all the solutions for this benchmark in META-S does not succeed. In certain form, the experimental results suggest that our proposal is not only promising but also interesting in its current state.

In any case, the "superior" performance of 𝒯𝒪𝒴 with respect to META-S has to be interpreted carefully. One reason for 𝒯𝒪𝒴's advantage may be that the numerical solvers connected in the current version of META-S have been implemented just to experiment with the concepts of the underlying theoretical framework described in Hofstedt (2001) and Hofstedt and Pepper (2007), without much concern for optimization, while 𝒯𝒪𝒴 relies on the optimized solvers provided by SICStus

Prolog. Another advantage of $\mathcal{TOY}$ is the availability of global constraints such as *alldifferent* that are lacking in META-S. Admittedly, a better comparison of the performance results in both systems would be obtained by comparing independently the integrated solvers in each of the systems, and then normalizing the global results for the systems; or alternatively, by connecting the same solvers to both systems. This would be possible if all the integrated solvers were effectively black-boxes that can be unplugged from the systems. Unfortunately, this is not the case, as the solvers attached to $\mathcal{TOY}$ are used as provided by SICStus Prolog and they were not internally adjusted to work in a cooperation system, whereas the solvers used in META-S were implemented with regard to their integration into the implementation of META-S as a system with cooperating components.

In favor of META-S, we mention that the cooperation model proposed in META-S seems to be more flexible than the cooperation model currently implemented in $\mathcal{TOY}$, and provides facilities not yet available in $\mathcal{TOY}$. For instance, META-S allows to integrate and/or redefine evaluation strategies (Frank *et al.* 2003b), whereas $\mathcal{TOY}$ relies on a fixed strategy for goal solving and constraint evaluation. Also, the projection mechanism currently implemented in $\mathcal{TOY}$ is less powerful than in META-S, because projections cannot be applied to the constraints inside the constraint stores. Finally, META-S enables the integration of different host languages (Frank *et al.* 2005), whereas the $CCLNC(\mathcal{C})$ goal-solving calculus implemented in $\mathcal{TOY}$ is intended for declarative languages fitting the $CFLP$ scheme.

6 Related work

In this section, in addition to already mentioned related works, we extend the discussion to other proposals developed in the area of cooperative constraint solving. Of course, the issues of communication and cooperation are relevant to many aspects of computation. Here, we discuss a selection of the literature concerning proposals for communication and cooperation in constraint and declarative programming. Existing cooperative systems are very diverse and range from domain combinations to a mix of distinct techniques for solving constraints over the same domain. Moreover, the cooperating systems may be very different in nature: some of them perform complete constraint solving, whereas others just execute basic forms of propagation. In general, depending on the nature of the cooperation, we catalogue cooperative constraint solving in four nondisjoint categories:

(1) Cooperation of (built-in) domains coexisting in the same system.
(2) Interchange of information between different solvers/domains via special constructs.
(3) Interoperability or communication between independent solvers.
(4) Combination or integration of entities with distinct nature (i.e., methods and/or solvers based on different algorithms, or languages with different resolution mechanisms).

In the following four subsections we discuss some of the relevant work done in each of these categories, as well as their relation to our own approach.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

6.1 Cooperation of (built-in) domains coexisting in the same system

There are a number of constraint systems that provide support for the interaction between built-in and predefined domains. In these systems, a solver is viewed as a device that transforms the original set of constraints to an equivalent reduced set. As examples, we can cite the following systems:

- CLP(BNR) (Benhamou and Older 1997), Prolog III (Colmerauer 1990), and Prolog IV (N'Dong 1997) allow solver cooperation, mainly limited to Booleans, reals, and naturals (as well as term structures such as lists and trees).
- The language NCL (Zhou 2000) provides an integrated constraint framework that strongly combines Boolean logic, integer constraints, and set reasoning. Currently, NCL also integrates efficient CP domain cutting techniques and OR algorithms.

Most existing systems of this kind have two main problems: first, the cooperation is restricted to a limited set of computation domains supported by the system; and second, the cooperation mechanism is very dependant on the involved computation domains and thus presents difficulties to be generalized to other computation domains.

Our computational model for the cooperation of the domains $\mathcal{H}$, $\mathcal{FD}$, and $\mathcal{R}$ and its current $\mathcal{TOY}$ implementation can be catalogued in this category, and insofar it shares the two limitations just mentioned. However, our approach is more general because it is based on a generic scheme for *CFLP* programming over a parametrically given coordination domain $\mathcal{C}$. The cooperative goal-solving calculus $CLNC(\mathcal{C})$ presented in Section 3 refers to the particular coordination domain $\mathcal{C} = \mathcal{M} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{R}$, but it can be easily extended to other coordination domains, as sketched in our previous paper (Estévez-Martín *et al.* 2007b).

6.2 Interchange of information between computation domains/or solvers via special constructs

Another cooperation technique consists of providing special built-in constructs designed to propagate information among different computation domains that coexist in the same system. For example, this is the case with the reified constraints that enable a communication between arithmetic computations and a Boolean domain.

Within this type of cooperation we can cite *Conjunto* (Gervet 1997), a constraint language for propagating interval constraints defined over finite sets of integers. This language provides so-called *graduated constraints* which map sets onto arithmetic terms, thus allowing a one-way cooperative channel from the set domain to the integer domain. Graduated constraints can be used in a number of applications as, for instance, to handle optimization problems by applying a cost function to quantifiable terms (i.e., arithmetic terms which are associated to set terms).

Also, a generic framework for defining and solving interval constraints on any set of domains (finite or infinite) with a lattice structure is formulated in Fernández and

Hill (2004, 2006). This approach also belongs to the cooperation category described in Subsection 6.1. It enables the construction of new (compound) constraint solvers from existing solvers using lattice combinators, so that different solvers (possibly on distinct domains) can communicate and cooperate in solving a problem. The $clp(\mathcal{L})$ language presented in Fernández and Hill (2004) is a prototype implementation of this framework and allows information to be transmitted among different (possibly user-defined) computation domains.

Our proposal in this paper can also be considered to fit into the special constructs category by viewing bridge constraints as channels that enable the propagation of information between different computation domains.

6.3 Interoperability

A number of recent publications deal with approaches to solver cooperation requiring *interoperability*, understood as the behavior of some coordinating system that supports communication between several autonomous systems. In such settings, cooperation relies on suitable interfaces, which have to be specified and implemented according to the specific formats required by the various domains and solvers.

For instance, Goualard (2001) proposes a C++ constraint-solving library called aLiX for communicating between different solvers, possibly written in different languages. Two of the main aims of aLiX are to permit the transparent communication of solvers and ensure *type safety*, that is to say, the capacity to prevent *a priori* the connection of a solver that does not conform to the input format of the interface with another solver. The current version of aLiX is not mature yet, although its interoperability approach offers interesting possibilities. One of the main shortcomings of the current aLiX version is that a component for solving continuous constraints is not yet integrated into the system, and thus real constraints cannot be processed.

In the same spirit, many constraint systems provide both a linear and a nonlinear solver for the real domain. As the linear solver is more efficient of the two, it should be used whenever the constraints are linear, and there is a need for communication between the two real solvers. As an example, Monfroy *et al.* (1995) describe a client/server architecture to enable communication between the component solvers. This consists of managers for the system and the solvers that must be defined on the same computational domain (the real numbers, for example) but with different classes of admissible constraints (i.e., linear and nonlinear constraints). The *CLP* system *CoSAc* is an implementation of this architecture. A built-in platform permits the integration and connection of the components. The exchange of information is managed by means of pipes and the exchanged data are character strings. One of the main drawbacks of this system is the lack of type safety. Moreover, the cooperation happens at a fixed level that prevents the communication of solvers in a transparent way, since the solvers cannot obtain additional information from the structure of the internal constraint store. As already discussed at the end of Subsection 3.3, the current $\mathcal{TOY}$ implementation of our cooperative computation model suffers from a similar limitation, preventing the constraints already placed into the $\mathcal{FD}$ and $\mathcal{R}$

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

stores to be projected. This issue should be addressed in future improvements of our system.

As CoSAc does not permit solver combination, Monfroy designed a domain-independent environment for *solver collaboration*, and he used this concept in order to unify solver cooperation and combination. Basically, solver cooperation means the use of several solvers with data exchange between them, whereas solver combination is understood as the construction of new solvers from other previously defined solvers. In his Ph.D. thesis (Monfroy 1996), Monfroy developed the system BALI (binding architecture for solver integration) that facilitates the integration of heterogeneous solvers as well as the specification of solver cooperation via a number of cooperation primitives. Monfroy's approach assumes that all the solvers work over a common store, while our own proposal requires communication among different stores. Monfroy also designed SoleX (Monfroy and Ringeissen 1999), a domain-independent scheme for constraint-solver extension. This schema consists of a set of rules for transforming constraints that cannot be managed by a solver into constraints that can be treated by that solver, thus extending the range of solvable constraints. Unfortunately, as commented in Monfroy (1996: 195), SoleX and BALI were not integrated. Such an integration could lead to a framework including both solver collaboration and solver extension.

The interoperability category also includes a line of research dealing with the development of coordination languages, aiming at the specification of cooperation between solvers. There exist several proposals whose main goal is to study the use of control languages to specify elementary constraint solvers as well as the collaboration of solvers in a uniform and flexible way. For instance, Arbab and Monfroy (1998) propose to use the coordination language MANIFOLD for improving the constraint-solver collaboration language of BALI. More recent works such as Monfroy and Castro (2004) and Castro and Monfroy (2004) aim at providing means of designing strategies that are more complex than simple master–slave approaches. Basically, Castro and Monfroy propose an asynchronous language composed of interaction components that control external agents (in particular solvers) by managing the data flow. A software framework for constructing distributed constraint solvers, implemented in the coordination language MANIFOLD, has been described in Zoeteweiz (2003). A different point of view regarding solver cooperation is analyzed in Pajot and Monfroy (2003), where a paradigm to enable the user to separate computation strategies from the search phases is presented.

Also it is worth mentioning the project COCONUT[1] whose goal was to integrate techniques from mathematical programming, constraint programming, and interval analysis (and thus it can also be catalogued in the category of cooperation via techniques combination as described in Section 6.4). A modular solver environment, that can be extended with open-source and commercial solvers, was provided for nonlinear continuous global optimization. This framework was also designed for

[1] See `http://www.mat.univie.ac.at/ neum/glopt/coconut/`

distributed computing and has a strategy engine that can be programmed using a specific interpreted language based on Python.

Mircea Marin has developed in his Ph.D. thesis (Marin 2000) a *CFLP* scheme that combines Monfroy's approach to solver cooperation (Monfroy 1996) with a higher-order lazy narrowing calculus somewhat similar to López-Fraguas *et al.* (2004) and the goal-solving calculus presented in Section 3 of this paper. In this setting, Monfroy's ideas are used to provide various primitives for solver combination, and the *CFLP* scheme allows to embed the resulting solvers into a functional and logic programming language. In contrast to our proposal, Marin's approach allows for higher-order unification, which leads both to greater expressivity and to less efficient implementations. Another difference w.r.t. our approach is the intended application domain. The instance of *CFLP* implemented by Marin *et al.* (2001) combines four solvers over a constraint domain for algebraic symbolic computation. This line of research has been continued in works such as Kobayashi (2003) and Kobayashi *et a*l. (2001, 2002, 2003). These papers describe a collaborative *CFLP* system, called Open *CFLP*, which solves symbolic constraints by collaboration between distributed constraint solvers in an open environment such as Internet. The solvers act as providers of constraint-solving services, and Open *CFLP* is able to use them without knowing their location and implementation details. The common communication infrastructure (i.e., the protocol) and the specification language were implemented using CORBA and MathML, respectively.

Another recent proposal for the combination of solvers in a declarative programming language can be found in de la Banda *et al.* (2001). This paper deals with the construction of solvers in the HAL system, which supports the extension of existing solvers and the construction of hybrid ones. HAL provides semioptional type, mode and determinism declarations for predicates and functions as well as a system of type classes over which constraint solvers' capabilities are specified. In particular, HAL type classes can require that the types belonging to them must have a suitable associated constraint solver.

A quite general scheme for solver cooperation fitting the interoperability category has been proposed by Hofstedt (2000a, 2000b, 2001) and Hofstedt and Pepper (2007). Here, constraint domains are formalized by using Σ-structures in a sorted language, constraints are modeled as n-ary relations, and cooperation of solvers is achieved by two mechanisms: constraint propagation that submits a constraint belonging to its corresponding store; and projection of constraint stores that consults the contents of a given store $S_{\mathscr{D}}$ and deduces constraints for another domain. Relying on these mechanisms, different constraint solvers (possibly working over different domains, and implemented in various languages) can be used as components of an overall system, whose architecture provides a uniform interface for constraint solvers which allows a fine-grain formal specification of information exchange between them. This approach has been implemented in the system META-S (Frank *et al.* 2003a, 2003b, 2005) that supports the dynamic integration of arbitrary external (stand-alone) solvers to enable the collaborative processing of constraints. Some analogies and differences between this approach and our own have been discussed already at several places of this paper (see Introduction, and Sections 3.3 and 5).

As a more theoretical line of work related to the interoperability category, there are a number of formal approaches to the combination of constraint solvers on domains modeled as algebraic structures. This kind of research stems from a seminal paper by Nelson and Oppen (1979). More recent relevant work includes several papers by Baader and Schulz (1995). For instance, Baader and Schulz (1995) provide an abstract framework to combine constraint languages and constraint solvers, and focuse on ways in which different and independently defined solvers may be combined. This paper does not really deal with the constraint cooperation mechanism, but it focuses in defining algebraic properties needed for the combination of constraint languages and solvers. Later on, Baader and Schulz (1998) generalized a proposal from a previous paper (Baader and Schulz 1996) and presented a general method for the combination of constraint systems, which is is applicable to so-called *quasi-structures*. This general notion comprises various instances, such as (quotient) term algebras, rational trees, lists, sets, etc. The methods proposed in Baader and Schulz (1996, 1998) can be seen as extensions of previous approaches to the combination of unification algorithms for equational theories, viewing them as instances of constraint solvers (Kirchner and Ringeissen 1992, 1994). As pointed out in Kepser and Richts (1999), a weak point of these approaches is the lack of practical use.

Our proposal can clearly be catalogued in the interoperability category, because it aims at the cooperation of several constraint domains equipped with their respective solvers. Our main communication mechanism, namely bridges, has the advantage of syntactic simplicity, while being compatible with the static type systems used by many declarative languages. Moreover, our notion of coordination domain allows us to use a generic scheme for *CFLP* programming as a formal foundation.

6.4 Combining methods and/or solvers based on different algorithms

One popular approach to cooperation consists of combining solvers or methods based on different algorithms. In this category, we include the integration of different paradigms in one language. In the following, we provide a (nonexhaustive) list of proposals of this kind.

For instance, one of the initial forms of cooperating constraint solving consisted of using different problem solvers (viewed as algorithms) to work individually over different subparts of an overall problem. This was the approach used in Durfee *et al.* (1989) in order to integrate within a network a number of individual solvers intended to work over different parts of a problem. In a similar way, Khedro and Genesereth (1994) proposed a multiagent model where each agent acts independently to solve a distributed set of constraints that constitutes a distributed constraint satisfaction problem. The paper (Hong 1994) also studied the confluence of solvers to solve a common problem, suggesting to manage a set of algorithms each of which should be repeatedly applied on the problem until reaching a stable form.

Within the area of constraint programming, Benhamou (1996) described a unified framework for heterogeneous constraint solving. Here, the cooperation comes from the combination of different algorithms, possibly defined over distinct structures. The main idea is to represent the solvers as constraint narrowing operators (CNO) that are closure operators, and to use a generalized notion of arc-consistency. Conditions on the CNOs needed to ensure the main properties of the principal algorithm are identified. Solver communication involving shared common variables and sending and receiving information to each other is described. The paper also gives a fixed point semantics to describe the cooperation process. One of the main drawbacks of this proposal is that termination of the central algorithm relies on the finiteness of an *approximate domain* A built as a subset of the powerdomain $\wp(D)$ of the domain D under consideration, including D among its members and closed under intersection. For instance, termination cannot be guaranteed in case that D is the domain of sets of real numbers, which is useful for dealing with real interval constraints.

In relation to the problem of solving real constraints, Benhamou *et al.* (1999) have proposed the combination of hull consistency and box consistency with the objective of reducing the computation time achieved by using box consistency alone. This idea was reflected in DecLic (Benhamou *et al.* 1997; Goualard *et al.* 1999), a CLP language that mixes Boolean, integer, and real constraints in the framework of intervals. This system was shown to be fairly efficient on classical benchmarks but at the expense of decreasing the declarativity of the language as a consequence of allowing the programmer to choose the best kind of consistency to use for each constraint.

The combination of interval techniques for solving nonlinear systems is also tackled in Granvilliers (2001), who describes a cooperative strategy to combine the interval-based local consistencies methods (i.e., box and hull consistency) with the multidimensional interval Newton method and shows the efficiency of the main algorithm.

Another proposal for developing a cooperation technique for solving large-scale combinatorial optimization problems was described in Castro *et al.* (2004). This paper introduces a framework for designing cooperative strategies, describing a scheme for the sequential cooperation between Forward Checking and Hill-Climbing. A set of benchmarks for the Capacity Vehicle Routing Problem shows the advantages of using this framework that always outperforms a single solver.

The combination of linear programming solvers and interval solvers has also been specially fertile in the last decades (Marti and Rueher 1995). Many of the cooperating systems resulting from this combination have been implemented as (prototype) declarative systems, as, e.g., ICE (Beringer and Backer 1995), Prolog IV (N'Dong 1997), CIAL (Chiu and Lee 2002), and CCC (Rueher and Solnon 1997), among others.

The integration of mathematical programming techniques in the *CLP* scheme (van Hoeve 2000) may be considered another form of cooperation that has been treated extensively in the literature; see, e.g., the integration of Mixed Integer programming and *CLP* (Rodošěk *et al.* 1997; Harjunkoski *et al.* 2000; Thorsteinsson 2001), the combination of *CLP* and Integer Programming (Bockmayr and Kasper 2000), and the combination of *CLP* and Linear Programming (Vandecasteele and Rodošěk 1998), among others.

The domain cooperation framework presented in this paper is quite generic, and its current implementation in $\mathcal{TOY}$ relies on the availability of black box solvers

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

provided by SICStus Prolog (2007). Therefore, it cannot be catalogued into the cooperation category considered in this subsection, which is very specific and relies on a detailed control of the techniques and solvers involved. Nevertheless, the work described in this subsection points to combination techniques which lead to improved performance and may be useful for future implementations of our approach.

7 Conclusions and future work

The work presented in this paper is aimed as a contribution to the efficient use of constraint domains and solvers in declarative languages and systems. We have investigated foundational and practical issues concerning a computational framework for the cooperation of constraint domains in *CFLP*, using constraint projection guided by bridge constraints as the main cooperation tool. Taking a generic scheme as a formal basis, we have focused on a particular case of practical importance, namely the cooperation among the symbolic Herbrand domain $\mathcal{H}$ and the two numeric domains $\mathcal{R}$ and $\mathcal{FD}$.

The relation to our previous related work and some pointers to related work by other researchers have been presented in Section 1, and a more detailed discussion of the state-of-the-art concerning cooperation of constraint domains can be found in Section 6. In the rest of this section, we give a summary of the main results presented in the other sections of the paper, followed by some considerations concerning current limitations and possible lines of future work.

7.1 *Summary of main results*

Our results include a formal computation model for cooperative goal solving in *CFLP*, the development of an implemented system, and experimental evidence on the implementation's performance and its comparison with the closest related system we are aware of. More precisely:

- In Section 2, we have presented a formal framework for the cooperation of constraint domains in an improved version of an already existing *CFLP* scheme for *CFLP*. We have formalized a notion of constraint solver suitable for *CFLP* programming, as well as a mathematical construction of coordination domains, a special kind of hybrid domains built as a combination of several pure domains intended to cooperate. In addition to the facilities provided by their components, coordination domains supply special primitives for building bridge constraints to allow communication between different component domains. As particular case of practical interest, we have formalized a coordination domain $\mathcal{C} = \mathcal{M} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{R}$ tailored to the cooperation of three useful pure domains: the Herbrand domain $\mathcal{H}$ which supplies equality and disequality constraints over symbolic terms, the domain $\mathcal{R}$ which supplies arithmetic constraints over real numbers, and the domain $\mathcal{FD}$ which supplies finite domain constraints over integer numbers. Practical applications involving more that one of these pure domains can be naturally treated within the

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

$CFLP(\mathcal{C})$ instance of the $CFLP$ scheme. From a programmer's viewpoint, the domain $\mathcal{H}$ supports generic equality and disequality constraints over arbitrary user-defined datatypes, while $\mathcal{R}$ and $\mathcal{FD}$ provide more specific numeric constraints.

- Section 3 presents a formal calculus for cooperative goal solving in $CFLP(\mathcal{C})$. The main programming features available to $CFLP(\mathcal{C})$ programmers include a Milner's like polymorphic type system, lazy and possibly higher-order functions, predicates, and the cooperation of the three domains within $\mathcal{C}$. The goal-solving calculus is presented as a set of goal transformation rules for reducing initial goals into solved forms. There are rules that use lazy narrowing to process program defined function calls in a demand-driven way, domain cooperation rules dealing among other things with bridges and projections, and constraint-solving rules to invoke the solvers of the various pure domains involved in the cooperation. The section concludes with theoretical results ensuring soundness and completeness of the goal-solving calculus, where completeness is guaranteed for well-typed solutions as far as permitted by the completeness of the underlying solvers and some other more technical requirements.
- Section 4 presents the implementation of the cooperative goal-solving calculus for $CFLP(\mathcal{C})$ in a state-of-the-art declarative programming system. In addition to describing general aspects such as the software architecture, we have focused on the implementation of domain cooperation mechanisms, illustrating the correspondence between code generation in the implemented system, and the goal transformation rules for cooperation formalized in the previous section.
- Section 5 is devoted to performance analysis by means of a set of benchmarks. The experimental results obtained lead us to several conclusions. First, we conclude that the activation of the domain cooperation mechanisms between $\mathcal{FD}$ and $\mathcal{R}$ does not penalize the execution time in problems which can be solved by using the domain $\mathcal{FD}$ alone. Second, we also conclude that the cooperation mechanism using projections helps to speed up the execution time in problems where a real cooperation between $\mathcal{FD}$ and $\mathcal{R}$ is needed. Third, our experiments show a good performance of our implementation with respect to the closest related system we are aware of. In summary, we conclude that our approach to the cooperation of constraint domains has been effectively implemented in a practical system that is distributed as a free open-source Sourceforge project (`http://toy.sourceforge.net`) and runs on several platforms.

7.2 Some current limitations and planned future work

In the future, we would like to improve some of the limitations of our current approach to domain cooperation, concerning both the formal foundations and the implemented system. More precisely:

- The cooperative goal-solving calculus $CCLNC(\mathcal{C})$ presented in Section 3 should be generalized to allow for an arbitrary coordination domain $\mathcal{C}$ in

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

place of the concrete choice $\mathscr{M} \oplus \mathscr{H} \oplus \mathscr{FD} \oplus \mathscr{R}$. This is a straightforward task. However, for the purposes of the present paper we found more appropriate to deal just with the coordination domain supported by the current implementation.

- The implemented system should be expanded to support some of these more general coordination domains, which could include specific domains for Boolean values, sets, or different types of numeric values. More efficient and powerful constraint solvers for such domains should also be integrated within the implementation.
- $CCLNC(\mathscr{C})$ should also be expanded to allow the computation of projections from the primitive constraints placed within the constraint stores. These more powerful projections were allowed in the preliminary version of $CCLNC(\mathscr{C})$ presented in Estévez-Martín *et al.* (2007b), but they were not implemented and no completeness result was given. Currently, projections are computed only from the constraints placed in the constraint pool (see rule **PP** in Table 4 in Subsection 3.3) and the $\mathscr{TOY}$ implementation only supports this kind of projections. Allowing projections to act over stored constraints will require to solve new problems both on the formal level (where some substantial difficulties are expected for proving a completeness result) and on the implementation level (where the current system will have to be modified to enable a transparent access to the constraint stores).
- As a consequence of the previous improvement, the cooperative goal-solving process will show more complicated patterns of interaction among solvers. Therefore, some means to describe goal-solving strategies should be provided to enable users to specify some desired sequences of goal transformation rules, especially with regard to the activation of solvers and projections. In addition to being implemented as part of the practical system, goal-solving strategies are expected to be helpful for proving the completeness of a cooperative goal-solving calculus improved as described in the previous item.
- The experimentation with benchmarks and application cases should be further developed.
- Last but not least, the implemented system should be properly maintained and improved in various ways. In particular, library management should be standardized, both with respect to loading already existing libraries and with respect to developing new ones.

Appendix A [Auxiliary results and proofs]

This Appendix collects proofs of the results stated in Sections 2 and 3 omitted from the main text. Some of them rely on previously stated auxiliary results, especially Lemmata 1 and 2 from Subsection 2.2 and Lemma 3 from Subsection 2.3. In addition, some other auxiliary results will be included at the proper places.

A.1 Properties of constraint solvers and coordination domains

The first part of the Appendix includes the proofs of the main results stated in Section 2. First, we present the proof of Lemma 5, about general properties of proof transformation systems.

Proof of Lemma 5

(1) The transition relation $\Vdash_{\mathcal{D},\mathcal{X}}$ of the *sts* generates a tree with root $\Pi \Box \varepsilon$, whose leaves correspond to the stores belonging to $\mathcal{SF}_{\mathcal{D}}(\Pi, \mathcal{X})$. Since $\Vdash_{\mathcal{D},\mathcal{X}}$ is finitely branching and terminating, this tree is locally finite and has no infinite branches. By so-called *König's Lemma* (see Baader and Nipkow 1998; Section 2.2) the tree must be finite. Therefore, it must have finitely many leaves, and $\mathcal{SF}_{\mathcal{D}}(\Pi, \mathcal{X})$ is finite. For later use, we remark that $solve^{\mathcal{D}}(\Pi, \mathcal{X})$ can be characterized as

$$\bigvee\{\exists \overline{Y'}(\Pi' \Box \sigma') \mid \Pi \Box \varepsilon \Vdash_{\mathcal{D},\mathcal{X}} ! \ \Pi' \Box \sigma', \ \overline{Y'} = var(\Pi' \Box \sigma') \setminus var(\Pi)\}$$

(2) Assume that the *sts* has the fresh local variables property and the safe bindings property. Because of the remark at the end of item (1), for each $\mathcal{X}$-solved form $\exists \overline{Y'}(\Pi' \Box \sigma')$ computed by the call $solve^{\mathcal{D}}(\Pi, \mathcal{X})$ there is some sequence of $\Vdash_{\mathcal{D},\mathcal{X}}$ steps

$$\Pi \Box \varepsilon = \Pi'_0 \Box \mu'_0 \Vdash_{\mathcal{D},\mathcal{X}} \Pi'_1 \Box \mu'_1 \Vdash_{\mathcal{D},\mathcal{X}} \dots \Vdash_{\mathcal{D},\mathcal{X}} \Pi'_n \Box \mu'_n$$

such that $\Pi'_n \Box \mu'_n = \Pi' \Box \sigma'$ is irreducible, and the following conditions hold for all $1 \leqslant i \leqslant n$: $\Pi'_i \Box \mu'_i$ is a store with fresh local variables $\overline{Y'_i} = var(\Pi'_i \Box \mu'_i) \setminus var(\Pi'_{i-1} \Box \mu'_{i-1})$; $\mu'_i = \mu'_{i-1}\mu_i$ for some substitution μ_i verifying $vdom(\mu_i) \cup vran(\mu_i) \subseteq var(\Pi'_{i-1}) \cup \overline{Y'_i}$; and $\mu_i(X)$ is a constant for all $X \in \mathcal{X} \cap vdom(\mu_i)$. Then, $\overline{Y'} = \overline{Y'_1}, \dots, \overline{Y'_n}$, and an easy induction on n allows to prove that $vdom(\sigma') \cup vran(\sigma') \subseteq var(\Pi) \cup \overline{Y'}$ and that $\sigma'(X)$ is a constant for all $X \in \mathcal{X} \cap vdom(\sigma')$. Therefore, the solver $solve^{\mathcal{D}}$ also satisfies the fresh local variables property and the safe bindings property.

(3) Assume that the *sts* is locally sound. Because of the remark in item (1), to prove soundness of $solve^{\mathcal{D}}$ it is sufficient to show that the union

$$\bigcup\{Sol_{\mathcal{D}}(\exists \overline{Y'}(\Pi' \Box \sigma')) \mid \Pi \Box \sigma \Vdash_{\mathcal{D},\mathcal{X}} ! \ \Pi' \Box \sigma', \overline{Y'} = var(\Pi' \Box \sigma') \setminus var(\Pi \Box \sigma)\}$$

is a subset of $Sol_{\mathcal{D}}(\Pi \Box \sigma)$. In order to show this, we assume

$$\Pi \Box \sigma \Vdash^{n}_{\mathcal{D},\mathcal{X}} ! \ \Pi' \Box \sigma', \ \overline{Y'} = var(\Pi' \Box \sigma') \setminus var(\Pi \Box \sigma)$$

and prove $Sol_{\mathcal{D}}(\exists \overline{Y'}(\Pi' \Box \sigma')) \subseteq Sol_{\mathcal{D}}(\Pi \Box \sigma)$ by induction on n:
$\underline{n = 0}$: in this case $\overline{Y'} = \emptyset$, $\Pi' \Box \sigma' = \Pi \Box \sigma$. The inclusion to be proved is trivial.
$\underline{n > 0}$: in this case $\Pi \Box \sigma \Vdash_{\mathcal{D},\mathcal{X}} \Pi'_1 \Box \sigma'_1 \Vdash^{n-1}_{\mathcal{D},\mathcal{X}} ! \ \Pi' \Box \sigma'$ for some store $\Pi'_1 \Box \sigma'_1$. Let $\overline{Y'_1} = var(\Pi'_1 \Box \sigma'_1) \setminus var(\Pi \Box \sigma)$ and $\overline{Y''} = var(\Pi' \Box \sigma') \setminus var(\Pi'_1 \Box \sigma'_1)$. Then, $\overline{Y'} = \overline{Y'_1}, \overline{Y''} = var(\Pi' \Box \sigma') \setminus var(\Pi \Box \sigma)$. By induction hypothesis, we can assume $Sol_{\mathcal{D}}(\exists \overline{Y''}(\Pi' \Box \sigma')) \subseteq Sol_{\mathcal{D}}(\Pi'_1 \Box \sigma'_1)$. Then, for any given $\eta \in Sol_{\mathcal{D}}(\exists \overline{Y'}(\Pi' \Box \sigma'))$ we can prove $\eta \in Sol_{\mathcal{D}}(\Pi \Box \sigma)$ by the following reasoning: by definition of $Sol_{\mathcal{D}}$, there is $\eta' \in Sol_{\mathcal{D}}(\Pi' \Box \sigma')$ such that $\eta' =_{\setminus \overline{Y'}} \eta$

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

and hence $\eta' =_{var(\Pi \,\Box\, \sigma)} \eta$. Trivially, it follows that $\eta' \in Sol_{\mathscr{D}}(\exists \overline{Y''}(\Pi' \Box \sigma'))$, which implies $\eta' \in Sol_{\mathscr{D}}(\Pi'_1 \Box \sigma'_1)$ by induction hypothesis. Trivially again, it follows that $\eta' \in Sol_{\mathscr{D}}(\exists \overline{Y'_1}(\Pi'_1 \Box \sigma'_1))$ which implies $\eta' \in Sol_{\mathscr{D}}(\Pi \Box \sigma)$ due to local soundness. Since $\eta' =_{var(\Pi \,\Box\, \sigma)} \eta$, we can conclude that $\eta \in Sol_{\mathscr{D}}(\Pi \Box \sigma)$.

(4) Assume now a selected set $\mathscr{RS}$ of *strs* such that the *sts* is locally complete for $\mathscr{RS}$-free steps. Because of the remark in item 1., to prove completeness of $solve^{\mathscr{D}}$ for $\mathscr{RS}$-free invocations it is sufficient to show that $WTSol_{\mathscr{D}}(\Pi \Box \sigma)$ is a subset of the union

$$\bigcup\{WTSol_{\mathscr{D}}(\exists \overline{Y'}(\Pi' \Box \sigma')) \mid \Pi \Box \sigma \Vdash_{\mathscr{D},\mathscr{X}} ! \, \Pi' \Box \sigma', \, \overline{Y'} = var(\Pi' \Box \sigma') \backslash var(\Pi \Box \sigma)\}$$

under the additional assumption that $\Pi \Box \sigma$ is hereditarily $\mathscr{RS}$-irreducible. This can be viewed as a property of the store $\Pi \Box \sigma$ that can be proved by *well-founded induction* (see again Baader and Nipkow 1998; Section 2.2) on the terminating store transformation relation $\Vdash_{\mathscr{D},\mathscr{X}}$:

Base Case: $\Pi \Box \sigma$ is irreducible w.r.t. $\Vdash_{\mathscr{D},\mathscr{X}}$. In this case, the union reduces to the set $WTSol_{\mathscr{D}}(\Pi \Box \sigma)$ and the inclusion to be proved is trivial.

Inductive Case: $\Pi \Box \sigma$ is reducible w.r.t. $\Vdash_{\mathscr{D},\mathscr{X}}$. In this case, since $\Pi \Box \sigma$ is hereditarily $\mathscr{RS}$-irreducible and the *sts* is locally complete for $\mathscr{RS}$-free steps, for any $\eta \in WTSol_{\mathscr{D}}(\Pi \Box \sigma)$ there is some hereditarily $\mathscr{RS}$-irreducible $(\Pi'_1 \Box \sigma'_1)$ such that $\Pi \Box \sigma \Vdash_{\mathscr{D},\mathscr{X}} \Pi'_1 \Box \sigma'_1$ and $\eta \in WTSol_{\mathscr{D}}(\exists \overline{Y'_1}(\Pi'_1 \Box \sigma'_1))$ where $\overline{Y'_1} = var(\Pi'_1 \Box \sigma'_1) \setminus var(\Pi \Box \sigma)$. Then, by definition of $Sol_{\mathscr{D}}$, there is $\eta'_1 \in WTSol_{\mathscr{D}}(\Pi'_1 \Box \sigma'_1)$ such that $\eta'_1 =_{\backslash \overline{Y'_1}} \eta$. The induction hypothesis can be assumed for $\Pi'_1 \Box \sigma'_1$, and there must be some $\Pi' \Box \sigma'$ such that $\Pi'_1 \Box \sigma'_1 \Vdash_{\mathscr{D},\mathscr{X}} ! \, \Pi' \Box \sigma'$, $\overline{Y''} = var(\Pi' \Box \sigma') \setminus var(\Pi'_1 \Box \sigma'_1)$ and $\eta'_1 \in WTSol_{\mathscr{D}}(\exists \overline{Y''}(\Pi' \Box \sigma'))$. By definition of $Sol_{\mathscr{D}}$, there is $\eta' \in WTSol_{\mathscr{D}}(\Pi' \Box \sigma')$ such that $\eta' =_{\backslash \overline{Y''}} \eta'_1$. Moreover, we get $\Pi \Box \sigma \Vdash_{\mathscr{D},\mathscr{X}} ! \, \Pi' \Box \sigma'$ and $\overline{Y'} = \overline{Y'_1}, \overline{Y''} = var(\Pi' \Box \sigma') \setminus var(\Pi \Box \sigma)$ such that $\eta' =_{\backslash \overline{Y'}} \eta$, and thus $\eta \in WTSol_{\mathscr{D}}(\exists \overline{Y'}(\Pi' \Box \sigma'))$.

This completes the proof of the lemma. □

Table A 1 displayed in the next page and the two auxiliary lemmata stated and proved immediately afterward will be used in the subsequent proof of Theorem 1, the main result in this subsection. It ensures that $solve^{\mathscr{H}}$ satisfies the requirements for solvers listed in Definition 6 (except for a technical limitation concerning completeness). The proof of this theorem also relies on Lemma 5.

Lemma 7 (*Auxiliary Soundness Lemma*)
Assume $\Pi \subseteq PCon_{\mathscr{D}}$ and $\sigma, \sigma_1 \in Sub_{\mathscr{D}}$ such that σ is idempotent and $\Pi\sigma = \Pi$. Then $Sol_{\mathscr{D}}(\Pi\sigma_1) \cap Sol_{\mathscr{D}}(\sigma\sigma_1) \subseteq Sol_{\mathscr{D}}(\Pi) \cap Sol_{\mathscr{D}}(\sigma)$.

Proof of Lemma 7
The hypothesis of the lemma say that $\sigma = \sigma\sigma$ and $\Pi\sigma = \Pi$. On the other hand, because of the Substitution Lemma 3 and the definition of $Sol_{\mathscr{D}}$, any $\eta \in Val_{\mathscr{D}}$ verifies $\eta \in Sol_{\mathscr{D}}(\Pi\sigma_1) \cap Sol_{\mathscr{D}}(\sigma\sigma_1)$ iff $\sigma_1\eta \in Sol_{\mathscr{D}}(\Pi)$ and $\sigma\sigma_1\eta = \eta$. Therefore, to prove the lemma it suffices to assume

$$(a)\ \sigma = \sigma\sigma \qquad (b)\ \Pi\sigma = \Pi \qquad (c)\ \sigma_1\eta \in Sol_{\mathscr{D}}(\Pi) \qquad (d)\ \sigma\sigma_1\eta = \eta$$

and to deduce from these assumptions that $\eta \in Sol_{\mathscr{D}}(\Pi) \cap Sol_{\mathscr{D}}(\sigma)$.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Table A 1. *Well-founded progress ordering for* $>_{lex}$

Rules	P_1	P_2	P_3	P_4	P_5
$\mathbf{H}_1$	$\geqslant$	$\geqslant$	$\geqslant$	$>$	
$\mathbf{H}_2$	$\geqslant$	$\geqslant$	$\geqslant$	$>$	
$\mathbf{H}_3$	$\geqslant$	$\geqslant$	$>$		
$\mathbf{H}_4$	$\geqslant$	$\geqslant$	$\geqslant$	$\geqslant$	$>$
$\mathbf{H}_5$	$>$				
$\mathbf{H}_6$	$>$				
$\mathbf{H}_7$	$\geqslant$	$\geqslant$	$>$		
$\mathbf{H}_8$	$>$				
$\mathbf{H}_9$	$>$				
$\mathbf{H}_{10}$	$\geqslant$	$\geqslant$	$\geqslant$	$\geqslant$	$>$
$\mathbf{H}_{11a}$	$\geqslant$	$>$			
$\mathbf{H}_{11b}$	$>$				
$\mathbf{H}_{12}$	$>$				
$\mathbf{H}_{13}$	$>$				

First, we prove that $\eta \in Sol_{\mathcal{D}}(\Pi)$ as follows: by (c) and (b), we obtain $\sigma_1\eta \in Sol_{\mathcal{D}}(\Pi\sigma)$, which amounts to $\sigma\sigma_1\eta \in Sol_{\mathcal{D}}(\Pi)$ by the Substitution Lemma. By (d), this is the same as $\eta \in Sol_{\mathcal{D}}(\Pi)$.

Next, we note that $\eta \in Sol_{\mathcal{D}}(\sigma)$ is equivalent to $\sigma\eta = \eta$, which can be proved by the following chain of equalities: $\sigma\eta =_{(d)} \sigma\sigma\sigma_1\eta =_{(a)} \sigma\sigma_1\eta =_{(d)} \eta$. □

Lemma 8 (*Auxiliary Completeness Lemma*)
Assume $\Pi \subseteq PCon_{\mathcal{D}}$, $\sigma, \sigma_1 \in Sub_{\mathcal{D}}$ and $\eta, \eta' \in Val_{\mathcal{D}}$ such that $\eta \in Sol_{\mathcal{D}}(\Pi) \cap Sol_{\mathcal{D}}(\sigma)$, $\sigma_1\eta' = \eta'$ and $\eta' =_{\backslash \overline{Y'}} \eta$, where $\overline{Y'}$ are fresh variables away from $var(\Pi) \cup vdom(\sigma) \cup vran(\sigma)$. Then $\sigma\eta' = \eta'$ and $\eta' \in Sol_{\mathcal{D}}(\Pi\sigma_1) \cap Sol_{\mathcal{D}}(\sigma\sigma_1)$.

Proof of Lemma 8
In what follows we can assume $\sigma\eta = \eta$ due to the hypothesis $\eta \in Sol_{\mathcal{D}}(\sigma)$.

We prove $\sigma\eta' = \eta'$ by showing that $X\sigma\eta' = X\eta'$ holds for any variable $X \in \mathcal{V}ar$. This is trivial for $X \notin vdom(\sigma)$. For $X \in vdom(\sigma)$, we can assume that $\overline{Y'}$ is away from X and $var(X\sigma)$; therefore $\eta' =_{X,var(X\sigma)} \eta$ and hence $X\sigma\eta' = X\sigma\eta = X\eta = X\eta'$ (where the assumption $\sigma\eta = \eta$ has been used at the second step).

Now we prove $\eta' \in Sol_{\mathcal{D}}(\Pi\sigma_1)$. Because of the Substitution Lemma 3, this is equivalent to $\sigma_1\eta' \in Sol_{\mathcal{D}}(\Pi)$, which amounts to $\eta \in Sol_{\mathcal{D}}(\Pi)$ due to the hypothesis

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

$\sigma_1\eta' = \eta'$, $\eta' =_{\setminus \overline{Y'}} \eta$ and $\overline{Y'}$ away from $var(\Pi)$. But $\eta \in Sol_{\mathcal{D}}(\Pi)$ is also ensured by the hypothesis.

Finally, $\eta' \in Sol_{\mathcal{D}}(\sigma\sigma_1)$ is equivalent to $\sigma\sigma_1\eta' = \eta'$, which can be proved as follows: $\sigma\sigma_1\eta' = \sigma\eta' = \eta'$ (where the first step relies on the assumption $\sigma_1\eta' = \eta'$ and the second step relies on a previously proved equality). □

Proof of Theorem 1

Consider the *sts* for $\mathcal{H}$ stores with transition relation $\Vdash_{\mathcal{H},\mathcal{X}}$ as specified in Table 1 in Subsection 2.4.2, implicitly assuming that the notation used for the various *str*s is exactly the same as there. We prove that this *sts* satisfies the six properties enumerated in Definition 7. The last one (namely **Local Completeness**) holds for $\mathcal{URS}$-free steps, where $\mathcal{URS} = \{\mathbf{OH3}, \mathbf{OH7}, \mathbf{H13}\}$ is the set of unsafe $\mathcal{H}$-*str*s, as explained in Subsection 2.4.2.

(1) **Fresh Local Variables Property:** The specification of the *str*s in Table 1 clearly guarantees this property.
(2) **Safe Bindings Property:** An inspection of Table 1 shows that the *str*s **H1** and **H2** bind a variable to a constant, and the other *str*s never bind a variable $X \in \mathcal{X}$. Therefore, this property is also satisfied.
(3) **Finitely Branching Property:** This property holds because those *str*s that allow a nondeterministic choice of the next store provide only finitely many possibilities.
(4) **Termination Property:** Given a $\mathcal{H}$ store $\Pi \Box \sigma$ and a set $\mathcal{X} \subseteq cvar(\Pi)$, we define a 5-tuple of natural numbers $||\Pi \Box \sigma||_{\mathcal{X}} =_{def} (P_1, P_2, P_3, P_4, P_5) \in \mathbb{N}^5$ where

   $\mathbf{P}_1$ is the number of occurrences of atomic constraints in Π which are *unsolved* w.r.t. $\mathcal{X}$. In this context, an atomic constraint π occurring in Π is said to be unsolved w.r.t. $\mathcal{X}$ iff some of the *str*s can be applied taking π as the selected atomic constraint.

   $\mathbf{P}_2$ is the sum of the depths of all the occurrences of variables $X \in \mathcal{X}$ within patterns in Π.

   $\mathbf{P}_3$ is the sum of the syntactical sizes of all the patterns occurring in Π.

   $\mathbf{P}_4$ is the number of *unsolved* occurrences of obviously demanded variables in Π. In this context, an occurrence of an obviously demanded variable X in Π is called solved iff X occurs in a constraint of the form X == X, and unsolved otherwise.

   $\mathbf{P}_5$ is the number of occurrences of *misplaced* variables in Π. In this context, misplaced occurrences of X in Π are those occurrences of the form t == X or t /= X, with $t \in \mathcal{V}ar$ and $X \neq t$.

   Let $>_{lex}$ be the lexicographic ordering induced by $>_{\mathbb{N}}$ over $\mathbb{N}^5$. We claim that:

$$(\star)\, \Pi \Box \sigma \Vdash_{\mathcal{H},\mathcal{X}} \Pi' \Box \sigma' \;\Rightarrow\; ||\Pi \Box \sigma||_{\mathcal{X}} >_{lex} ||\Pi' \Box \sigma'||_{\mathcal{X}}$$

   This is justified by Table A 1, which shows the behavior of the different *str*s w.r.t. $>_{lex}$. In order to understand the table, note that two different cases have been distinguished for the application of the *str* $\mathbf{H}_{11}$, namely:

- $\mathbf{H}_{11a}$ Application of $\mathbf{H}_{11}$ choosing a value of i such that $\mathscr{X} \cap var(t_i) \neq \emptyset$.
- $\mathbf{H}_{11b}$ Application of $\mathbf{H}_{11}$ choosing a value of i such that $\mathscr{X} \cap var(t_i) = \emptyset$.

Since $>_{lex}$ is a well-founded ordering, termination of $\Vdash_{\mathscr{H},\mathscr{X}}$ can be concluded from ($\star$). The reader is referred to Section 2.3 in Baader and Nipkow (1998) for more information on this proof technique.

(5) **Local Soundness Property:** Given a $\mathscr{H}$ store $\Pi \,\Box\, \sigma$ and a set $\mathscr{X} \subseteq odvar_{\mathscr{H}}(\Pi)$, we must prove that the union

$$\bigcup\{Sol_{\mathscr{H}}(\exists \overline{Y'}(\Pi' \,\Box\, \sigma')) \mid \Pi \,\Box\, \sigma \Vdash_{\mathscr{H},\mathscr{X}} \Pi' \,\Box\, \sigma',\, \overline{Y'} = var(\Pi' \,\Box\, \sigma') \setminus var(\Pi \,\Box\, \sigma)\}$$

is a subset of $Sol_{\mathscr{D}}(\Pi \,\Box\, \sigma)$. Obviously, it suffices to prove the inclusion

$$(\dagger)\, Sol_{\mathscr{H}}(\exists \overline{Y'}(\Pi' \,\Box\, \sigma')) \subseteq Sol_{\mathscr{H}}(\Pi \,\Box\, \sigma)$$

for each $\Pi' \,\Box\, \sigma'$ such that $\Pi \,\Box\, \sigma \Vdash_{\mathscr{H},\mathscr{X}}! \,\Pi' \,\Box\, \sigma'$ with $\overline{Y'} = var(\Pi' \,\Box\, \sigma') \setminus var(\Pi \,\Box\, \sigma)$. However, ($\dagger$) is an easy consequence of

$$(\dagger\dagger)\, Sol_{\mathscr{H}}(\Pi' \,\Box\, \sigma') \subseteq Sol_{\mathscr{H}}(\Pi \,\Box\, \sigma)$$

In fact, assuming ($\dagger\dagger$) and an arbitrary $\eta \in Sol_{\mathscr{H}}(\exists \overline{Y'}(\Pi' \,\Box\, \sigma'))$, there must be some $\eta' \in Sol_{\mathscr{H}}(\Pi' \,\Box\, \sigma')$ such that $\eta =_{\setminus \overline{Y'}} \eta'$. Then, $\eta' \in Sol_{\mathscr{H}}(\Pi \,\Box\, \sigma)$ because of ($\dagger\dagger$), and thus $\eta \in Sol_{\mathscr{H}}(\Pi \,\Box\, \sigma)$ because $\eta =_{\setminus \overline{Y'}} \eta'$ and $\overline{Y'} \cap var(\Pi \,\Box\, \sigma) = \emptyset$. Having proved that ($\dagger\dagger$) entails ($\dagger$), we proceed to prove ($\dagger\dagger$) by a case distinction according to the *str* used in the step $\Pi \,\Box\, \sigma \Vdash_{\mathscr{H},\mathscr{X}} \Pi' \,\Box\, \sigma'$. In each case, we assume that the stores $\Pi \,\Box\, \sigma$ and $\Pi' \,\Box\, \sigma'$ occurring in ($\dagger\dagger$) have exactly the form displayed for the corresponding transformation in Table 1 displayed in Subsection 2.4.2. For instance, in the case of transformation **H1** we write $(t == s) \to! \, R,\ \Pi \,\Box\, \sigma$ in place of $\Pi \,\Box\, \sigma$. Moreover, in all the cases we silently use the fact that the constraints and variables within any store are not affected by the substitution kept in that store.

H1 Assume $\eta \in Sol_{\mathscr{H}}((t == s, \Pi)\sigma_1 \,\Box\, \sigma\sigma_1)$. Then $\eta \in Sol_{\mathscr{H}}((t == s, \Pi)\sigma_1) \cap Sol_{\mathscr{H}}(\sigma\sigma_1)$. We must prove $\eta \in Sol_{\mathscr{H}}((t == s) \to! \, R, \Pi \,\Box\, \sigma)$.
Since $(t == s, \Pi) = (t == s, \Pi)\sigma$, we can infer $\eta \in Sol_{\mathscr{H}}(t == s, \Pi) \cap Sol_{\mathscr{H}}(\sigma)$ from our assumptions and Lemma 7.
It remains to prove that $\eta \in Sol_{\mathscr{H}}((t == s) \to! \, R)$. Since we already know that $\eta \in Sol_{\mathscr{H}}(t == s)$, it suffices to prove that $R\eta = \texttt{true}$. But $\eta \in Sol_{\mathscr{H}}(\sigma\sigma_1)$ means $\sigma\sigma_1\eta = \eta$, and therefore $R\eta = R\sigma\sigma_1\eta = R\sigma_1\eta = \texttt{true}\ \eta = \texttt{true}$.

H2 Very similar to **H1**.

H3 Trivial. Clearly, $Sol_{\mathscr{H}}(\overline{t_m == s_m}) = Sol_{\mathscr{H}}(h\,\overline{t_m} == h\,\overline{s_m})$.

H4 Trivial. Clearly, $Sol_{\mathscr{H}}(X == t) = Sol_{\mathscr{H}}(t == X)$.

H5 Assume $\eta \in Sol_{\mathscr{H}}(tot(t), \Pi\sigma_1 \Box \sigma\sigma_1)$. Then, $t\eta$ is a total pattern and $\eta \in Sol_{\mathscr{H}}(\Pi\sigma_1) \cap Sol_{\mathscr{H}}(\sigma\sigma_1)$. We must prove $\eta \in Sol_{\mathscr{H}}(X == t, \Pi \,\Box \sigma)$.
Since $\Pi = \Pi\sigma$, we can infer $\eta \in Sol_{\mathscr{H}}(\Pi) \cap Sol_{\mathscr{H}}(\sigma)$ from our assumptions and Lemma 7. It remains to prove that $\eta \in Sol_{\mathscr{H}}(X == t)$. But $\eta \in Sol_{\mathscr{H}}(\sigma\sigma_1)$ means $\sigma\sigma_1\eta = \eta$. Thus, $X\eta = X\sigma\sigma_1\eta = X\sigma_1\eta = t\eta$, which implies $\eta \in Sol_{\mathscr{H}}(X == t)$, because $t\eta$ is total.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

H6 Trivial, because $\eta \in Sol_{\mathscr{H}}(\blacksquare)$ is false for any η.
H7 Trivial. Clearly, $Sol_{\mathscr{H}}(\overline{t_i\ /=s_i}) \subseteq Sol_{\mathscr{H}}(h\,\overline{t_m}\ /=\ h\,\overline{s_m})$.
H8 Trivial, because $\eta \in Sol_{\mathscr{H}}(h\,\overline{t_n}\ /=\ h'\,\overline{s_m})$ holds for any η.
H9 Trivial, for the same reason as **H6**.
H10 Trivial. Clearly, $Sol_{\mathscr{H}}(X\ /=\ t) = Sol_{\mathscr{H}}(t\ /=\ X)$.
H11 Assume $\eta \in Sol_{\mathscr{H}}((Z_i\ /=\ t_i, \Pi)\sigma_1 \,\Box\, \sigma\sigma_1)$. Then $\eta \in Sol_{\mathscr{H}}((Z_i\ /=\ t_i)\sigma_1)$ and $\eta \in Sol_{\mathscr{H}}(\Pi\sigma_1) \cap Sol_{\mathscr{H}}(\sigma\sigma_1)$. We must prove $\eta \in Sol_{\mathscr{H}}(X\ /=\ c\,\bar{t}_n, \Pi \,\Box\, \sigma)$. Since $\Pi = \Pi\sigma$, we can infer $\eta \in Sol_{\mathscr{H}}(\Pi) \cap Sol_{\mathscr{H}}(\sigma)$ from our assumptions and Lemma 7. It remains to prove that $\eta \in Sol_{\mathscr{H}}(X\ /=\ c\,\bar{t}_n)$. Because of $\eta \in Sol_{\mathscr{H}}(\sigma\sigma_1)$, we know that $\sigma\sigma_1\eta = \eta$. Therefore, it suffices to prove $\sigma\sigma_1\eta \in Sol_{\mathscr{H}}(X\ /=\ c\,\bar{t}_n)$, which can be reasoned as follows:

$$\begin{aligned}&\sigma\sigma_1\eta \in Sol_{\mathscr{H}}(X\ /=\ c\,\bar{t}_n) \Leftrightarrow_{(1)} \eta \in Sol_{\mathscr{H}}(X\ /=\ c\,\bar{t}_n)\sigma\sigma_1\\ &\Leftrightarrow \eta \in Sol_{\mathscr{H}}(X\ /=\ c\,\bar{t}_n)\sigma_1 \Leftarrow_{(2)} \eta \in Sol_{\mathscr{H}}(Z_i\ /=\ t_i\sigma_1)\\ &\Leftarrow_{(3)} \eta \in Sol_{\mathscr{H}}(Z_i\ /=\ t_i)\sigma_1\end{aligned}$$

where (1) holds because of the Substitution Lemma 3, (2) and (3) hold by construction of σ_1, and $\eta \in Sol_{\mathscr{H}}(Z_i\ /=\ t_i)\sigma_1$ holds because of the assumptions of this case.
H12 Assume $\eta \in Sol_{\mathscr{H}}(\Pi\sigma_1\Box\sigma\sigma_1)$. Then, $\eta \in Sol_{\mathscr{H}}(\Pi\sigma_1) \cap Sol_{\mathscr{H}}(\sigma\sigma_1)$. We must prove $\eta \in Sol_{\mathscr{H}}(X\ /=\ c\,\bar{t}_n, \Pi \,\Box\, \sigma)$.
Since $\Pi = \Pi\sigma$, we can infer $\eta \in Sol_{\mathscr{H}}(\Pi) \cap Sol_{\mathscr{H}}(\sigma)$ from our assumptions and Lemma 7. It remains to prove that $\eta \in Sol_{\mathscr{H}}(X\ /=\ c\,\bar{t}_n)$. This is the case because $X\eta = X\sigma\sigma_1\eta = X\sigma_1\eta = (d\,\overline{Z}_m)\eta$, where the first equality holds because of the assumption $\eta \in Sol_{\mathscr{H}}(\sigma\sigma_1)$ and the third equality holds by construction of σ_1.
H13 Trivial, for the same reason as **H6**.

(6) **Local Completeness Property** for $\mathscr{URS}$-free steps: Recall the set of unsafe *strs* $\mathscr{URS} = \{\mathbf{OH3}, \mathbf{OH7}, \mathbf{H13}\}$ defined in Subsection 2.4.2. Assume a $\mathscr{H}$ store $\Pi \,\Box\, \sigma$ and a set $\mathscr{X} \subseteq odvar_{\mathscr{H}}(\Pi)$, such that $\Pi \,\Box\, \sigma$ is $\mathscr{URS}$-irreducible but not in $\mathscr{X}$-solved form. We must prove that $WTSol_{\mathscr{D}}(\Pi \,\Box\, \sigma)$ is a subset of the union

$$\bigcup\{WTSol_{\mathscr{D}}(\exists\overline{Y'}(\Pi'\Box\sigma')) \mid \Pi\Box\sigma \Vdash_{\mathscr{H},\mathscr{X}} \Pi'\Box\sigma', \overline{Y'} = var(\Pi'\Box\sigma') \setminus var(\Pi\Box\sigma)\}$$

Given any well-typed solution $\eta \in WTSol_{\mathscr{H}}(\Pi \,\Box\, \sigma)$ (which satisfies in particular $\sigma\eta = \eta$), we must find $\Pi' \,\Box\, \sigma'$ and η' such that

$$(\ddagger)\ \Pi \,\Box\, \sigma \Vdash_{\mathscr{H},\mathscr{X}} \Pi' \,\Box\, \sigma',\ \eta' \in WTSol_{\mathscr{H}}(\Pi' \,\Box\, \sigma'),\ \eta =_{\setminus\overline{Y'}} \eta'$$

so that $\eta \in WTSol_{\mathscr{H}}(\exists\overline{Y'}(\Pi' \,\Box\, \sigma'))$ will be ensured. Because of the assumptions on $\Pi \,\Box\, \sigma$, there must be some *str* $\mathbf{H}_i \notin \mathscr{URS}$ that can be used to transform $\Pi \,\Box\, \sigma$. Below we analyze all the possibilities for $\mathbf{H}_i$, considering all the *strs* shown in Table 1 in Subsection 2.4.2 except **OH3**, **OH7,** and **H13**. In all the cases we conclude that the conditions (‡) displayed above can be ensured. When considering different *strs* that can be alternatively applied to one and the same store (as, e.g., **H1** and **H2**) we group all the possibilities within the

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

same case, arguing that some rule in the group can be chosen to transform $\Pi \square \sigma$ ensuring (‡). In all the cases, we assume that the stores $\Pi \square \sigma$ and $\Pi' \square \sigma'$ occurring in (‡) have exactly the form displayed for the corresponding transformation in Table 1, we note the selected atomic constraint as π, and we silently use the fact that the constraints and variables within any store are not affected by the substitution kept in that store.

H1, H2 In this case π is $(t == s) \to! R$, $\eta \in WTSol_{\mathcal{H}}(t == s \to! R, \Pi \square \sigma)$ and $\overline{Y'} = \emptyset$. Because of $\eta \in WTSol_{\mathcal{H}}(\pi)$, one of the two following subcases must hold:

(a) $\eta(R) = \texttt{true}$ and $\eta \in WTSol_{\mathcal{H}}(t == s)$ or else
(b) $\eta(R) = \texttt{false}$ and $\eta \in WTSol_{\mathcal{H}}(t \mathrel{/=} s)$

Assume that subcase (a) holds. Then, (‡) can be ensured by transforming the given store with **H1** and proving $\eta' = \eta \in WTSol_{\mathcal{H}}(\Pi\sigma_1 \square \sigma\sigma_1)$. Note that Lemma 8 can be applied with $\overline{Y'} = \emptyset$, $\eta' = \eta$ and $\sigma_1 = \{R \mapsto true\}$, because the condition $\sigma_1\eta = \eta$ follows trivially from $\eta(R) = \texttt{true}$. Then, $\eta \in Sol_{\mathcal{H}}(\Pi\sigma_1) \cap Sol_{\mathcal{H}}(\sigma\sigma_1)$ is ensured by Lemma 8, and η obviously remains a well-typed solution.

Assume now that subcase (b) holds. Then a similar argument can be used, but choosing **H2** instead of **H1**.

H3 In this case π is $h\,\overline{t}_m == h\,\overline{s}_m$ and (‡) can be ensured by choosing to transform the given store with **H3** and taking $\overline{Y'} = \emptyset$ and $\sigma' = \sigma$. Note that h must be m-transparent because of the $\mathcal{URS}$-freeness assumption, and the Transparency Lemma 2 can be applied to ensure that η remains a well-typed solution of the new store.

H4 In this case π is $t == X$, where t is not a variable, and (‡) can be trivially ensured by choosing to transform the given store with **H4** and taking $\overline{Y'} = \emptyset$ and $\sigma' = \sigma$.

H5 In this case π is $t == X$, with $X \notin \mathcal{X}, X \notin var(t), X \neq t$. Moreover, $\eta \in WTSol_{\mathcal{H}}(X == t, \Pi \square \sigma)$ and $\overline{Y'} = \emptyset$. Then (‡) can be ensured by transforming the given store with **H5** and proving $\eta' = \eta \in WTSol_{\mathcal{H}}(tot(t), \Pi\sigma_1 \square \sigma\sigma_1)$. The assumption $\eta \in WTSol_{\mathcal{H}}(\pi)$ means that $\eta(X) = t\eta$ is a total pattern, so that $\eta(Y)$ is also a total pattern for each variable $Y \in var(t)$. In these conditions, $\eta \in Sol_{\mathcal{H}}(tot(t))$ and $\sigma_1\eta = \eta$ holds for $\sigma_1 = \{X \mapsto t\}$. This allows to apply Lemma 8 with $\overline{Y'} = \emptyset$, $\eta' = \eta$ and σ_1, ensuring that $\eta \in Sol_{\mathcal{H}}(\Pi\sigma_1) \cap Sol_{\mathcal{H}}(\sigma\sigma_1)$. Obviously, η remains a well-typed solution.

H7 In this case, π is $h\,\overline{t}_m \mathrel{/=} h\,\overline{s}_m$. Because of $\eta \in WTSol_{\mathcal{H}}(\pi)$, there must be some index i such that $1 \leqslant i \leqslant m$ and $\eta \in WTSol_{\mathcal{H}}(t_i \mathrel{/=} s_i)$. Then (‡) can be ensured by choosing to transform the given store with **H7** and this particular value of i, and taking $\overline{Y'} = \emptyset$, $\sigma' = \sigma$. Note that h must be m-transparent because of the $\mathcal{URS}$-freeness assumption, and the Transparency Lemma 2 can be applied to ensure that η remains a well-typed solution of the new store.

H8 In this case π is $h\,\overline{t}_n$ /= $h'\,\overline{s}_m$ with $h \neq h'$ or $n \neq m$, and (‡) can be trivially ensured by choosing to transform the given store with **H8**, taking $\overline{Y'} = \emptyset$ and $\sigma' = \sigma$.

H10 This is a trivial case, similar to **H4**.

H11, H12 In this case π is X /= $c\,\overline{t}_n$, with $X \notin \mathscr{X}$, $c \in DC^n$ and $\mathscr{X} \cap var(c\,\overline{t}_n) \neq \emptyset$, $\eta \in WTSol_{\mathscr{H}}(X$ /= $c\,\overline{t}_n\,\Pi \,\Box\, \sigma)$. Because of $\eta \in WTSol_{\mathscr{H}}(\pi)$, one of the two following subcases must hold for $\eta(X)$:

(a) $\eta(X) = c\,\overline{s}_n$, where s_i/= $t_i\eta$ holds for some $1 \leqslant i \leqslant n$.

(b) $\eta(X) = d\,\overline{s}_m$, where $d \in DC^m$ belongs to the same datatype as c, but $d \neq c$.

Assume that subcase (a) holds. Then (‡) can be ensured by choosing to transform the given store with **H11** and a particular value of i such that s_i/= $t_i\eta$ holds, taking $\overline{Y'} = \overline{Z}_n$, defining η' as the valuation that satisfies $\eta'(Z_j) = s_j$ for all $1 \leqslant j \leqslant n$ and $\eta'(Y) = \eta(Y)$ for any other variable Y and proving $\eta' \in WTSol_{\mathscr{H}}((Z_i$ /= $t_i, \Pi)\sigma_1 \,\Box\, \sigma\sigma_1)$.

Obviously, $\eta =_{\setminus \overline{Y'}} \eta'$. Moreover, $\sigma_1\eta' = \eta'$, since $X\sigma_1\eta' = (c\,\overline{s}_n)\eta' = c\,\overline{s}_n = X\eta = X\eta'$ and $Y\sigma_1\eta' = Y\eta'$ for any variable $Y \neq X$. Therefore, Lemma 8 can be applied to ensure that $\eta' \in Sol_{\mathscr{H}}(\Pi\sigma_1) \cap Sol_{\mathscr{H}}(\sigma\sigma_1)$. Since η was a well-typed solution and data constructors have the transparency property (see Subsection 2.1), η' can also be well-typed under appropriated type assumptions for the new variables $\overline{Y'} = \overline{Z}_n$ introduced by the transformation step. It only remains to prove that $\eta' \in Sol_{\mathscr{H}}((Z_i$/=$t_i)\sigma_1)$. This can be reasoned by a chain of equivalences, ending with the condition known to hold in subcase (a):

$$\eta' \in Sol_{\mathscr{H}}((Z_i /{=} t_i)\sigma_1) \Leftrightarrow_{(1)} \sigma_1\eta' \in Sol_{\mathscr{H}}(Z_i /{=} t_i) \Leftrightarrow_{(2)}$$
$$\eta' \in Sol_{\mathscr{H}}(Z_i /{=} t_i) \Leftrightarrow \eta'(Z_i) /{=} t_i\eta' \Leftrightarrow_{(3)} s_i /{=} t_i\eta'$$

Note that (1) holds because of Lemma 3, (2) holds because $\sigma_1\eta' = \eta'$, and (3) holds by construction of η'. This finishes the proof for this subcase.

Finally, assume now that subcase (b) holds. Then (‡) can be ensured by choosing to transform the given store with **H12** and the particular data constructor $d \in DC^m$ for which we know that $\eta(X) = d\,\overline{s}_m$, taking $\overline{Y'} = \overline{Z}_m$, defining η' as the valuation that satisfies $\eta'(Z_j) = s_j$ for all $1 \leqslant j \leqslant m$ and $\eta'(Y) = \eta(Y)$ for any other variable Y and proving $\eta' \in WTSol_{\mathscr{H}}(\Pi\sigma_1 \,\Box\, \sigma\sigma_1)$.

Obviously, $\eta =_{\setminus \overline{Z}_m} \eta'$. Moreover, $\sigma_1\eta' = \eta'$ can be easily checked, as in subcase (a). Therefore, Lemma 8 can be applied to ensure that $\eta' \in Sol_{\mathscr{H}}(\Pi\sigma_1) \cap Sol_{\mathscr{H}}(\sigma\sigma_1)$. Finally, since η was a well-typed solution, η' is clearly also well-typed under appropriated type assumptions for the new variables $\overline{Y'} = \overline{Z}_n$ introduced by the transformation step.

Using items (1) to (6) above and Lemma 5, we can now claim that $solve^{\mathscr{H}}$ satisfies the requirements for solvers enumerated in Definition 6, except that the **Completeness**

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Property is guaranteed to hold only for safe (i.e., $\mathcal{URS}$-free) solver invocations and the **Discrimination Property** has not been proved yet.

The remark in item (1) of the proof of Lemma 5 allows to rephrase the *Discrimination Property* as follows: if a given $\mathcal{H}$ store $\Pi \Box \sigma$ satisfies neither (a) $\mathcal{X} \cap odvar(\Pi) \neq \emptyset$ nor (b) $\mathcal{X} \cap var(\Pi) = \emptyset$, then $\Pi \Box \sigma$ can be reduced by some $\Vdash_{\mathcal{H},\mathcal{X}}$ transformation. Assume that $\Pi \Box \sigma$ satisfies neither (a) nor (b). Because of $\neg(b)$, there must be some $\pi \in \Pi$ such that (c) $\mathcal{X} \cap var(\pi) \neq \emptyset$. Because of $\neg(a)$, this π must satisfy (d) $\mathcal{X} \cap odvar(\pi) = \emptyset$, which together with (c) entails (e) $\mathcal{X} \cap cvar(\pi) \neq \emptyset$. Using $(d), (e)$ and reasoning by case distinction on the syntactic form of π, we find in all the cases some $\Vdash_{\mathcal{H},\mathcal{X}}$ transformation which can be used to transform the store $\Pi \Box \sigma$ taking π as the selected atomic constraint. The cases are as follows:

- π is $(t == s) \to! R$. In this case the store can be transformed by means of **H1** or **H2**.
- π is $h\,\bar{t}_m == h\,\bar{s}_m$. In this case the store can be transformed by means of **H3**.
- π is $t == X$ with $t \notin \mathcal{V}ar$. In this case the store can be transformed by means of **H4**.
- π is $X == t$ with $X \notin var(t)$, $X \neq t$. Because of (d) above we know that $X \notin \mathcal{X}$, and the store can be transformed by means of **H5**.
- π is $X == t$ with $X \in var(t)$, $X \neq t$. In this case the store can be transformed by means of **H6**.
- π is $h\,\bar{t}_m$ /= $h\,\bar{s}_m$. In this case the store can be transformed by means of **H7**.
- π is $h\,\bar{t}_n$ /= $h'\,\bar{s}_m$ with $h \neq h'$ or $n \neq m$. In this case the store can be transformed by means of **H8**.
- π is t /= t with $t \in \mathcal{V}ar \cup DC \cup DF \cup SPF$. In this case the store can be transformed by means of **H9**.
- π is t /= X with $t \notin \mathcal{V}ar$. In this case the store can be transformed by means of **H10**.
- π is X /= $c\,\bar{t}_n$, with $c \in DC^n$. Because of $(d), (e)$ above we know that $X \notin \mathcal{X}$ and $\mathcal{X} \cap var(c\,\bar{t}_n) \neq \emptyset$. Therefore, the store can be transformed by means of **H11** or **H12**.
- π is X /= $h\,\bar{t}_m$ with $h \notin DC^m$. Because of $(d), (e)$ above we know that $X \notin \mathcal{X}$ and $\mathcal{X} \cap var(h\,\bar{t}_m) \neq \emptyset$. Therefore, the store can be transformed by means of **H13**.

This completes the proof of the Discrimination Property and the Theorem. □

We refrain to include in this Appendix a proof of Theorem 3, stated in Subsection 2.6 and ensuring the properties required for the solver $solve^{\mathcal{M}}$. The proof would follow exactly the same pattern as the previous one, but with much simpler arguments, since the *sts* for $\mathcal{M}$ stores involves no decompositions. Actually, this *sts* is finitely branching, terminating, locally sound, and locally complete. Therefore, Lemma 5 can be applied to ensure all the properties required for solvers, including unrestricted completeness.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

We end this subsection with the proof of Theorem 2, ensuring that the amalgamated sums presented in Subsection 2.5 are well-defined domains behaving as a conservative extension of their components.

Proof of Theorem 2
Assume $\mathscr{S} = \mathscr{D}_1 \oplus \cdots \oplus \mathscr{D}_n$ of signature Σ constructed as the amalgamated sum of n pairwise joinable domains $\mathscr{D}_i$ of signatures Σ_i, $1 \leqslant i \leqslant n$. Note that the information ordering $\sqsubseteq$ introduced in Subsection 2.2 has the same syntactic definition for any specific domain signature. Note also that any arguments concerning well typing needed for this proof can refer to the principal type declarations within signature Σ, which includes those within signature Σ_i for all $1 \leqslant i \leqslant n$. Let us now prove the four claims of the theorem in order.

(1) $\mathscr{S}$ is well defined as a constraint domain; i.e., the interpretations of primitive function symbols $p \in SPF$ in $\mathscr{S}$ satisfy the four conditions listed in Definition 1 from Subsection 2.3. We consider them one by one, assuming that p is not the primitive == except in the fourth condition.
   (a) **Polarity**: Assume $p \in SPF^m$ and $\overline{t}_m, \overline{t'}_m, t, t' \in \mathscr{U}_{\mathscr{S}}$ such that $p^{\mathscr{S}}\, \overline{t}_m \to t$, $\overline{t}_m \sqsubseteq \overline{t'}_m$ and $t \sqsupseteq t'$. In case that t is $\bot$, we trivially conclude $p^{\mathscr{S}}\, \overline{t'}_n \to t'$ because t' must be also $\bot$. Otherwise, by the first assumption and the definition of $p^{\mathscr{S}}$, there must be some $1 \leqslant i \leqslant n$ and some $\overline{t''}_m, t'' \in \mathscr{U}_{\mathscr{D}_i}$ such that $\overline{t''}_m \sqsubseteq \overline{t}_m$, $t'' \sqsupseteq t$ and $p^{\mathscr{D}_i}\, \overline{t''}_m \to t''$. Since $\overline{t''}_m \sqsubseteq \overline{t}_m \sqsubseteq \overline{t'}_m$ and $t'' \sqsupseteq t \sqsupseteq t'$, $p^{\mathscr{D}_i}\, \overline{t''}_m \to t''$ implies $p^{\mathscr{S}}\, \overline{t'}_m \to t'$ by definition of $p^{\mathscr{S}}$.
   (b) **Radicality**: Assume $p \in SPF^m$ and $\overline{t}_m, t \in \mathscr{U}_{\mathscr{S}}$ such that $p^{\mathscr{S}}\, \overline{t}_m \to t$ and t is not $\bot$. By the definition of $p^{\mathscr{S}}$ there must be some $1 \leqslant i \leqslant n$ and some $\overline{t''}_m$, $t'' \in \mathscr{U}_{\mathscr{D}_i}$ such that $\overline{t''}_m \sqsubseteq \overline{t}_m$, $t'' \sqsupseteq t$ and $p^{\mathscr{D}_i}\, \overline{t''}_m \to t''$. By the radicality condition for $\mathscr{D}_i$, there must be some total $t' \in \mathscr{U}_{\mathscr{D}_i}$ such that $p^{\mathscr{D}_i}\, \overline{t''}_m \to t' \sqsupseteq t''$. Note that $t' \sqsupseteq t'' \sqsupseteq t$, and because of $\overline{t''}_m \sqsubseteq \overline{t}_m$ and $t' \sqsupseteq t'$, $p^{\mathscr{D}_i}\, \overline{t''}_m \to t'$ implies $p^{\mathscr{S}}\, \overline{t}_m \to t'$ by definition of $p^{\mathscr{S}}$.
   (c) **Well-typedness**: Assume $p \in SPF^m$, a monomorphic instance $\overline{\tau'}_m \to \tau'$ of p's principal type and $\overline{t}_m, t \in \mathscr{U}_{\mathscr{S}}$ such that $\Sigma \vdash_{WT} \overline{t}_m :: \overline{\tau'}_m$ and $p^{\mathscr{S}}\, \overline{t}_m \to t$. In case that t is $\bot$, the type judgement $\Sigma \vdash_{WT} \bot :: \tau'$ holds trivially. Otherwise, by the assumption $p^{\mathscr{S}}\, \overline{t}_m \to t$ and the definition of $p^{\mathscr{S}}$ there exist $1 \leqslant i \leqslant n$ and $\overline{t'}_m, t' \in \mathscr{U}_{\mathscr{D}_i}$ such that $\overline{t'}_m \sqsubseteq \overline{t}_m$, $t' \sqsupseteq t$ and $p^{\mathscr{D}_i}\, \overline{t'}_m \to t'$. Moreover, since $\overline{t'}_m \sqsubseteq \overline{t}_m$ the assumption $\Sigma \vdash_{WT} \overline{t}_m :: \overline{\tau'}_m$ and the Type Preservation Lemma 1 imply $\Sigma \vdash_{WT} \overline{t'}_m :: \overline{\tau'}_m$ Then, the well-typedness assumption for $\mathscr{D}_i$ guarantees $\Sigma \vdash_{WT} t' :: \tau'$, which implies $\Sigma \vdash_{WT} t :: \tau'$ because of $t \sqsubseteq t'$ and Lemma 1.
   (d) **Strict Equality**: The primitive == (in case that it belongs to SPF) is interpreted as *strict equality* over $\mathscr{U}_{\mathscr{S}}$. This is automatically guaranteed by the amalgamated sum construction.

(2) Given an index $1 \leqslant i \leqslant n$, a primitive function symbol $p \in SPF_i^m$ and values $\overline{t}_m, t \in \mathscr{U}_{\mathscr{D}_i}$, we must prove: $p^{\mathscr{D}_i}\, \overline{t}_m \to t$ iff $p^{\mathscr{S}}\, \overline{t}_m \to t$. By definition of $p^{\mathscr{S}}$, we know that $p^{\mathscr{S}}\, \overline{t}_m \to t$ holds iff there are some $\overline{t'}_m, t' \in \mathscr{U}_{\mathscr{D}_i}$ such that $\overline{t'}_m \sqsubseteq \overline{t}_m$, $t' \sqsupseteq t$ and $p^{\mathscr{D}_i}\, \overline{t'}_m \to t'$. But this condition is equivalent to $p^{\mathscr{D}_i}\overline{t}_m \to t$ because $p^{\mathscr{D}_i}$ satisfies the polarity property.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

(3) Given an index $1 \leqslant i \leqslant n$, a set of primitive constraints $\Pi \subseteq APCon_{\mathcal{D}_i}$ and a valuation $\eta \in Val_{\mathcal{D}_i}$, we will prove: $\eta \in Sol_{\mathcal{D}_i}(\Pi) \Leftrightarrow \eta \in Sol_{\mathcal{S}}(\Pi)$. The corresponding equivalence for the case of well-typed solutions follows then easily. Since

$$\eta \in Sol_{\mathcal{D}_i}(\Pi) \Leftrightarrow \forall \pi \in \Pi : \eta \in Sol_{\mathcal{D}_i}(\pi) \Leftrightarrow \forall \pi \in \Pi : \eta \in Sol_{\mathcal{S}}(\pi) \Leftrightarrow \eta \in Sol_{\mathcal{S}}(\Pi)$$

it suffices to prove the equivalence

$$(\star)\, \eta \in Sol_{\mathcal{D}_i}(\pi) \Leftrightarrow \eta \in Sol_{\mathcal{S}}(\pi)$$

for a fixed $\pi \in \Pi$. Note that π must have the form $p\,\bar{t}_m \;\rightarrow ! \, t$ for some $p \in SPF_i^m$, $\bar{t}_m \in Pat_{\mathcal{D}_i}$ and total $t \in Pat_{\mathcal{D}_i}$. In case that p is $==$, $(\star)$ is trivially true because $t_1\eta ==^{\mathcal{D}_i} t_2\eta \;\rightarrow !t\eta$ and $t_1\eta ==^{\mathcal{S}} t_2\eta \;\rightarrow !t\eta$ hold under the same conditions, as specified in Definition 1 from Subsection 2.3. In case that p is not $==$, let $\overline{t'}_m = \bar{t}_m\eta$ and $t' = t\eta$. If t' is not a total pattern, then neither $\eta \in Sol_{\mathcal{D}_i}(\pi)$ nor $\eta \in Sol_{\mathcal{S}}(\pi)$ hold. Otherwise,

$$\eta \in Sol_{\mathcal{D}_i}(\pi) \Leftrightarrow p^{\mathcal{D}_i}\overline{t'}_m \;\rightarrow\; t' \Leftrightarrow_{(\star\star)} p^{\mathcal{S}}\overline{t'}_m \;\rightarrow\; t' \Leftrightarrow \eta \in Sol_{\mathcal{S}}(\pi)$$

where the $(\star\star)$ step holds by the second item of this theorem, because $\overline{t'}_m, t' \in \mathcal{U}_{\mathcal{D}_i}$.

(4) Given an index $1 \leqslant i \leqslant n$, a set of $\mathcal{D}_i$-specific primitive constraints $\Pi \subseteq APCon_{\mathcal{D}_i}$ and a valuation $\eta \in Val_{\mathcal{S}}$, we will prove: $\eta \in Sol_{\mathcal{S}}(\Pi) \Leftrightarrow |\, \eta \,|_{\mathcal{D}_i} \in Sol_{\mathcal{D}_i}(\Pi)$. The corresponding equivalence for the case of well-typed solutions follows then easily.
First, we prove $\eta \in Sol_{\mathcal{S}}(\Pi) \Leftarrow |\, \eta \,|_{\mathcal{D}_i} \in Sol_{\mathcal{D}_i}(\Pi)$. Assume $|\, \eta \,|_{\mathcal{D}_i} \in Sol_{\mathcal{D}_i}(\Pi)$. Applying the previous item of this theorem, we obtain $|\, \eta \,|_{\mathcal{D}_i} \in Sol_{\mathcal{S}}(\Pi)$. Since $|\, \eta \,|_{\mathcal{D}_i} \sqsubseteq \eta$, we can apply the Monotonicity Lemma 4 and get $\eta \in Sol_{\mathcal{S}}(\Pi)$, as desired.
Now we prove $\eta \in Sol_{\mathcal{S}}(\Pi) \Rightarrow |\, \eta \,|_{\mathcal{D}_i} \in Sol_{\mathcal{D}_i}(\Pi)$. Assume $\eta \in Sol_{\mathcal{S}}(\Pi)$. Since Π is $\mathcal{D}_i$-specific, we can also assume that $\eta(X) \in \mathcal{U}_{\mathcal{D}_i}$ for all $X \in var(\Pi)$. Then $\eta(X) = |\, \eta \,|_{\mathcal{D}_i}(X)$ holds for all $X \in var(\Pi)$, and therefore $|\, \eta \,|_{\mathcal{D}_i} \in Sol_{\mathcal{S}}(\Pi)$, which implies $|\, \eta \,|_{\mathcal{D}_i} \in Sol_{\mathcal{D}_i}(\Pi)$, again because of the previous item of this theorem. □

A.2 Properties of the $CCLNC(\mathcal{D})$ calculus

The second part of the Appendix includes the proofs of the main results stated in Subection 3.6. First, we present an auxiliary result which is not stated in the main text of the article. The $(WT)Sol$ notation is intended to indicate that the lemma holds both for plain solutions and for well-typed solutions.

Lemma 9 (*Auxiliary Result for Checking Goal Solutions*)
Let $G \equiv \exists \overline{U}.\ P \,\square\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R$ be an admissible goal for a given $CFLP(\mathcal{C})$-program $\mathcal{P}$. Assume new variables $\overline{Y'}$ away from $\overline{U}$ and the other variables in G, and two valuations $\mu, \hat{\mu} \in Val_{\mathcal{C}}$ such that $\hat{\mu} =_{\setminus \overline{U}, \overline{Y'}} \mu$ and $\hat{\mu} \in (WT)Sol_{\mathcal{P}}(P \,\square\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)$. Then $\mu \in (WT)Sol_{\mathcal{P}}(G)$.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Proof
Consider $\hat{\hat{\mu}} \in Val_{\mathscr{C}}$ univocally defined by the two conditions $\hat{\hat{\mu}} =_{\setminus \overline{Y'}} \hat{\mu}$ and $\hat{\hat{\mu}} =_{\overline{Y'}} \mu$. By hypothesis, $\hat{\mu} \in (WT)Sol_{\mathscr{P}}(P \Box C \Box M \Box H \Box F \Box R)$ and the variables $\overline{Y'}$ do not occur in G. Therefore, $\hat{\hat{\mu}} \in (WT)Sol_{\mathscr{P}}(P \Box C \Box M \Box H \Box F \Box R)$ is ensured by the construction of $\hat{\hat{\mu}}$. Recalling Definition 10 (see Subsection 3.6), we only need to prove $\hat{\hat{\mu}} =_{\setminus \overline{U}} \mu$ in order to conclude $\mu \in (WT)Sol_{\mathscr{P}}(G)$. In fact, given any variable $X \notin \overline{U}$, either $X \in \overline{Y'}$ or $X \notin \overline{Y'}$. In the first case, $\hat{\hat{\mu}}(X) = \mu(X)$ by construction of $\hat{\hat{\mu}}$. In the second case, $\hat{\hat{\mu}}(X) = \hat{\mu}(X)$ by construction of $\hat{\hat{\mu}}$ and $\hat{\mu}(X) = \mu(X)$ because of one of the hypothesis. □

Next, we present the proof of Theorem 4 which guarantees *local* soundness and completeness for the *one-step* transformation of a given goal.

Proof of Theorem 4
Assume a given program $\mathscr{P}$, an admissible goal G for $\mathscr{P}$ which is not in solved form, and a rule **RL** applicable to a selected part γ of G. The claim that there are finitely many possible transformations $G \Vdash_{\mathbf{RL},\gamma,\mathscr{P}} G'_j$ $(1 \leqslant j \leqslant k)$ can be trivially checked by inspecting all the rules in Tables 3, 4, 7, and 8 one by one. We must prove two additional claims:

(1) **Local Soundness:** $Sol_{\mathscr{P}}(G) \supseteq \bigcup_{j=1}^{k} Sol_{\mathscr{P}}(G'_j)$.
(2) **Limited Local Completeness:** $WTSol_{\mathscr{P}}(G) \subseteq \bigcup_{j=1}^{k} WTSol_{\mathscr{P}}(G'_j)$, provided that the application of **RL** to the selected part γ of G is *safe*; i.e., it is neither an opaque application of **DC** nor an application of a rule from Table 8 involving an incomplete solver invocation.

Claims (1) and (2) must be proved for each **RL** separately. In case that **RL** is some of the rules displayed in Table 3, proving (1) and (2) involves building suitable witnesses as multisets of $CRWL(\mathscr{C})$ proof trees, using techniques originally stemming from González-Moreno *et al.* (1996, 1999) and later extended to *CFLP* programs without domain cooperation in López-Fraguas *et al.* (2004). In case that **RL** is some of the rules shown in Tables 4, 7, and 8, proving (1) and (2) requires almost no work with building witnesses.

We will consider rules **DF** and **FC** as representatives for Table 3, and most of the rules from Tables 4, 7, and 8, which are the main novelty in this paper. When dealing with each rule **RL**, we will assume that G resp. G'_j are exactly as the original resp. transformed goal as displayed in the presentation of **RL** in Subsection 3.2, 3.3, or 3.4. In our reasonings we will use the notation $\mathscr{M} : \mathscr{P} \vdash_{CRWL(\mathscr{C})} (P \Box C)\mu'$ to indicate that the witness $\mathscr{M}$ is a multiset of $CRWL(\mathscr{C})$ proof trees that prove $(P \Box C)\mu'$ from program $\mathscr{P}$, using the inference rules of the $CRWL(\mathscr{C})$ logic presented in López-Fraguas *et al.* (2004).

A.2.1 Selected rules from Table 3

Rule **DF, Defined Function**. In this case, γ is a production $f\,\overline{e}_n \to t$.

(1) **Local Soundness:** Assume $\mu \in Sol_{\mathscr{P}}(G'_j)$ for some $1 \leqslant j \leqslant k$. Then there exists $\mu' =_{\setminus \overline{Y},\overline{U}} \mu$ such that $\mu' \in Sol_{\mathscr{P}}(\overline{e_n \to t_n}, r \to t, P \Box C', C \Box M \Box H \Box F \Box R)$.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

From this we deduce that $\mu' \in Sol_{\mathcal{C}}(M \Box H \Box F \Box R)$ and $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (\overline{e_n \to t_n}, r \to t, P \Box C', C)\mu'$ for a suitable witness $\mathcal{M}'$. A part of $\mathcal{M}'$ proves $(\overline{e_n \to t_n}, r \to t, C')\mu'$, which allows to deduce $(f\,\overline{e}_n \to t)\mu'$ using the $CRWL(\mathcal{C})$ inference rule which deals with defined functions. Therefore, $\mathcal{M}'$ can be used to build another witness $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (f\,\overline{e}_n \to t, P \Box C)\mu'$. Since $\mu' =_{\backslash \overline{U}} \mu$, we can conclude that $\mu \in Sol_{\mathcal{P}}(G)$.

(2) **Limited Local Completeness:** Assume $\mu \in WTSol_{\mathcal{P}}(G)$. Then there is some $\mu' =_{\backslash \overline{U}} \mu$ such that $\mu' \in WTSol_{\mathcal{P}}(f\,\overline{e}_n \to t, P \Box C \Box M \Box H \Box F \Box R)$. Then, $\mu' \in WTSol_{\mathcal{C}}(M \Box H \Box F \Box R)$ and $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (f\,\overline{e}_n \to t, P \Box C)\mu'$ for a suitable witness $\mathcal{M}$. Note that $\mathcal{M}$ must include a $CRWL(\mathcal{C})$ proof tree $\mathcal{T}$ proving the production $(f\,\overline{e}_n \to t)\mu'$ using some instance of $Rl : f\,\overline{t_n} \to r \Leftarrow C'$, suitably chosen as a variant of some $\mathcal{P}$ rule with new variables $\overline{Y} = var(Rl)$. Let us choose j so that G'_j is the result of applying **DF** with $f\,\overline{e}_n \to t$ as the selected part of G and Rl as the selected $\mathcal{P}$ rule for f. Consider a well-typed $\mu'' \in Val_{\mathcal{C}}$ that instantiates the variables in $\overline{Y}$ as required by the proof tree $\mathcal{T}$, and instantiates any other variable V to $\mu'(V)$. By suitably reusing parts of $\mathcal{M}$, it is possible to build a witness $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (\overline{e_n \to t_n}, r \to t, P \Box C', C)\mu''$. Since $\mu'' =_{\backslash \overline{Y}, \overline{U}} \mu$, we can conclude that $\mu \in WTSol_{\mathcal{P}}(G'_j)$.

Rule **FC, Flatten Constraint**. In this case, γ is an atomic constraint $p\,\overline{e}_n \to!\, t$ such that some e_i is not a pattern and $k = 1$. We write G' instead of G'_1. For the sake of simplicity, we consider $p\,e_1\,t_2 \to!\, t$, where e_1 is not a pattern. The presentation of the rule is then as in Table 3 with $n = 2$, $m = 1$.

(1) **Local Soundness:** Assume $\mu \in Sol_{\mathcal{P}}(G')$. Then there exists $\mu' =_{\backslash V_1, \overline{U}} \mu$ such that $\mu' \in Sol_{\mathcal{P}}(e_1 \to V_1, P \Box p\,V_1\,t_2 \to!\, t, C \Box M \Box H \Box F \Box R)$. Then, we get $\mu' \in Sol_{\mathcal{C}}(M \Box H \Box F \Box R)$ and $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (e_1 \to V_1, P \Box p\,V_1\,t_2 \to!\, t, C)\mu'$ for a suitable witness $\mathcal{M}'$. A part of $\mathcal{M}'$ proves $(e_1 \to V_1, p\,V_1\,t_2 \to!\, t)\mu'$, which allows to deduce $(p\,e_1\,t_2 \to!\, t)\mu'$ using the $CRWL(\mathcal{C})$ inference rule which deals with primitive functions. Therefore, $\mathcal{M}'$ can be used to build another witness $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \Box p\,e_1\,t_2 \to!\, t, C)\mu'$. Since $\mu' =_{\backslash \overline{U}} \mu$, we can conclude that $\mu \in Sol_{\mathcal{P}}(G)$.

(2) **Limited Local Completeness:** Assume $\mu \in WTSol_{\mathcal{P}}(G)$. Then there is some $\mu' =_{\backslash \overline{U}} \mu$ such that $\mu' \in WTSol_{\mathcal{P}}(P \Box p\,e_1\,t_2 \to!\, t, C \Box M \Box H \Box F \Box R)$. Then, $\mu' \in WTSol_{\mathcal{C}}(M \Box H \Box F \Box R)$ and $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \Box p\,e_1\,t_2 \to!\, t, C)\mu'$ for a suitable witness $\mathcal{M}$. Note that $\mathcal{M}$ must include a $CRWL(\mathcal{C})$ proof tree $\mathcal{T}$ proving the atomic constraint $(p\,e_1\,t_2 \to!\, t)\mu'$. A part of $\mathcal{T}$ must prove a production of the form $e_1\mu' \to t_1$ for some suitable pattern t_1. Consider a well-typed $\mu'' \in Val_{\mathcal{C}}$ such that $\mu''(V_1) = t_1$ and $\mu'' =_{\backslash V_1} \mu'$. By suitably reusing parts of $\mathcal{M}$, it is possible to build a witness $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (e_1 \to V_1, P \Box p\,V_1\,t_2 \to\, !\, t, C)\mu''$. Since $\mu'' =_{\backslash V_1, \overline{U}} \mu$, we can conclude that $\mu \in WTSol_{\mathcal{P}}(G')$.

A.2.2 Rules from Table 4

Rule **SB, Set Bridges**. In this case, γ is a primitive atomic constraint π which can be used to compute bridges, and $k = 1$. We write G' instead of G'_1. The application of

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

the rule computes $\exists\overline{V'}\,B' = bridges^{\mathcal{D}\to\mathcal{D}'}(\pi, B_M) \neq \emptyset$, where $\mathcal{D} = \mathcal{FD}$ and $\mathcal{D}' = \mathcal{R}$ or vice versa, according to the two cases (i) and (ii) explained in Table 4.

(1) **Local Soundness:** Assume $\mu \in Sol_{\mathcal{P}}(G')$. Then there exists $\mu' =_{\setminus \overline{V'},\overline{U}} \mu$ such that $\mu' \in Sol_{\mathcal{P}}(P \,\square\, \pi,\ C \,\square\, M' \,\square\, H \,\square\, F \,\square\, R)$. Therefore, $\mu' \in Sol_{\mathcal{C}}(M' \,\square\, H \,\square\, F \,\square\, R)$ and $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \,\square\, \pi,\ C)\mu'$ for a suitable witness $\mathcal{M}'$. Since M' is B', M, we get $\mu' \in Sol_{\mathcal{C}}(M \,\square\, H \,\square\, F \,\square\, R)$ and $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \,\square\, \pi,\ C)\mu'$, which implies $\mu \in Sol_{\mathcal{P}}(G)$ because of Lemma 9.

(2) **Limited Local Completeness:** Assume $\mu \in WTSol_{\mathcal{P}}(G)$. Then there is some $\mu' =_{\setminus \overline{U}} \mu$ such that $\mu' \in WTSol_{\mathcal{P}}(P \,\square\, \pi,\ C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)$. Therefore, $\mu' \in WTSol_{\mathcal{C}}(M \,\square\, H \,\square\, F \,\square\, R)$ and $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \,\square\, \pi,\ C)\mu'$ for a suitable witness $\mathcal{M}$. Since π is primitive, these conditions imply $\mu' \in WTSol_{\mathcal{C}}(\pi \wedge B_M)$. By item (2) of Proposition 1 from Subsection 3.3, we know that $\overline{V'}$ are new fresh variables and $WTSol_{\mathcal{C}}(\pi \wedge B_M) \subseteq WTSol_{\mathcal{C}}(\exists\overline{V'}(\pi \wedge B_M \wedge B'))$. From this we can conclude that $\mu' \in WTSol_{\mathcal{C}}(\exists\overline{V'}(\pi \wedge B_M \wedge B'))$ and therefore there is some $\mu'' =_{\setminus \overline{V'}} \mu'$ such that $\mu''' \in WTSol_{\mathcal{C}}(\pi \wedge B_M \wedge B')$. Since $\overline{V'}$ are new variables not occurring in G, it is easy to check that $\mu'' \in WTSol_{\mathcal{C}}(M' \,\square\, H \,\square\, F \,\square\, R)$ and $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \,\square\, \pi,\ C)\mu''$, which ensures $\mu \in WTSol_{\mathcal{P}}(G')$.

Rule **PP, Propagate Projections**. In this case, γ is a primitive atomic constraint π which can be used to compute projections, and $k = 1$. We write G' instead of G'_1. The application of the rule obtains G' from G by computing $\exists\overline{V'}\,\Pi' = proj^{\mathcal{D}\to\mathcal{D}'}(\pi, B_M) \neq \emptyset$, where $\mathcal{D} = \mathcal{FD}$ and $\mathcal{D}' = \mathcal{R}$ or vice versa, according to the two cases (i) and (ii) explained in Table 4. The reasonings for local soundness and limited local completeness are quite similar to those used in the case of rule **SB**, except that item (3) of Proposition 1 must be used in place of item (2).

Rule **SC, Submit Constraints**. In this case, γ is a primitive atomic constraint π and $k = 1$. We write G' instead of G'_1.

(1) **Local Soundness:** Assume $\mu \in Sol_{\mathcal{P}}(G')$. Then there exists $\mu' =_{\setminus \overline{U}} \mu$ such that $\mu' \in Sol_{\mathcal{P}}(P \,\square\, C \,\square\, M' \,\square\, H' \,\square\, F' \,\square\, R')$. Therefore, $\mu' \in Sol_{\mathcal{C}}(M' \,\square\, H' \,\square\, F' \,\square\, R')$ and $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \,\square\, C)\mu'$ for a suitable witness $\mathcal{M}'$. Because of the syntactic relationship between G and G' (see Table 4), $\mu' \in Sol_{\mathcal{C}}(M' \,\square\, H' \,\square\, F' \,\square\, R')$ amounts to $\mu' \in Sol_{\mathcal{C}}(M \,\square\, H \,\square\, F \,\square\, R)$ and $\mu' \in Sol_{\mathcal{C}}(\pi)$. Because of $\mu' \in Sol_{\mathcal{C}}(\pi)$, the witness $\mathcal{M}'$ can be expanded to another witness $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \,\square\, \pi,\ C)\mu'$. Thanks to this new witness we obtain $\mu' \in Sol_{\mathcal{P}}(P \,\square\, \pi,\ C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)$ and thus $\mu \in Sol_{\mathcal{P}}(G)$.

(2) **Limited Local Completeness:** Assume $\mu \in WTSol_{\mathcal{P}}(G)$. Then there is some $\mu' =_{\setminus \overline{U}} \mu$ such that $\mu' \in WTSol_{\mathcal{P}}(P \,\square\, \pi,\ C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)$. Therefore, $\mu' \in WTSol_{\mathcal{C}}(M \,\square\, H \,\square\, F \,\square\, R)$ and $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \,\square\, \pi,\ C)\mu'$ for a suitable witness $\mathcal{M}$. Because of the syntactic relationship between G and G' and the fact that π is primitive, we can conclude that $\mu' \in WTSol_{\mathcal{C}}(M' \,\square\, H' \,\square\, F' \,\square\, R')$. Let $\mathcal{M}'$ be the witness constructed from $\mathcal{M}$ by omitting the $CRWL(\mathcal{C})$ proof tree for $\pi\mu'$ which is part of $\mathcal{M}$. Then $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \,\square\, C)\mu'$. This allows to conclude $\mu' \in WTSol_{\mathcal{P}}(P \,\square\, C \,\square\, M' \,\square\, H' \,\square\, F' \,\square\, R')$ and thus $\mu \in WTSol_{\mathcal{P}}(G')$.

A.2.3 Rules from Table 7

Rule **IE, Infer Equalities**. This rule includes two similar cases. Here we will treat only the first one, the second one being completely analogous. The selected part γ is a pair of bridges of the form X #== RX, X' #== RX and $k = 1$. We write G' instead of G'_1.

(1) **Local Soundness:** Assume $\mu \in Sol_{\mathcal{P}}(G')$. Then there exists $\mu' =_{\setminus \overline{U}} \mu$ such that $\mu' \in Sol_{\mathcal{P}}(P \Box C \Box X$ #== $RX, M \Box H \Box X$ == $X', F \Box R)$. This implies two facts: first, $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \Box C)\mu'$ for a suitable witness $\mathcal{M}'$; and second, $\mu' \in Sol_{\mathcal{C}}(X$ #== $RX, M \Box H \Box X$ == $X', F \Box R)$. The second fact clearly implies $\mu' \in Sol_{\mathcal{C}}(X$ #== RX, X' #== $RX, M \Box H \Box F \Box R)$. Along with the witness $\mathcal{M}'$, this condition guarantees $\mu' \in Sol_{\mathcal{P}}(P \Box C \Box X$ #== RX, X' #== $RX, M \Box H \Box F \Box R)$ and hence $\mu \in Sol_{\mathcal{P}}(G)$.

(2) **Limited Local Completeness:** Assume $\mu \in WTSol_{\mathcal{P}}(G)$. Then there is some $\mu' =_{\setminus \overline{U}} \mu$ such that $\mu' \in WTSol_{\mathcal{P}}(P \Box C \Box X$ #== RX, X' #== $RX, M \Box H \Box F \Box R)$. This implies two facts: first, $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \Box C)\mu'$ for a suitable witness $\mathcal{M}$; and second, $\mu' \in WTSol_{\mathcal{C}}(X$ #== RX, X' #== $RX, M \Box H \Box F \Box R)$. The second fact clearly implies $\mu' \in WTSol_{\mathcal{C}}(X$ #== $RX, M \Box H \Box X$ == $X', F \Box R)$. Then, $\mu' \in WTSol_{\mathcal{P}}(P \Box C \Box X$ #== $RX, M \Box H \Box X$ == $X', F \Box R)$ holds thanks to the same witness $\mathcal{M}$, and therefore $\mu \in Sol_{\mathcal{P}}(G')$.

Rule **ID, Infer Disequalities**. This rule includes two similar cases. Here we consider only the first one, the second one being completely analogous. The selected part γ is an antibridge of the form X #/== u' placed within the M store, and $k = 1$. We write G' instead of G'_1. The application of the rule obtains G' from G by dropping X #/== u' from M and adding a semantically equivalent disequality constraint X /= u to the F store. The reasonings for local soundness and limited local completeness are very similar to those used in the case of rule **IE**.

A.2.4 Rules from Table 8

Here we present only the proofs concerning the two rules **FS** and **SF**. Note that the soundness and completeness properties of the $\mathcal{FD}$ solver refer to valuations over the universe $\mathcal{U}_{\mathcal{FD}}$, that must be related to valuations over the universe $\mathcal{U}_{\mathcal{C}}$ by means of Theorem 2 from Subsection 2.5, as we will see below. The same technique can be applied to the rules **MS** and **RS**. Rule **HS** can be also handled similarly to **FS**, but in this case Theorem 2 is not needed because the soundness and completeness properties of the extensible $\mathcal{H}$ solver refer directly to valuations over the universe $\mathcal{U}_{\mathcal{C}}$.

Rule **FS** **$\mathcal{FD}$-Constraint Solver (*black-box*)**. The selected part γ is the $\mathcal{FD}$ store F.

(1) **Local Soundness:** Let us choose G' as one of the finitely many goals G'_j such that $G \Vdash_{\mathbf{FS},\gamma,\mathcal{P}} G'_j$. Then $G' = \exists \overline{Y}', \overline{U}.(P \Box C \Box M \Box H \Box (\Pi' \Box \sigma_F) \Box R)@_{\mathcal{FD}}\sigma'$ for some $\exists \overline{Y}'(\Pi' \Box \sigma')$ chosen as one of the alternatives computed by the $\mathcal{FD}$

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

solver, i.e., such that $\Pi_F \Vdash_{solve^{\mathcal{FD}}} \exists \overline{Y'}(\Pi' \Box \sigma')$. Assume now $\mu \in Sol_{\mathcal{P}}(G')$. Then there exists $\mu' =_{\setminus \overline{Y'},\overline{U}} \mu$ such that

$$\mu' \in Sol_{\mathcal{P}}((P \Box C \Box M \Box H \Box (\Pi' \Box \sigma_F) \Box R) @_{\mathcal{FD}} \sigma')$$

for some $\exists \overline{Y'}(\Pi' \Box \sigma')$ such that $\Pi_F \Vdash_{solve^{\mathcal{FD}}} \exists \overline{Y'}(\Pi' \Box \sigma')$. Since $\Pi' \Box \sigma'$ is a store, we can assume $\Pi'\sigma' = \Pi'$ and deduce the following conditions:

$$(0)\, \mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \Box C)\sigma'\mu' \text{ for a suitable witness } \mathcal{M}'$$

$$(1)\, \mu' \in Sol_{\mathcal{C}}(\Pi_M\sigma' \Box \sigma_M \star \sigma') \quad (2)\, \mu' \in Sol_{\mathcal{C}}(\Pi_H\sigma' \Box \sigma_H \star \sigma')$$

$$(3)\, \mu' \in Sol_{\mathcal{C}}(\Pi'\sigma' \Box \sigma_F\sigma'), \text{ where } \Pi'\sigma' = \Pi' \quad (4)\, \mu' \in Sol_{\mathcal{C}}(\Pi_R\sigma' \Box \sigma_R \star \sigma')$$

In particular, (3) implies $\mu' \in Sol(\sigma_F\sigma')$, i.e.,

$$(5)\, \sigma_F\sigma'\mu' = \mu'$$

In order to conclude that $\mu \in Sol_{\mathcal{P}}(G)$, we show that the hypothesis of the auxiliary Lemma 9 hold for $\hat{\mu} = \mu'$. Clearly, $\hat{\mu} =_{\setminus \overline{U},\overline{Y'}} \mu$ and the new variables $\overline{Y'}$ are away from $\overline{U}$ and the other variables in G. We still have to prove that $\mu' \in Sol_{\mathcal{P}}(P \Box C \Box M \Box H \Box F \Box R)$.

- Proof of $\mu' \in Sol_{\mathcal{P}}(P \Box C)$: Because of the invariant properties of admissible goals, $(P \Box C) = (P \Box C)\sigma_F$. Using this equality and (5) we get $(P \Box C)\sigma'\mu' = (P \Box C)\sigma_F\sigma'\mu' = (P \Box C)\mu'$. Therefore, $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \Box C)\mu'$ follows from (0).
- Proof of $\mu' \in Sol_{\mathcal{C}}(S)$, S being any of the stores M, H, R: According to the choice of S we can use (1), (2) or (4) to conclude

$$(6)\, \mu' \in Sol_{\mathcal{C}}(\Pi_S\sigma') \quad \text{and} \quad (7)\, \mu' \in Sol(\sigma_S \star \sigma'), \text{ i.e., } (\sigma_S \star \sigma')\mu' = \mu'$$

  — Proof of $\mu' \in Sol_{\mathcal{C}}(\Pi_S)$: Because of the invariant properties of admissible goals, $\Pi_S = \Pi_S\sigma_F$. Then (6) is equivalent to $\mu' \in Sol_{\mathcal{C}}(\Pi_S\sigma_F\sigma')$. By applying the Substitution Lemma 3 we deduce $\sigma_F\sigma'\mu' \in Sol_{\mathcal{C}}(\Pi_S)$, which amounts to $\mu' \in Sol_{\mathcal{C}}(\Pi_S)$ because of (5).

  — Proof of $\mu' \in Sol(\sigma_S)$: Assume any variable $X \in vdom(\sigma_S)$. Then

$$X\mu' = X\sigma_S\sigma'\mu' = X\sigma_S\sigma_F\sigma'\mu' = X\sigma_S\mu'$$

  where the first equality holds because of (7), the second equality holds because the admissibility properties of G guarantee $\sigma_S \star \sigma_F = \sigma_S$, and the third equality holds because of (5).

- Proof of $\mu' \in Sol_{\mathcal{C}}(F)$: First, we claim that

$$(8)\ \ | \sigma'\mu' |_{\mathcal{FD}} \in Sol(\sigma'), \text{ i.e., } \sigma' \ | \sigma'\mu' |_{\mathcal{FD}} = | \sigma'\mu' |_{\mathcal{FD}}$$

  To prove the claim, assume any $X \in vdom(\sigma')$. Because of Postulate 2 there are two possible cases:

  (a) $\sigma'(X)$ is an integer value n. Then:

$$X\sigma' \ | \sigma'\mu' |_{\mathcal{FD}} = n = | X\sigma'\mu' |_{\mathcal{FD}} = X \ | \sigma'\mu' |_{\mathcal{FD}}$$

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

(b) $X \in var(\Pi_F)$ and $\sigma'(X)$ is a variable $X' \in var(\Pi_F)$. Then $\sigma'(X') = X'$ because σ' is idempotent, and:

$$X\sigma' \mid \sigma'\mu' \mid_{\mathcal{FD}} = X' \mid \sigma'\mu' \mid_{\mathcal{FD}} = \mid X'\sigma'\mu' \mid_{\mathcal{FD}} =$$

$$\mid X'\mu' \mid_{\mathcal{FD}} = \mid X\sigma'\mu' \mid_{\mathcal{FD}} = X \mid \sigma'\mu' \mid_{\mathcal{FD}}$$

We continue our reasoning using (8).

— Proof of $\mu' \in Sol_{\mathscr{C}}(\Pi_F)$: From (3) and the Substitution Lemma 3 we get $\sigma'\mu' \in Sol_{\mathscr{C}}(\Pi')$. Because of Postulate 2 we can assume that all the constraints belonging to Π' are $\mathcal{FD}$-specific. Then, item (4) of Theorem 2 can be applied to conclude $\mid \sigma'\mu' \mid_{\mathcal{FD}} \in Sol_{\mathcal{FD}}(\Pi')$. Using (8) we get $\mid \sigma'\mu' \mid_{\mathcal{FD}} \in Sol_{\mathcal{FD}}(\Pi' \Box \sigma')$, which trivially implies $\mid \sigma'\mu' \mid_{\mathcal{FD}} \in Sol_{\mathcal{FD}}(\exists \overline{Y'}(\Pi' \Box \sigma'))$. Because of the soundness property of the $\mathcal{FD}$ solver (see Definition 6 and Postulate 2) we obtain $\mid \sigma'\mu' \mid_{\mathcal{FD}} \in Sol_{\mathcal{FD}}(\Pi_F)$. Applying again item (4) of Theorem 2, we get $\sigma'\mu' \in Sol_{\mathscr{C}}(\Pi_F)$. Since $\Pi_F \Box \sigma_F$ is a store, $\Pi_F = \Pi_F\sigma_F$ and therefore $\sigma'\mu' \in Sol_{\mathscr{C}}(\Pi_F\sigma_F)$. Then, the Substitution Lemma 3 yields $\sigma_F\sigma'\mu' \in Sol_{\mathscr{C}}(\Pi_F)$, which is the same as $\mu' \in Sol_{\mathscr{C}}(\Pi_F)$ because of (5).

— Proof of $\mu' \in Sol(\sigma_F)$: $\mu' = \sigma_F\mu'$ follows from the following chain of equalities, which relies on (5) and the idempotency of σ_F:

$$\mu' = \sigma_F\sigma'\mu' = \sigma_F\sigma_F\sigma'\mu' = \sigma_F\mu'$$

(2) **Limited Local Completeness:** At this point we assume that rule **FS** can be applied to G in a safe way, i.e., that the solver invocation $solve^{\mathcal{FD}}(\Pi_F)$ satisfies the completeness property for solvers stated in Definition 6 (see Subsection 2.4.1). Assume $\mu \in WTSol_{\mathscr{P}}(G)$. Then there is some $\mu' =_{\setminus \overline{U}} \mu$ such that $\mu' \in WTSol_{\mathscr{P}}(P \Box C \Box M \Box H \Box F \Box R)$. Consequently, we can assume:

(9) $(P \Box C)\mu'$ is well-typed and $\mathscr{M} : \mathscr{P} \vdash_{CRWL(\mathscr{C})} (P \Box C)\mu'$ for some witness $\mathscr{M}$

$$(10)\ \mu' \in WTSol_{\mathscr{C}}(M) \quad (11)\ \mu' \in WTSol_{\mathscr{C}}(H)$$

$$(12)\ \mu' \in WTSol_{\mathscr{C}}(F) \quad (13)\ \mu' \in WTSol_{\mathscr{C}}(R)$$

In particular, (12) implies $\mu' \in WTSol_{\mathscr{C}}(\Pi_F)$. Thanks to Postulate 2 we can assume that Π_F is $\mathcal{FD}$-specific and apply item 4 of Theorem 2 to conclude $\mid \mu' \mid_{\mathcal{FD}} \in WTSol_{\mathcal{FD}}(\Pi_F)$. By completeness of the solver invocation $solve^{\mathcal{FD}}(\Pi_F)$ there is some alternative $\exists \overline{Y'}(\Pi' \Box \sigma')$ computed by the solver (i.e., such that $\Pi_F \Vdash_{solve^{\mathcal{FD}}} \exists \overline{Y'}(\Pi' \Box \sigma')$) verifying

$$(14)\quad \mid \mu' \mid_{\mathcal{FD}} \in WTSol_{\mathcal{FD}}(\exists \overline{Y'}(\Pi' \Box \sigma'))$$

Then $G' = \exists \overline{Y'}, \overline{U}.(P \Box C \Box M \Box H \Box (\Pi' \Box \sigma_F) \Box R)@_{\mathcal{FD}}\sigma'$ is one of the the finitely many goals G'_j such that $G \Vdash_{\mathbf{FS},\gamma,\mathscr{P}} G'_j$. In the rest of the proof we will show that $\mu \in WTSol_{\mathscr{P}}(G')$ by finding $\mu'' =_{\setminus \overline{Y'},\overline{U}} \mu$ such that

$$(\dagger)\ \mu'' \in Sol_{\mathscr{P}}((P \Box C \Box M \Box H \Box (\Pi' \Box \sigma_F) \Box R)@_{\mathcal{FD}}\sigma')$$

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Because of (14) there is $\hat{\hat{\mu}} \in Val_{\mathcal{FD}}$ such that

$$(15) \ \mid \mu' \mid_{\mathcal{FD}} =_{\backslash \overline{Y'}} \hat{\hat{\mu}} \in WTSol_{\mathcal{FD}}(\Pi' \Box \sigma')$$

Let $\mu'' \in Val_{\mathcal{C}}$ be univocally defined by the conditions $\mu'' =_{\overline{Y'}} \hat{\hat{\mu}}$ and $\mu'' =_{\backslash \overline{Y'}} \mu'$. Since $\mu =_{\backslash \overline{U}} \mu'$, it follows that $\mu'' =_{\backslash \overline{Y'}, \overline{U}} \mu$. Moreover, $\mid \mu'' \mid_{\mathcal{FD}} = \hat{\hat{\mu}}$, because for any variable $X \in \mathcal{V}ar$ there are two possible cases: either $X \in \overline{Y'}$ and then $\mid \mu'' \mid_{\mathcal{FD}} (X) = \mid \hat{\hat{\mu}} \mid_{\mathcal{FD}} (X) = \hat{\hat{\mu}}(X)$, since $\hat{\hat{\mu}} \in Val_{\mathcal{FD}}$; or else $X \notin \overline{Y'}$ and then $\mid \mu'' \mid_{\mathcal{FD}} (X) = \mid \mu' \mid_{\mathcal{FD}} (X) = \hat{\hat{\mu}}(X)$, since $\mid \mu' \mid_{\mathcal{FD}} =_{\backslash \overline{Y'}} \hat{\hat{\mu}}$. From (15) and $\mid \mu'' \mid_{\mathcal{FD}} = \hat{\hat{\mu}}$ we obtain $\mu'' \in WTSol_{\mathcal{FD}}(\Pi')$ by applying item 4 of Theorem 2. We now claim:

$$(16)\, \mu'' \in WTSol_{\mathcal{FD}}(\Pi' \Box \sigma')$$

To justify this claim, it is sufficient to prove $\mu'' \in Sol(\sigma')$, i.e., $\sigma'\mu'' = \mu''$. In order to prove this let us assume any $X \in vdom(\sigma')$. Because of Postulate 2, there are two possible cases:

(a) $\sigma'(X)$ is an integer value n. From (15) we know $\hat{\hat{\mu}} \in Sol(\sigma')$ and therefore $\hat{\hat{\mu}}(X) = n$. Since $\mid \mu'' \mid_{\mathcal{FD}} = \hat{\hat{\mu}}$, it follows that $\mu''(X) = n$, and then $X\sigma'\mu'' = n = X\mu''$.

(b) $X \in var(\Pi_F)$ and $\sigma'(X)$ is a variable $X' \in var(\Pi_F)$. Then:

$$\begin{array}{lr}
X\sigma'\mu'' = X'\mu'' & \\
= X'\mu' & (\text{using } \mu'' =_{\backslash \overline{Y'}} \mu' \text{ and } X' \notin \overline{Y'}) \\
= \mid X'\mu' \mid_{\mathcal{FD}} & (\text{using the fact that } \Pi_F \text{ is } \mathcal{FD}\text{-specific and (12)}) \\
= X' \mid \mu' \mid_{\mathcal{FD}} & \\
= X'\hat{\hat{\mu}} & (\text{using (15) and } X' \notin \overline{Y'}) \\
= X\sigma'\hat{\hat{\mu}} = X\hat{\hat{\mu}} & (\text{using (15)}) \\
= X \mid \mu' \mid_{\mathcal{FD}} & (\text{using (15) and } X \notin \overline{Y'}) \\
= \mid X\mu' \mid_{\mathcal{FD}} = X\mu' & (\text{using the fact that } \Pi_F \text{ is } \mathcal{FD}\text{-specific and (12)}) \\
= X\mu'' & (\text{using } \mu'' =_{\backslash \overline{Y'}} \mu' \text{ and } X \notin \overline{Y'})
\end{array}$$

We are now in a position to prove (†), thereby finishing the proof:

- Proof of $\mu'' \in WTSol_{\mathcal{P}}(P \Box C)\sigma'$: Because of the Substitution Lemma 3, this is equivalent to $\sigma'\mu'' \in WTSol_{\mathcal{P}}(P \Box C)$. Because of (16), $\sigma'\mu'' = \mu''$. Since $\mu' =_{\backslash \overline{Y'}} \mu''$ and the variables $\overline{Y'}$ do not occur in $P \Box C$, $\mu'' \in WTSol_{\mathcal{P}}(P \Box C)$ is equivalent to $\mu' \in WTSol_{\mathcal{P}}(P \Box C)$, which is ensured by the same witness $\mathcal{M}$ given by (9).
- Proof of $\mu'' \in WTSol_{\mathcal{C}}(S \star \sigma')$, S being any of the stores M, H, R: According to the choice of S we can use (10), (11), or (13) to conclude

$$(17)\, \mu' \in WTSol_{\mathcal{C}}(\Pi_S) \ \text{ and } \ (18)\, \mu' \in Sol(\sigma_S), \text{ i.e., } \sigma_S\mu' = \mu'$$

— Proof of $\mu'' \in WTSol_{\mathcal{C}}(\Pi_S\sigma')$: Since $\mu'' =_{\backslash \overline{Y'}} \mu'$ and the variables $\overline{Y'}$ do not occur in Π_S, (17) implies $\mu'' \in WTSol_{\mathcal{C}}(\Pi_S)$, which is equivalent to $\sigma'\mu'' \in WTSol_{\mathcal{C}}(\Pi_S)$ because of (16). Then, $\mu'' \in WTSol_{\mathcal{C}}(\Pi_S\sigma')$ follows from the Substitution Lemma 3.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

— Proof of $\mu'' \in WTSol(\sigma_S \star \sigma')$: Assume any variable $X \in vdom(\sigma_S)$. Then

$$X\sigma_S\sigma'\mu'' = X\sigma_S\mu'' = X\sigma_S\mu' = X\mu' = X\mu''$$

where the first equality holds because of (16), the second equality holds because $\mu'' =_{\backslash \overline{Y'}} \mu'$ and the variables $\overline{Y'}$ do not occur in $X\sigma_S$, the third equality holds because of (18), and the fourth equality holds because $\mu'' =_{\backslash \overline{Y'}} \mu'$ and the variables $\overline{Y'}$ do not include X.

- Proof of $\mu'' \in WTSol_{\mathcal{C}}(\Pi'\sigma' \,\Box\, \sigma_F\sigma')$:
  - — Proof of $\mu'' \in WTSol_{\mathcal{C}}(\Pi'\sigma')$: This is a trivial consequence of (16), since $\Pi'\sigma' = \Pi'$ (because $\Pi' \,\Box\, \sigma'$ is a store).
  - — Proof of $\mu'' \in Sol(\sigma_F\sigma')$: Because of (16) we can assume that $\mu'' \in Sol(\sigma')$, i.e., $\sigma'\mu'' = \mu''$. We must prove $\sigma_F\sigma'\mu'' = \mu''$. Assume any variable $X \in vdom(\sigma_F\sigma')$. Because of the invariant properties of admissible goals, there are three possible cases:

    (a) $X \in vdom(\sigma_F)$ and $\sigma_F(X)$ is an integer value n. Because of (12), we know that $\mu' \in Sol(\sigma_F)$ and hence $X\sigma_F\mu' = n = X\mu'$. Moreover, $X\mu'' = X\mu' = n$ because $\mu'' =_{\backslash \overline{Y'}} \mu'$ and the variables $\overline{Y'}$ do not include X. Then we can conclude that $X\sigma_F\sigma'\mu'' = n = X\mu''$.

    (b) $X \in vdom(\sigma_F)$ and $\sigma_F(X) = X' \in var(\Pi_F)$. Then:

$$\begin{aligned} X\sigma_F\sigma'\mu'' &= X'\sigma'\mu'' \\ &= X'\mu'' && \text{(using (16))} \\ &= X'\mu' && \text{(using } \mu'' =_{\backslash \overline{Y'}} \mu' \text{ and } X' \notin \overline{Y'}) \\ &= X\sigma_F\mu' = X\mu' && \text{(using (12))} \\ &= X\mu'' && \text{(using } \mu'' =_{\backslash \overline{Y'}} \mu' \text{ and } X \notin \overline{Y'}) \end{aligned}$$

    (c) $X \notin vdom(\sigma_F)$. Then $X\sigma_F = X$, and we can use $\mu'' \in Sol(\sigma')$ to deduce that $X\sigma_F\sigma'\mu'' = X\sigma'\mu'' = X\mu''$.

Rule **SF, Solving Failure**. The selected part γ is one of the four stores of the goal, the number k of possible transformations $G \Vdash_{\mathbf{RL},\gamma,\mathcal{P}} G'_j$ of G into a nonfailed goal G'_j is 0, and therefore $\bigcup_{j=1}^{k} WTSol_{\mathcal{P}}(G'_j) = \emptyset$.

(1) **Local Soundness:** The inclusion $Sol_{\mathcal{P}}(G) \supseteq \emptyset$ holds trivially.
(2) **Limited Local Completeness:** The inclusion $WTSol_{\mathcal{P}}(G) \subseteq \emptyset$ is equivalent to $WTSol_{\mathcal{P}}(G) = \emptyset$. In order to prove this, we assume that the application of **SF** to G has relied on a complete invocation of the $\mathcal{D}$ solver. Since the invocation of the solver has failed (i.e., $\Pi_S \Vdash_{solve^{\mathcal{D}}_{\mathcal{X}}} \blacksquare$) but it is assumed to be complete, we know that $WTSol_{\mathcal{D}}(\Pi_S) = \emptyset$. From this we can conclude $WTSol_{\mathcal{C}}(\Pi_S) = \emptyset$, using item (4) of Theorem 2 in case that $\mathcal{D}$ is not $\mathcal{H}$. Finally, $WTSol_{\mathcal{P}}(G) = \emptyset$ is a trivial consequence of $WTSol_{\mathcal{C}}(\Pi_S) = \emptyset$. □

A.2.5 Proof of the progress lemma

In this Subsection we prove the Progress Lemma 6 used in Subsection 3.6 to obtain the Global Completeness Theorem 6. First, we define a *well-founded progress ordering*

$\rhd$ between pairs $(G, \mathcal{M})$ formed by an admissible goal G without free occurrences of higher-order variables and a witness $\mathcal{M} = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ for the fact that $\mu \in Sol_{\mathcal{P}}(G)$. Given such a pair, we define a 7-tuple $||(G, \mathcal{M})|| =_{def} (O_1, O_2, O_3, O_4, O_5, O_6, O_7)$ (where O_1 is a finite multiset of natural numbers and $O_2, \ldots, O_7$ are natural numbers) as follows:

O$_1$ is the *restricted size of the witness* $\mathcal{M}$, defined as the multiset of natural numbers $\{| \mid \mathcal{T}_1 \mid, \ldots, \mid \mathcal{T}_n \mid |\}$, where $\mid \mathcal{T}_i \mid$ $(1 \leqslant i \leqslant n)$ denotes the *restricted size* of the $CRWL(\mathcal{C})$ proof tree $\mathcal{T}_i$ as defined in López-Fraguas *et al.* (2007), namely as the number of nodes in $\mathcal{T}_i$ corresponding to $CRWL(\mathcal{C})$ inference steps that depend on the meaning of primitive functions p (as interpreted in the coordination domain $\mathcal{C}$) plus the number of nodes in $\mathcal{T}_i$ corresponding to $CRWL(\mathcal{C})$ inference steps that depend on the meaning of user-defined functions f (according to the current program $\mathcal{P}$).

O$_2$ is the sum of $||p\,\overline{e}_n||$ for all the total applications $p\,\overline{e}_n$ of primitive functions $p \in PF^n$ occurring in the parts P and C of G, where $||p\,\overline{e}_n||$ is defined as the number of argument expressions $e_i\,(1 \leqslant i \leqslant n)$ that are not patterns.

O$_3$ is the number of occurrences of rigid and passive expressions $h\,\overline{e}_n$ that are not patterns in the productions P of G.

O$_4$ is the sum of the syntactic sizes of the right hand sides of all the productions occurring in P.

O$_5$ is the sum $sf_M + sf_H + sf_F + sf_R$ of the *solvability flags* of the four constraint stores occurring in G. The solvability flag sf_M takes the value 1 if rule **MS** from Table 8 can be applied to G, and 0 otherwise. The other three flags are defined analogously.

O$_6$ is the number of bridges occurring in the mediatorial store M of G.

O$_7$ is the number of antibridges occurring in the mediatorial store M of G.

Let $>_{lex}$ be the lexicographic product of the 7 orderings $>_i$ $(1 \leqslant i \leqslant 7)$, where $>_1$ is the multiset ordering $>_{mul}$ over multisets of natural numbers, and $>_i$ is the ordinary ordering $>$ over natural numbers for $2 \leqslant i \leqslant 7$. Finally, let us define the progress ordering $\rhd$ by the condition $(G, \mathcal{M}) \rhd (G', \mathcal{M}')$ iff $||(G, \mathcal{M})|| >_{lex} ||(G', \mathcal{M}')||$. As proved in Baader and Nipkow (1998), $>_{mul}$ is a well-founded ordering and the lexicographic product of well-founded orderings is again a well-founded ordering. Therefore, $\rhd$ is well-founded.

Now we can prove the Progress Lemma 6.

Proof of Lemma 6

Consider an admissible goal $G \equiv \exists \overline{U}.\ P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R$ for a program $\mathcal{P}$, a well-typed solution $\mu \in WTSol_{\mathcal{P}}(G)$ and a witness $\mathcal{M}$ for the fact that $\mu \in Sol_{\mathcal{P}}(G)$. Assume that neither $\mathcal{P}$ nor G have free occurrences of higher-order variables, and that G is not in solved form.

(1) Let us prove that there must be some rule **RL** applicable to G which is not a failure rule. Since G is not in solved form, we know that either $P \neq \emptyset$, or else $C \neq \emptyset$, or else some of the transformations displayed in Tables 7 and 8 can be applied to G. Note that **CF** cannot be applied to G because G has got

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

solutions. Moreover, if the failing rule **SF** would be applicable to G, then some of the other rules in Table 8 would be applicable also. Let $\mathcal{PR}$ be the set of those transformation rules displayed in Table 3 which are different of **CF**, **EL,** and **FC**. In the following items, we analyze different cases according to the from of G. In each case we either find some rule **RL** that can be applied to G or we make some assumption that can be used to reason in the subsequent cases. In the last item we conclude that rule **EL** can be applied, if no previous item has allowed to prove the applicability of another rule.

(a) If some of the transformation rules in Tables 7 and 8 can be applied to G, then we are ready. In the following items, we assume that this is not the case.

(b) If $P \neq \emptyset$ and some rule $\textbf{RL} \in \mathcal{PR}$ can be applied to G, then we are ready. In the following items we assume that this is not the case.

(c) Because of the hypothesis that G has no free occurrences of higher-order variables, from this point on we can assume that each production occurring in P must have one of the three following forms:

  i $h\,\overline{e}_m \to X$, with $h\,\overline{e}_m$ passive but not a pattern.

  ii $f\,\overline{e}_n\,\overline{a}_k \to X$, with $f \in DF^n$ and $k \geqslant 0$.

  iii $p\,\overline{e}_n \to X$, with $p \in PF^n$.

  If this were not the case, then P would include some production $e \to t$ of some other form, and a simple case analysis of the syntactic form of $e \to t$ would lead to the conclusion that some rule $\textbf{RL} \in \mathcal{PR}$ could be applied to it.

(d) If $C \neq \emptyset$ and includes some atomic constraint α that is not primitive, then the rule **FC** from Table 3 can be applied to α, and we are ready. In the following items we assume that this is not the case.

(e) If $C \neq \emptyset$ and only includes primitive atomic constraints π, then at least rule rule **SC** from Table 4 (and maybe also rules **SB** and **PP**) can be applied to G taking π as the selected part, and we are ready. In the following items, we assume that $C = \emptyset$.

(f) At this point, if there would be some variable $X \in pvar(P) \cap odvar(G)$, this X would be the right-hand side of some production in P with one of the three forms i, or ii or iii displayed in item (c) above, and one of the three rules **IM** or **DF** or **PC** could be applied, which contradicts the assumptions made at item (b). From this point on, we can assume that $pvar(P) \cap odvar(G) = \emptyset$.

(g) Let $S = \Pi_S \,\square\, \sigma_S$ be any of the four stores, let $\mathcal{D}$ be the corresponding domain, and let $\chi = pvar(P) \cap var(\Pi_S)$. Because of the assumptions made at item (a), S must be in χ-solved form and the discrimination property of the solver $solve^{\mathcal{D}}$ ensures that one of the two following conditions must hold:

  i $\chi \cap odvar_{\mathcal{D}}(\Pi_S) \neq \emptyset$, i.e., $pvar(P) \cap var(\Pi_S) \cap odvar_{\mathcal{D}}(\Pi_S) \neq \emptyset$.

  ii $\chi \cap var_{\mathcal{D}}(\Pi_S) = \emptyset$, i.e., $pvar(P) \cap var(\Pi_S) = \emptyset$.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Since i contradicts the assumption $pvar(P) \cap odvar(G) = \emptyset$ made at item (f), ii must hold for the four stores. On the other hand, the invariant properties of admissible goals guarantee that produced variables cannot occur in the answer substitutions σ_S.

(h) At this point, because of the assumptions made at the previous items, we can assume that $C = \emptyset$, the four stores are in solved form and include no produced variables, and all the productions occurring in P have the form $e \to X$, where X is a variable. Since G is not solved, it must be the case that $P \neq \emptyset$.

Note that $pvar(P)$ is finite and not empty. Moreover, the transitive closure $\gg_P^+$ of the production relation $\gg_P$ between produced variables must be irreflexive due to the invariant properties of admissible goals. Therefore, there is some production $(e \to X) \in P$ such that X is minimal w.r.t. $\gg_P$.

The variable X cannot occur in e because this would imply $X \gg_P X$, contradicting the irreflexivity of $\gg_P^+$. For any other production $(e' \to X') \in P$, X must be different of X' because of the invariant properties of admissible goals, and X cannot occur in e' because this would imply $X \gg_P X'$, contradicting the minimality of X w.r.t. $\gg_P$. Moreover, X cannot occur in the stores because they include no produced variables.

Therefore, X does not occur in the rest of the goal, and the rule **EL** can be applied to eliminate $e \to X$.

(2) Assume now any choice of a rule **RL** (not a failure rule) and a part γ of G, such that **RL** can be applied to γ in a safe manner, i.e., involving neither an opaque application of **DC** nor an incomplete solver invocation. We must prove the existence of a finite computation $G \Vdash^+_{\mathbf{RL},\gamma,\mathscr{P}} G'$ and a witness $\mathscr{M}' : \mu \in WTSol_{\mathscr{P}}(G')$ such that $(G, \mathscr{M}) \rhd (G', \mathscr{M}')$. Due to the Limited Local Completeness of $CCLNC(\mathscr{C})$ (Theorem 4, item (2)), there is one step $G \Vdash_{\mathbf{RL},\gamma,\mathscr{P}} G'_1$ such that $\mathscr{M}' : \mu \in WTSol_{\mathscr{P}}(G')$ with a witness $\mathscr{M}'$ constructed as we have sketched in the proof of Theorem 4. We define the desired finite computation by distinction of cases as follows:

(a) If **RL** is different from the two rules **SB** and **PP**, then the finite computation is chosen as $G \Vdash_{\mathbf{RL},\gamma,\mathscr{P}} G'_1$ and G' is G'_1.
(b) If **RL** is **SB** and **PP** is applicable to γ, then the finite computation is chosen as $G \Vdash_{\mathbf{SB},\gamma,\mathscr{P}} G'_1 \Vdash_{\mathbf{PP},\gamma,\mathscr{P}} G'_2 \Vdash_{\mathbf{SC},\gamma,\mathscr{P}} G'_3$ and G' is G_3.
(c) If **RL** is **SB** and **PP** is not applicable to γ, then the finite computation is chosen as $G \Vdash_{\mathbf{SB},\gamma,\mathscr{P}} G'_1 \Vdash_{\mathbf{SC},\gamma,\mathscr{P}} G'_2$ and G' is G_2.
(d) If **RL** is **PP** and **SB** is applicable to γ, then the finite computation is chosen as $G \Vdash_{\mathbf{PP},\gamma,\mathscr{P}} G'_1 \Vdash_{\mathbf{SB},\gamma,\mathscr{P}} G'_2 \Vdash_{\mathbf{SC},\gamma,\mathscr{P}} G'_3$ and G' is G_3.
(e) If **RL** is **PP** and **SB** is not applicable to γ, then the finite computation is chosen as $G \Vdash_{\mathbf{PP},\gamma,\mathscr{P}} G'_1 \Vdash_{\mathbf{SC},\gamma,\mathscr{P}} G'_2$ and G' is G_2.

Note that cases (b), (c), (d), and (e) above refer to the rules in Table 4. In all these cases, the Limited Local Completeness of $CCLNC(\mathscr{C})$ allows to find all the computation steps and a witness $\mathscr{M}' : \mu \in WTSol_{\mathscr{P}}(G')$. In all the cases, we claim that $(G, \mathscr{M}) \rhd (G', \mathscr{M}')$, i.e., $||(G, \mathscr{M})|| >_{lex} ||(G', \mathscr{M}')||$. This can be

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Table A 2. *Well-founded progress ordering $\rhd$ for $CCLNC(\mathscr{C})$*

Rules	O_1	O_2	O_3	O_4	O_5	O_6	O_7
DC	$\geqslant_{mul}$	$\geqslant$	$\geqslant$	$>$			
SP	$\geqslant_{mul}$	$\geqslant$	$\geqslant$	$>$			
IM	$\geqslant_{mul}$	$\geqslant$	$>$				
EL	$\geqslant_{mul}$	$\geqslant$	$\geqslant$	$>$			
DF	$>_{mul}$						
PC	$\geqslant_{mul}$	$\geqslant$	$\geqslant$	$>$			
FC	$\geqslant_{mul}$	$>$					
(b),(c),(d),(e)	$>_{mul}$						
IE	$\geqslant_{mul}$	$\geqslant$	$\geqslant$	$\geqslant$	$\geqslant$	$>$	
ID	$\geqslant_{mul}$	$\geqslant$	$\geqslant$	$\geqslant$	$\geqslant$	$\geqslant$	$>$
MS	$\geqslant_{mul}$	$\geqslant$	$\geqslant$	$\geqslant$	$>$		
HS	$\geqslant_{mul}$	$\geqslant$	$\geqslant$	$\geqslant$	$>$		
FS	$\geqslant_{mul}$	$\geqslant$	$\geqslant$	$\geqslant$	$>$		
RS	$\geqslant_{mul}$	$\geqslant$	$\geqslant$	$\geqslant$	$>$		

justified by Table A 2. Each file of this table corresponds to a possibility for the rule **RL** used in a one-step finite computation $G \Vdash^{+}_{\mathbf{RL},\gamma,\mathscr{P}} G'$ of type (a), except for one file which corresponds to a finite computation $G \Vdash^{+}_{\mathbf{RL},\gamma,\mathscr{P}} G'$ of type (b), (c), (d), or (e). Each column $1 \leqslant i \leqslant 7$ shows the variation in O_i according to $>_i$ when going from $||(G,\mathscr{M})||$ to $||(G',\mathscr{M}')||$ by means of the corresponding finite computation. For instance, the file for **IE** shows that the application of this rule does not increase O_i for $1 \leqslant i \leqslant 5$ and decreases O_6.

It only remains to show that the information displayed in Table A 2 is correct. Here we limit ourselves to explain the key ideas. A more precise proof could be presented on the basis of a more detailed construction of the witnesses $\mathscr{M}' : \mu \in WTSol_{\mathscr{P}}(G')$.

- For every rule **RL**, the application of **RL** does not increase O_1, as shown by the first column of the table. This happens because the witness $\mathscr{M}'$ can be constructed from $\mathscr{M}$ in such a way that all the inference steps within $\mathscr{M}'$ dealing with primitive and defined functions are borrowed from $\mathscr{M}$.
- The application of any of the rule **DF** strictly decreases O_1, as seen in the table. The reason is that the witness $\mathscr{M}$ includes a $CRWL(\mathscr{C})$ proof

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

tree $\mathscr{T}$ for an appropriate instance of a production of the form $f\,\overline{e}_n \rightarrow t$. The root inference of this proof tree contributes to the restricted size of $\mathscr{M}$ and disappears in the witness $\mathscr{M}'$ constructed from $\mathscr{M}$ as sketched in Subsection A.2.1. Therefore, the restricted size of $\mathscr{M}'$ decreases by one w.r.t. the restricted size of $\mathscr{M}$.

- The table also shows that finite computations of type (b), (c), (d), or (e) strictly decrease O_1. The reason is that such finite computations always work with a fixed primitive atomic constraint π which is ultimately moved from the constraint pool C of G to one of the stores in G' when performing the last **SC** computation step. The witness $\mathscr{M} : \mu \in WTSol_{\mathscr{P}}(G)$ includes a $CRWL(\mathscr{C})$ proof tree for an appropriate instance of π, while no corresponding proof tree is needed in the witness $\mathscr{M}'$. Therefore, the restricted size of $\mathscr{M}'$ decreases by some positive amount.
- The application of rule **FC** decrements O_2, because G includes a production $p\,\overline{e}_n \rightarrow t$ with $||p\,\overline{e}_n|| > 0$, which is replaced in G' by a primitive atomic constraint $p\,\overline{t}_n \rightarrow !\,t$ with $||p\,\overline{t}_n|| = 0$ and some new productions $e_i \rightarrow V_i$ whose contribution to the O_2 measure of G' must be smaller than $||p\,\overline{e}_n||$.
- The application of rule **IE** decreases O_6 and does not increment O_i for $1 \leqslant i \leqslant 5$. This is because in this case the witness $\mathscr{M}'$ can be chosen as $\mathscr{M}$ itself, the measures O_2, O_3, O_4, and O_5 are obviously not affected by **IE**, and the measure O_6 obviously decreases by 1 when **IE** is applied.
- Because of similar reasons, the application of rule **ID** decreases O_7 and does not increment O_i for $1 \leqslant i \leqslant 6$.
- Let **RL** be any of the four constraint solving transformations **MS**, **HS**, **FS,** and **RS**. The witness $\mathscr{M}' : \mu \in WTSol_{\mathscr{P}}(G')$ can be guaranteed to exist only if the solver invocation has been a complete one. In this case, $\mathscr{M}'$ can be chosen as the same witness $\mathscr{M}$, and therefore the O_1 measure does not increase when going from G to G'. Measures O_2, O_3, and O_4 are not affected by the bindings created by the solver invocations (since they substitute patterns for variables). Measure O_5 obviously decreases, since the solvability flag sf_S for the store that has been solved descends from 1 to 0. □

References

Antoy, S., Echahed, R. and Hanus, M. 1994. A needed narrowing strategy. In *Proc. POPL'94*. ACM Press, 268–279.

Antoy, S., Echahed, R. and Hanus, M. 2000. A needed narrowing strategy. *Journal of the ACM 74,* 4, 776–822.

Antoy, S. and Hanus, M. 2000. Compiling multi-paradigm declarative programs into Prolog. In *FroCoS*. LNCS, vol. 1794. Springer, 171–185.

Arbab, F. and Monfroy, E. 1998. Using coordination for cooperative constraint solving. In *SAC*. 139–148.

Arenas, P., Estévez, S., Fernández, A., Gil, A., López-Fraguas, F., Rodríguez-Artalejo, M. and Sáenz-Pérez, F. 2007. *$\mathscr{TOY}$: A Multiparadigm Declarative Language* (Version 2.3.1). Technical Report. R. Caballero and J. Sánchez, Eds. Available at `http://toy.sourceforge.net`.

Arenas, P., Gil, A. and López-Fraguas, F. 1994. Combining lazy narrowing with disequality constraints. In *PLILP'94*. LNCS, vol. 844. Springer, 385–399.

Baader, F. and Nipkow, T. 1998. *Term Rewriting and All That*. Cambridge University Press.

Baader, F. and Schulz, K. 1995. On the combination of symbolic constraints, solution domains and constraints solvers. In *CP'95*. LNCS, vol. 976. Springer, 380–397.

Baader, F. and Schulz, K. 1996. Unification in the union of disjoint equational theories: combining decision procedures. *Journal of Symbolic Computation 21,* 2, 211–243.

Baader, F. and Schulz, K. 1998. Combination of constraint solvers for free and quasi-free structures. *TCS 192,* 1, 107–161.

Benhamou, F. 1996. Heterogeneous constraint solving. In *ALP'96*. LNCS, vol. 1139. Springer, 62–76.

Benhamou, F., Goualard, F. and Granvilliers, L. 1997. Programming with the declic language. In *2nd International Workshop on Interval Constraints*. Port-Jefferson, New York, 1–13.

Benhamou, F., Goualard, F., Granvilliers, L. and Puget, J.-F. 1999. Revising hull and box consistency. In *ICLP'99*. The MIT Press, 230–244.

Benhamou, F. and Older, W. 1997. Applying interval arithmetic to real, integer and Boolean constraints. *The Journal of Logic Programming 32,* 1, 1–24.

Beringer, H. and Backer, B. D. 1995. Combinatorial problem solving in constraint logic programming with cooperating solvers. In *Logic Programming: Formal Methods and Practical Applications*. Elsevier Science B.V., North-Holland, 245–272.

Bockmayr, A. and Kasper, T. 2000. Branch and infer: a unifying framework for integer and finite domain constraint programming. *INFORMS Journal of Computing 10,* 3, 287.

Buntine, W. L. and Bürckert, H. J. 1994. On solving equations and disequations. *Journal of the ACM 41,* 4, 591–629.

Castro, C. and Monfroy, E. 2004. Designing hybrid cooperations with a component language for solving optimisation problems. In *AIMSA 2004*. LNCS, vol. 3192. Springer, 447–458.

Castro, C., Moossen, M. and Riff, M.-C. 2004. A cooperative framework based on local search and constraint programming for solving discrete global optimisation. In *SBIA'04*. LNCS, vol. 3171. Springer, 93–102.

Chiu, C.-K. and Lee, J. H.-M. 2002. Efficient interval linear equality solving in constraint logic programming. *Reliable Computing 8,* 2, 139–174.

Colmerauer, A. 1984. Equations and inequations on finite and infinite trees. In *Proc. of the 2nd International Conference on Fifth Generation Computer Systems*, K.-L. Clark and S.-A. Tarnlund, Eds. Tokyo, 85–99.

Colmerauer, A. 1990. An introduction to PROLOG III. *Communications of the ACM (CACM) 33,* 7, 69–90.

Comon, H. 1991. Disunification: a survey. In *Computational Logic—Essays in Honor of Alan Robinson*. MIT Press, 322–359.

Comon, H. and Lescanne, P. 1989. Equational problems and disunification. *Journal of Symbolic Computation 7*, 371–425.

Damas, L. and Milner, R. 1982. Principal type-schemes for functional programs. In *POPL'82*. ACM Press, 207–212.

de la Banda, M., Jeffery, D., Marriott, K., Nethercote, N., Stuckey, P. and Holzbaur, C. 2001. Building constraint solvers with HAL. In *ICLP'01*. LNCS, vol. 2237. Springer, 90–104.

del Vado-Vírseda, R. 2003. A demand-driven narrowing calculus with overlapping definitional trees. In *PPDP'03*. ACM Press, 213–227.

del Vado-Vírseda, R. 2005. Declarative constraint programming with definitional trees. In *Proc. FroCoS'05*. LNAI, vol. 3717. Springer, 184–199.

del Vado-Vírseda, R. 2007. A higher-order demand-driven narrowing calculus with definitional trees. In *ICTAC'07*. LNCS, vol. 4711. Springer, 165–179.

Durfee, E., Lesser, V. and Corkill, D. 1989. Cooperative distributed problem solving. In *The Handbook of Artificial Intelligence*, Vol. IV, E. Feigenbaum, A. Barr and P. Cohen, Eds. Addison-Wesley, 83–147.

Estévez-Martín, S., Fernández, A., Hortalá-González, T., Rodríguez-Artalejo, M., Sáenz-Pérez, F. and del Vado-Vírseda, R. 2007a. A proposal for the cooperation of solvers in constraint functional logic programming. *ENTCS 188*, 37–51.

Estévez-Martín, S., Fernández, A. and Sáenz-Pérez, F. 2006. *Implementing TOY: A Constraint Functional Logic Programming with Solver Cooperation*. Research Report LCC ITI 06-8, Universidad de Málaga.

Estévez-Martín, S., Fernández, A. J., Hortalá-González, M. T., Rodríguez-Artalejo, M. and del Vado Vírseda, R. 2007b. A fully sound goal solving calculus for the cooperation of solvers in the *CFLP* scheme. *ENTCS 177*, 235–252.

Estévez-Martín, S., Fernández, A. J., Hortalá-González, M. T., Rodríguez-Artalejo, M. and del Vado Vírseda, R. 2008a. Cooperation of constraint domains in the toy system. In *PPDP'08*. ACM Press, 258–268.

Estévez-Martín, S., Fernández, A. J. and Sáenz-Pérez, F. 2007c. About implementing a constraint functional logic programming system with solver cooperation. In *Proc. of CICLOPS'07*. 57–71.

Estévez-Martín, S., Fernández, A. J. and Sáenz-Pérez, F. 2008b. Playing with $\mathcal{T}OY$: constraints and domain cooperation. In *Proc. ESOP'08*. LNCS, vol. 4960. Springer, 112–115.

Fernández, A. J. and Hill, P. M. 2004. An interval constraint system for lattice domains. *ACM Transactions on Programming Languages and Systems 26,* 1, 1–46.

Fernández, A. J. and Hill, P. M. 2006. An interval constraint branching scheme for lattice domains. *Journal of Universal Computer Science 12,* 11, 1466–1499.

Fernandez, M. 1992. Narrowing based procedures for equational disunification. *Applicable Algebra in Engineering, Communication and Computing 3,* 1, 1–26.

Frank, S., Hofstedt, P. and Mai, P. R. 2003a. A flexible meta-solver framework for constraint solver collaboration. In *Proc. KI 2003*. LNCS, vol. 2821. Springer, 520–534.

Frank, S., Hofstedt, P. and Mai, P. R. 2003b. Meta-s: a strategy-oriented meta-solver framework. In *Proc. FLAIRS*. AAAI Press, 177–181.

Frank, S., Hofstedt, P., Pepper, P. and Reckmann, D. 2007. Solution strategies for multi-domain constraint logic programs. In *6th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics (PSI 2006)*, I. Virbitskaite and A. Voronkov, Eds. LNCS, vol. 4378. Springer, Novosibirsk, Russia, 209–222. Revised Papers.

Frank, S., Hofstedt, P. and Reckman, D. 2005. Meta-s—combining solver cooperation and programming languages. In *Proc. W(C)LP 2005*. Ulmer Informatik-Berichte 2005-01, 159–162.

Frank, S. and Mai, P. R. 2002. *Strategies for Cooperating Constraint Solvers*. Diploma Thesis, University of Technology, Berlin.

Frühwirth, T. 1998. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming 37,* 1–3, 95–138.

Gervet, C. 1997. Interval propagation to reason about sets: definition and implementation of a practical language. *Constraints 1,* 3, 191–244.

González-Moreno, J., Hortalá-González, M., López-Fraguas, F. and Rodríguez-Artalejo, M. 1996. A rewriting logic for declarative programming. In *Proc. ESOP'96*. LNCS, vol. 1058. Springer, 156–172.

González-Moreno, J., Hortalá-González, M., López-Fraguas, F. and Rodríguez-Artalejo, M. 1999. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming 40*, 47–87.

González-Moreno, J. C., Hortalá-González, M. T. and Rodríguez-Artalejo, M. 2001. Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming 1*, 1–71.

Goualard, F. 2001. Component programming and interoperatibility in constraint solver design. In *ERCIM Workshop on Constraints*. The Computing Research Repository (CoRR).

Goualard, F., Benhamou, F. and Granvilliers, L. 1999. An extension of the WAM for hybrid interval solvers. *The Journal of Functional and Logic Programming 1*, 1–36.

Granvilliers, L. 2001. On the combination of interval constraint solvers. *Reliable Computing 7,* 6, 467–483.

Granvilliers, L., Monfroy, E. and Benhamou, F. 2001. Cooperative solvers in constraint programming: a short introduction. *ALP Newsletter 14*, 2.

Hanus, M. 2006. *Curry: An Integrated Functional Logic Language* (Version 0.8.2 of March 28, 2006). Technical Report. Available at `http://www.informatik.uni-kiel.de/~mh/curry/`.

Harjunkoski, Jain, V. and Grossmann, I. 2000. Hybrid mixed-integer/constraint logic programming strategies for solving scheduling and combinatorial optimization problems. *Computers and Chemical Engineering 24*, 337–343.

Hindley, R. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society 146*, 29–60.

Hofstedt, P. 2000a. Better communication for tighter cooperation. In *1st International Conference on Computational Logic (CL'2000)*. LNCS, vol. 1861. Springer, 342–357.

Hofstedt, P. 2000b. Cooperating constraint solvers. In *CP'2000*. LNCS, vol. 1894. Springer, 520–524.

Hofstedt, P. 2001. *Cooperation and Coordination of Constraint Solvers*. Ph.D. Thesis, Technischen Universität Dresden, Fakultät Informatik.

Hofstedt, P. and Pepper, P. 2007. Integration of declarative and constraint programming. *Theory and Practice of Logic Programming 7,* 1–2, 93–121.

Hong, H. 1994. *Confluency of Cooperative Constraint Solving*. Technical Report 94-08, Johannes Kepler University, Austria.

Hooker, J. 2000. Logic-based methods for optimization, combining optimization and constraint satisfaction. In *Discrete Mathematics and Optimization*. John Wiley & Sons, New York.

Jaffar, J. and Lassez, J. 1987. Constraint logic programming. In *Proc. POPL'87*. ACM Press, 111–119.

Jaffar, J. and Maher, M. 1994. Constraint logic programming: a survey. *Journal of Logic Programming 19–20*, 503–581.

Jaffar, J., Maher, M., Marriott, K. and Stuckey, P. 1998. Semantics of constraints logic programs. *Journal of Logic Programming 37,* 1–3, 1–46.

Jaffar, J., Michaylov, S., Stuckey, P. and Yap, R. 1992. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems 14*, 3, 339–395.

Kepser, S. and Richts, J. 1999. Optimisation techniques for combining constraint solvers. In *FroCoS'98*. 193–210.

Khedro, T. and Genesereth, M. 1994. Modeling multiagent cooperation as distributed constraint satisfaction problem solving. In *ECAI'94*, A. Cohn, Ed. John Wiley & Sons, 249–253.

Kirchner, H. and Ringeissen, C. 1992. A constraint solver in finite algebras and its combination with unification algorithms. In *Proc. of the Joint International Conference and Symposium on Logic Programming(JICSLP'92)*, K. R. Apt, Ed. MIT Press, 225–239.

Kirchner, H. and Ringeissen, C. 1994. Combining symbolic constraint solvers on algebraic domains. *Journal of Symbolic Computation 18,* 2, 113–155.

Kobayashi, N. 2003. *Study of an Open Equational Solving System*. Ph.D. Thesis, Institute of Information Sciences and Electronics, University of Tsukuba, Japan.

Kobayashi, N., Marin, M. and Ida, T. 2001. Collaborative constraint functional logic programming in an open environment. In *The Second Asian Workshop on Programming Languages and Systems (APLAS'01)*. Korea, 49–59.

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.

Kobayashi, N., Marin, M., Ida, T. and Che, Z. 2002. Open *CFLP*: an open system for collaborative constraint functional logic programming. In *11th International Workshop on Functional and (Constraint) Logic Programming (WFLP'02)*. Grado, Italy, 229–232.

Krzikalla, O. 1997. *Constraint solver für lineare constraints über reellen zahlen*. Großer Beleg, Technische Universität Dresden.

Lassez, J., Maher, M. and Marriot, K. 1988. Unification revisited. In *Foundations of logic and functional programming*. LNCS, vol. 306. Springer, 67–113.

Loogen, R., López-Fraguas, F. and Rodríguez-Artalejo, M. 1993. A demand driven computation strategy for lazy narrowing. In *Proc. PLILP'93*. LNCS, vol. 714. Springer, 184–200.

López-Fraguas, F., Rodríguez-Artalejo, M. and del Vado-Virseda, R. 2004. A lazy narrowing calculus for declarative constraint programming. In *PPDP'04*. ACM Press, 43–54.

López-Fraguas, F., Rodríguez-Artalejo, M. and del Vado Vírseda, R. 2007. A new generic scheme for functional logic programming with constraints. *Higher-Order and Symbolic Computation 20,* 1–2, 73–122.

Maher, M. J. 1988. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *LICS'88*. IEEE Computer Society, 348–357.

Marin, M. 2000. *Functional Logic Programming with Distributed Constraint Solving*. Ph.D. Thesis, Johannes Kepler Universität Linz.

Marin, M., Ida, T. and Schreiner, W. 2001. *CFLP*: a Mathematica implementation of a distributed constraint solving system. In *IMS'99*, vol. 8. WIT Press, 287–300.

Marti, P. and Rueher, M. 1995. A distributed cooperating constraints solving system. *International Journal of Artificial Intelligence Tools 4,* 1–2, 93–113.

Milner, R. 1978. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences 17*, 348–375.

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

Monfroy, E. 1998. A solver collaboration in BALI. In *Proc. of JCSLP'98*. MIT Press, 349–350.

Monfroy, E. and Castro, C. 2004. A component language for hybrid solver cooperations. In *ADVIS'04*. LNCS, vol. 3261. Springer, 192–202.

https://doi.org/10.1017/S1471068409003780 Published online by Cambridge University Press

Monfroy, E. and Ringeissen, C. 1999. An open automated framework for constraint solver extension: the solex approach. *Fundamenta Informaticae 39,* 1–2, 167–187.

Monfroy, E., Rusinowitch, M. and Schott, R. 1995. *Implementing Non-linear Constraints with Cooperative Solvers*. Research Report 2747, CRI Nancy.

N'Dong, S. 1997. Prolog IV ou la programmation par contraintes selon PrologIA. In *JFPLC'97*. Edition HERMES, 235–238.

Nelson, G. and Oppen, D. 1979. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems 1,* 2, 245–257.

Pajot, B. and Monfroy, E. 2003. Separating search and strategy in solver cooperations. In *Proc. PSI'03*. LNCS, vol. 2890. Springer, 401–414.

Peyton-Jones, S. 2002. *Haskell 98 Language and Libraries: The Revised Report*. Technical Report. `Avilable at: http://www.haskell.org/onlinereport/`.

Rodošěk, R., Wallace, M. and Hajian, M. 1997. A new approach to integrating mixed integer programming and constraint logic programming. In *Annals of Operations Research*, vol. 86. 63–87.

Rodríguez-Artalejo, M. 2001. Functional and constraint logic programming. In *Constraints in Computational Logics: Theory and Applications*. LNCS, vol. 2002. Springer, 202–270.

Rueher, M. and Solnon, C. 1997. Concurrent cooperating solvers over reals. *Reliable Computing 3,* 3, 325–333.

Sáenz-Pérez, F. 2007. ACIDE: an integrated development environment configurable for LaTeX. *The PracTeX Journal 3*, 1–17.

Saraswat, V. 1992. The category of constraint systems is cartesian closed. In *Proc. of the 7th Annual IEEE Symposium on Logic in Computer Science*. IEEE Press, 341–345.

SICStus Prolog. 2007. `Available at http://www.sics.se/isl/sicstus`.

Thorsteinsson, E. 2001. Branch-and-check: a hybrid framework integrating mixed integer programming and constraint logic programming. In *CP 2001*, T. Walsh, Ed. LNCS, vol. 2239. Springer, 16–30.

van Hentenryck, P., Saraswat, V. and Deville, Y. 1998. Design, implementation and evaluation of the constraint language cc(FD). *Journal of Logic Programming 37*, 139–164.

van Hentenryck, P., Simonis, H. and Dincbas, M. 1994. Constraint satisfaction using constraint logic programming. *Artificial Intelligence 58*, 113–159.

van Hoeve, W. 2000. *Towards the Integration of Constraint Logic Programming and Mathematical Programming*. Master's Thesis, University of Twente, The Netherlands.

Vandecasteele, H. and Rodošěk. 1998. Modelling combinatorial problems for CLP(FD+R). In *Proc. of the Benelux Workshop on Logic Programming*. 1–9.

Zhou, J. 2000. Introduction to the constraint language NCL. *The Journal of Logic Programming 45,* 1-3, 71–103.

Zoeteweij, P. 2003. Coordination-based distributed constraint solving in DICE. In *ACM Symposium on Applied Computing (SAC'03)*. ACM, Melbourne, FL, 360–366.