INCLP($\mathbb{R}$)

Interval-based Nonlinear Constraint Logic Programming over the Reals

Leslie De Koninck*, Tom Schrijvers**, Bart Demoen

Department of Computer Science, K.U.Leuven, Belgium
{leslie,toms,bmd}@cs.kuleuven.be

Abstract We present the INCLP($\mathbb{R}$) system, a new Constraint Logic Programming system for nonlinear constraints over the reals, based on interval techniques. A first evaluation shows that we can improve on other systems in a number of areas. INCLP($\mathbb{R}$) is written in Prolog using Constraint Handling Rules and is the first nonlinear CLP system implemented using this technology. Directions for future research are given.

1 Introduction

In this paper, we present a new Constraint Logic Programming system, INCLP($\mathbb{R}$), that solves sets of nonlinear constraints over the real numbers. An example of such a constraint set is Example 1.

Example 1. The constraint set $x^2 = y$, $x = 2 \cdot y$ has two solutions: $x = 0$, $y = 0$ and $x = 0.5$, $y = 0.25$. Due to the term x^2, the constraint set is nonlinear.

Our system is designed to be a general purpose, robust and efficient CLP system. The decision to design a *new* system is mainly motivated by three reasons. The first two are based on our experience with the CLP($\mathbb{R}$) system [14] which can be found in SICStus Prolog [22] and which we ported recently to SWI-Prolog [27]. The third motivation concerns the limited availability and lack of integration in Prolog of nonlinear systems. We now describe our motivation in more detail:

1. The CLP($\mathbb{R}$) system is based on floating point arithmetic which leads to rounding errors. There are many techniques in numerical analysis that minimize these errors by reordering operations, which is incompatible with the incremental nature of CLP. A good alternative is the use of interval arithmetic which offers explicit error bounds. Because of this and the many other advantages that interval arithmetic offers, our system operates on real intervals.
2. Many problems in application areas as various as chemical engineering [8], computer graphics [19] and economics [21], require some nonlinear constraints to be satisfied. These problems cannot be solved by a system like CLP($\mathbb{R}$) that only solves linear constraints. For example, CLP($\mathbb{R}$) cannot solve Example 1. This limitation does not fit in our view of a general purpose CLP system, so it is our aim to support nonlinear constraints as well.
3. Several interval-based nonlinear constraint solvers exist today. Typically, these are commercial systems. On the one hand, there are non-CLP systems (i.e. not based on Logic Programming) like Newton [11], Helios [18] and Numerica [10] as well as ILOG Solver [23]. On the other hand, there are extended Prolog implementations like ECL^{i}PS^e [15] and Prolog IV [20] whose constraint solver cannot be easily ported to other Prolog systems.

Overview We start in Section 2 with definitions and notation that are used in this paper. The context of the system is sketched in Section 3 where we also give an illustration. In Section 4 we present an overview of the different components of the system. Implementation details are given in Section 5 and an evaluation follows in Section 6. We conclude in Section 7 where we describe some areas of future work and research.

* Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen) and GOA 2003/8
** Research Assistant of the Research Foundation - Flanders (F.W.O.-Vlaanderen).

2 Definitions and Notation

We assume some general knowledge of interval analysis. See for example [1]. We give a brief overview of the notation used. Our notation follows [16].[1] Intervals and interval boxes are denoted in boldface. We only consider intervals and boxes with IEEE 754 floating point bounds (including $\pm\infty$).

A real function f can be computed by using an *expression* of f: a sequence of arithmetic operations, denoted by sans serif f. We need this distinction because interval arithmetic does not satisfy certain properties of real arithmetic, like associativity or distributivity, and so different expressions of the same real function may have different evaluations when performed in interval arithmetic.

An interval extension of function f with respect to expression f is denoted by f_*. We denote the natural interval extension by f_N and the Taylor interval extension by f_T. The natural interval extension of a function f with respect to expression f is formed by replacing all real arithmetic operations and primitive functions by their interval arithmetic counterpart. All interval extensions used here are inclusion isotonic, i.e. $\mathbf{x} \subseteq \mathbf{y} \implies \mathsf{f}(\mathbf{x}) \subseteq \mathsf{f}(\mathbf{y})$. We define the *existential interval extension* of a relation $\omega \in \{=, <, >\}$ by $\mathbf{x}\omega^{\exists}\mathbf{y} \iff \exists x \in \mathbf{x}, y \in \mathbf{y} : x\omega y$.

3 Context and Illustration

The INCLP($\mathbb{R}$) system is designed to be work in many different programming environments. Therefore, portability to other Prolog implementations is essential. The system is implemented in SWI-Prolog [27] and makes use of attributed variables [5] and the Constraint Handling Rules library [7]. The system can be easily ported to other Prolog environments that have attributed variables and CHR. A port to hProlog [4] required little effort.

3.1 Advantages of Using CHR

The INCLP($\mathbb{R}$) system is implemented using Constraint Handling Rules. This rule-based language is designed to simplify the implementation of constraint solvers and can be used on top of a host language like Prolog. Here, we present the two main reasons for using CHR to write a constraint solver. They are based on our experiences with the K.U.Leuven CHR-system [26] on top of SWI-Prolog.

Implicit Constraint Store A constraint solver keeps track of constraints and other relevant data. In Prolog, one can program this either by passing around all necessary data in predicate arguments, or by explicitly using attributes of constraint variables or global variables. Explicitly programming the constraint store by using one of these techniques, distracts from the main task of modelling the problem.

CHR implicitly maintains a constraint store which allows storing and retrieving constraints in a uniform manner. The K.U.Leuven CHR system chooses an optimal implementation based on mode information and look-up patterns.

Multiple Heads Constraint propagation relies on the detection of patterns of simultaneous constraints in the constraint store. This is typically implemented by a search mechanism that uses matching. The implementation of such a mechanism is quite involved and must take into account the constraint representation. In CHR the user implicitly defines the relevant matchings by using multi-headed rules. The CHR system takes care of how to implement this matching in an efficient way.

It uses hash tables whenever that is appropriate. It also optimizes the code for the matchings that actually occur in the program. As a consequence, the constraint solver can be adapted without reprogramming the storage structures whenever a new type of matching is added or an old type disappears.

[1] To simplify terminology and notation, we speak of real intervals, actually meaning extended real intervals.

3.2 User Interface and Illustration

The user interface is similar to the one of the CLP($\mathbb{R}$) system. New constraints are entered using the `{}/1` predicate which takes a constraint or a conjunction of constraints as its argument. As an illustration, we encode Example 1:

```
?- {X**2=Y, X=2*Y}.

    {-0.0 =< Y =< 0.25}
    {-0.0 =< X =< 0.5}
```

Searching for the different solutions is done by using the `solve/0` predicate as in:

```
?- {X**2=Y,X=2*Y}, solve.

    {-0.0 =< Y =< 4.9406565e-323}
    {-0.0 =< X =< 9.8813129e-323} ;

    {0.25 =< Y =< 0.25}
    {0.5 =< X =< 0.5}
```

As the examples show, a solution does not necessarily instantiate the variables. Because of the finite nature of floating point numbers, the system can often only give a small interval in which the actual value of the variable resides.

4 System Overview

The INCLP($\mathbb{R}$) system solves constraint sets by using a combination of search and maintaining consistency. Consistency is established by repeatedly applying a domain narrowing operator which reduces the size of the variables' domains.

Interval extensions of the constraint functions are used to safely remove regions of the search space that do not satisfy the constraints. This is possible because an interval extension of a function over a given domain, contains the range of that function over that domain.

The solutions of the current constraint set can be found by using the `solve/0` predicate, which returns small, consistent regions of the search space. It operates by iteratively bisecting the variables' domains while returning to a consistent state after each bisection until the desired precision is reached.

Figure 1 shows the components of the system. Their function is described below.

Constraint Preparation This component prepares new constraints to be handled by the other components.

Unification Handling Here, the constraint store is made consistent after a unification has occurred.

Search Operator The search operator subdivides the search space to localize different solution regions of the constraint system. It is activated by using the `solve/0` predicate which bisects a domain whenever the consistency checker reaches a fixed point and the required precision has not been reached yet. In INCLP($\mathbb{R}$), the search is organized in a round robin fashion.

Scheduler This component schedules consistency checks. It makes sure that all inconsistencies are detected while minimizing redundant work.

Consistency Checker The consistency checker checks whether a certain consistency criterion is satisfied with respect to a given combination of constraint, variable and interval extension. The consistency criterion used in the INCLP($\mathbb{R}$) system is called box-consistency [3,9].

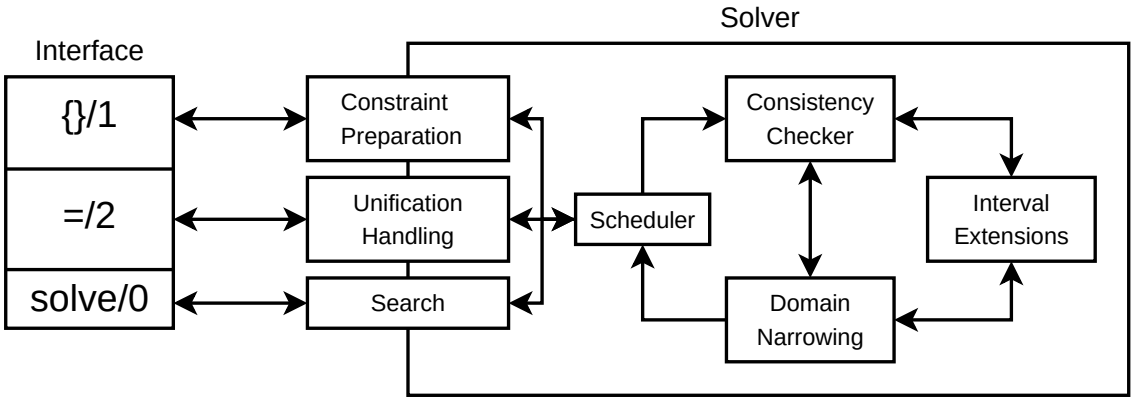

Figure1. System Design

Domain Narrowing This component reduces the domain of a variable whenever one of its bounds is found to be inconsistent by the consistency checker. It operates by using interval extensions. Whenever a domain is reduced, the changes are propagated to other domains by scheduling new consistency checks. The domain narrowing operator in the INCLP($\mathbb{R}$) system is a componentwise interval Newton operator [12].

Interval Extensions The interval extensions give an outer approximation of the range of the constraint functions over a given domain. They are used by both the consistency checker and the domain narrowing operator. The INCLP($\mathbb{R}$) system offers the natural interval extension as well as (two forms of) the Taylor interval extension.

Implementation details of some of these components are worked out in Section 5.

5 Implementation Details

This section discusses a selection of interesting implementation details of the components of the system. We present constraint preparation and interval arithmetic in Section 5.1, interval extensions in Section 5.2, unification handling in Section 5.3 and consistency checking and domain narrowing in Section 5.4.

5.1 Constraint Preparation and Interval Arithmetic

In this section, we describe two small but nonetheless important components: constraint preparation and interval arithmetic.

Constraint Preparation When new constraints are inserted into the system, some preparatory work is needed. In this phase, the variables in the constraints get an initial domain, a global ordering between the different variables is established, the base forms for the interval extensions are constructed and consistency checks are scheduled. Base forms are presented in Section 5.2.

Interval Arithmetic The interval arithmetic is implemented in pure Prolog. It is based on two components: an extension of the floating point arithmetic with the infinities $\pm\infty$ and an interval arithmetic over these extended floating point numbers.

The extension of the floating point numbers is necessary because the ISO-Prolog standard does not support $\pm\infty$. It can be removed in ports to Prolog implementations, like hProlog or SICStus Prolog, that do provide the infinities.

Our implementation of interval arithmetic is inspired by [13]. Because we cannot control rounding modes in Prolog, an explicit outward rounding is used by taking the floating point number just

before or after the result. This often leads to larger intervals than necessary, but is guaranteed to be correct.

5.2 Interval Extensions

In this section, interval extensions are presented. We first describe the two types of the Taylor interval extension that are supported in INCLP($\mathbb{R}$) and then describe the evaluation mechanism of interval extensions.

Taylor Interval Extension In one dimension, the Taylor interval extension is defined as follows:

$$\mathsf{f}_T(\mathbf{x}) = f(\hat{x}) + \mathsf{f}'_*(\mathbf{x}) \cdot (\mathbf{x} - \hat{x})$$

with $\hat{x}$ the center of $\mathbf{x}$. In practice, the natural interval extension of the derivative is used. In multiple dimensions, most often one uses the form

$$\mathsf{f}_T(\mathbf{x}) = f(\hat{x}) + \sum_{i=1}^{n} \frac{\partial}{\partial x_i} \mathsf{f}_*(\mathbf{x}_1, ..., \mathbf{x}_n) \cdot (\mathbf{x}_i - \hat{x}_i) \tag{1}$$

which is based on the Taylor series expansion into the direction of $\mathbf{x} - \hat{x}$. The INCLP($\mathbb{R}$) system offers an alternative form based on componentwise Taylor series expansions:

$$\mathsf{f}_T(\mathbf{x}) = f(\hat{x}) + \sum_{i=1}^{n} \frac{\partial}{\partial x_i} \mathsf{f}_*(\mathbf{x}_1, ..., \mathbf{x}_i, \hat{x}_{i+1}, ..., \hat{x}_n) \cdot (\mathbf{x}_i - \hat{x}_i) \tag{2}$$

The same principle is applied by Rump in [25] on slope intervals. It is clear that the form (2) is more precise than the form (1). We call (1) the standard Taylor interval extension and (2) the componentwise Taylor interval extension.

Evaluation Mechanism The evaluation of interval extensions is a performance bottleneck in our system. We use a form of partial evaluation to speed up this evaluation. We distinguish three forms of an interval expression representing a constraint:

Base form This is the expression of an interval extension with no domain information being used.

Partial evaluation The partial evaluation with respect to variable x uses domain information from all variables except for x.

Full evaluation The full evaluation is an interval and uses domain information from all variables.

The base form is the basis for the partial evaluation and only changes when unification occurs. The partial evaluation is created just before consistency checking and forms the basis for the full evaluation. It can be used as long as none of the domains involved is changed. The full evaluation is used during consistency checking and domain narrowing. We now illustrate these forms with an example.

Example 2. For the expression $\mathsf{f}(x, y) = x^2 \cdot y^2$, $x \in [-10, 10]$, $y \in [0, 5]$ and for the componentwise Taylor interval extension, we have:

Base form: $(\hat{x}^2 \cdot \hat{y}^2) + (2 \cdot \mathbf{x} \cdot \hat{y}^2) \cdot (\mathbf{x} - \hat{x}) + (2 \cdot \mathbf{x}^2 \cdot \mathbf{y}) \cdot (\mathbf{y} - \hat{y})$

Partial evaluation w.r.t. x: $(6.25 \cdot \hat{x}^2) + (12.5 \cdot \mathbf{x}) \cdot (\mathbf{x} - \hat{x}) + ([0, 10] \cdot \mathbf{x}^2) \cdot ([-2.5, 2.5])$

Partial evaluation w.r.t. y: $([-20, 20] \cdot \hat{y}^2) \cdot [-10, 10] + ([0, 200] \cdot \mathbf{y}) \cdot (\mathbf{y} - \hat{y})$

Full evaluation: $[-3750, 3750]$

When a new constraint is entered in the system, a base form for all the interval extensions is generated. We generate both base forms for the interval extensions of the constraint function and of the derivative of the constraint function. The latter is needed for our domain narrowing operator.

In the base form, we replace the original problem variables by fresh ones so we can distinguish between occurrences that have to be replaced by the variable's domain and occurrences that have to be replaced by the center of the domain. This also makes sure that unification of these variables does not reactivate other constraints.

The evaluation of the interval extensions is done by unifying the fresh variables with their respective domains and centers of domains. A partial evaluation unifies all variables but one. By using backtracking, we are able to reuse the base forms and partial evaluations.

5.3 Unification Handling

Because we are working in Prolog, we can use unification as a way to handle equality constraints. This offers extra flexibility to the user of the INCLP($\mathbb{R}$) system who can use variables occurring in constraints in a natural way. It is also important for efficiency as unification enables a stronger form of consistency. The user is allowed to directly unify two variables x_i and x_j, the result of which is denoted by $x_{i\cup j}$. Unification of a variable with a number is also supported.

After unification, some data structures become inconsistent and need updating. Also, new consistency checks need to be scheduled. Because all updates need to be done before consistency checking and because the order in which updates are made is important, we use an explicit triggering mechanism for which we rely on the refined operational semantics of CHR [6].

States We introduce a pair of singleton constraints: `state_unify/0` and `state_normal/0` which represent the operation mode of the system. In normal mode, consistency checking and related functions are performed. In unification mode all unification updates are performed, trying rules from top to bottom. The unification handling rules are activated by the `state_unify/0` constraint, all other constraints in the head are always passive.

Unification handling is based on pairs of rules: one for the case where two variables are unified and one for the case where a variable is unified with a number. These rules may schedule consistency checks that will be applied when the system returns to normal mode, after all updates are done.

The following rules illustrate how this works. It is a simplified version of how unification is handled for the natural interval extension. Typically, these kind of rules remove redundant constraints, update others, and schedule new consistency checks.

```
state_unify \ constraint_projection(ID,n,V,CP1) #P1,
    constraint_projection(ID,n,V,CP2) #P2, varlist(ID,n,Varlist) #P3 <=>
        remove_duplicates(Varlist,NewVarlist),
        merge_projections(CP1,CP2,NewCP),
        constraint_projection(ID,n,V,NewCP), % update constraint projection
        varlist(ID,n,NewVarlist),            % update list of variables
        schedule_variables(n,V)              % schedule connected variables
        pragma passive(P1), passive(P2), passive(P3).
state_unify \ constraint_projection(ID,n,N,_) #P1, varlist(ID,n,Varlist) #P2 <=>
        number(N) |
        remove_nonvariables(Varlist,NewVarlist),
        varlist(ID,n,NewVarlist), % update list of variables
        schedule_constraint(n,ID) % schedule all variables in the constraint
        pragma passive(P1), passive(P2).
```

Taylor Interval Extension Because the Taylor interval extension is quite complex as far as unification is concerned, we look at it in more detail. The Taylor interval extension represented by

equation (2) contains a sum of partial derivatives:

$$\sum_{i=1}^{n} \frac{\partial}{\partial x_i} \mathsf{f}_*(\mathbf{x}_1, ..., \mathbf{x}_i, \hat{x}_{i+1}, ..., \hat{x}_n) \cdot (\mathbf{x}_i - \hat{x}_i) \tag{3}$$

When two variables x_i and x_j are unified into $x_{i\cup j}$, (3) contains two different derivatives for the difference $(\mathbf{x}_{i\cup j} - \hat{x}_{i\cup j})$. Because of the chain rule for derivatives, the derivative for the unified variable is equal to the sum of these two derivatives.

For the componentwise Taylor interval extension, there is another issue: after unification, some partial derivatives will contain occurrences of $\mathbf{x}_{i\cup j}$ and $\hat{x}_{i\cup j}$. This is an inconsistency that has to be removed. The only affected partial derivatives are the ones 'between' the original x_i and x_j.

Finally, the partial derivatives of a function are used in the domain narrowing operator and a Taylor interval extension is made of them. Unification causes the system to have two different partial derivatives for the same function with respect to the unified variable $x_{i\cup j}$. These have to be merged by componentwise adding partial derivatives in order to return to a consistent state.

5.4 Consistency Checker and Domain Narrowing

In this section we describe the consistency checker and domain narrowing operator. They are invoked with respect to a constraint, an interval extension and a variable. We denote these by the *current* constraint, interval extension and variable.

Consistency Checker The consistency checker is based on the partial evaluation form of the current interval extension. It creates a full evaluation form by replacing the current variable by its domain bounds. Consistency is checked by using the existential evaluation of interval relations. Whenever one of the bounds is found to be inconsistent, the domain narrowing operator is invoked. The following code illustrates how consistency checks are activated. It is a simplified version of the rule for the natural interval extension.

```
active_phase(n), active_var(n,V), state_normal,
    constraint_projection(ID,n,V,CP) #P1, varlist(ID,n,Varlist) #P2 <=>
        create_partial_evaluation(n,CP,PEval),
        get_domain(V,Domain),
        check(n,PEval,V,Varlist,Domain,NewDomain),
        (   Domain \== NewDomain
        ->  set_domain(V,NewDomain), % The domain of V has changed
            schedule_variables(n,V)  % Schedule connected variables
        ;   true % The domain of V has not changed: nothing to be done
        ).
```

Domain Narrowing Operator The domain narrowing operator is invoked whenever a variable's domain has an inconsistent bound. Its task is to find a new bound for the domain that is consistent without excluding any solutions. This happens by searching for the leftmost or rightmost quasi-zero in the variable's domain. Quasi-zeros are zeros of the projection of a function on one variable, where all other variables are replaced by their domains.

Our system uses a componentwise interval Newton operator combined with a bisection mechanism. If the interval Newton operator fails, this means that there are no solutions in the given domain. If it succeeds, we either have a reduced domain, a domain with a 'gap' or no domain reduction. In the event of no domain reduction, bisection is applied.

6 Evaluation

The INCLP($\mathbb{R}$) system is optimized for constraints which contain different variables and multiple occurrences of the same variable. In this section, we make a first evaluation of the system with respect to these optimizations. The benchmarks are performed on a Pentium IV, 2.8 GHz using SWI-Prolog 5.5.37. The first two benchmarks also use ECL^{i}PS^e 5.6.

Box Consistency The INCLP($\mathbb{R}$) system uses box-consistency which is advantageous if there are multiple occurrences of the same variable in a constraint, but is very expensive compared to hull consistency if variables have only one occurrence.

We compare our box-consistency with the hull-consistency of ECL^{i}PS^e, which is much cheaper, but also much more susceptible to dependency problems. The benchmark consists of finding all the zeroes of a one-dimensional polynomial. The results are shown in the table below. The degree of the polynomial is shown in the first column. The time in seconds is given in the second column for the INCLP($\mathbb{R}$) system and in the third column for ECL^{i}PS^e. The last column shows the factor of ECL^{i}PS^e compared to INCLP($\mathbb{R}$). It gives an indication of the bad scalability of hull-consistency techniques when the number of occurrences of a variable increases.

Degree	$T_{INCLP(\mathbb{R})}$(s)	$T_{ECL^iPS^e}$(s)	$T_{ECL^iPS^e}/T_{INCLP(\mathbb{R})}$
6	0.20	1.85	9
8	0.42	6.43	15
10	0.81	24.77	31
12	2.20	157.59	72

For each degree, ten polynomials are chosen for which box-consistency is able to reach the required precision of eight significant digits. ECL^{i}PS^e is unable to reach satisfactory precision by using consistency and generates many small intervals around the solutions. INCLP($\mathbb{R}$)'s overhead of using box-consistency is largely compensated by the better pruning. This test is of course in no way a complete comparison between INCLP($\mathbb{R}$) and ECL^{i}PS^e.

Another often used benchmark is the n-dimensional Broyden Banded function, defined as $x_i \cdot (2+5 \cdot x_i^2)+1-\sum_{j \in J_i} x_j \cdot (1+x_j) = 0$ where $J_i = \{j | j \neq i \text{ and } \max(1, i-5) \leq j \leq \min(m, i+1)\}$ for $1 \leq i \leq n$. Again, we compare with ECL^{i}PS^e.

Dimension	$T_{INCLP(\mathbb{R})}$(s)	$T_{ECL^iPS^e}$(s)	$T_{ECL^iPS^e}/T_{INCLP(\mathbb{R})}$
10	2.39	13.83	6
20	5.84	138.06	24
30	9.66	423.01	44
40	13.98	740.01	53
50	18.25	1055.00	58

As before, the multiple occurrences of variables in the constraints cause INCLP($\mathbb{R}$) to outperform ECL^{i}PS^e. In this benchmark, the constraints do not become more complex when n increases, so the runtime should scale linearly with increasing n. INCLP($\mathbb{R}$) exhibits a higher time complexity, mainly due to some overhead caused by the scheduler.

Taylor Interval Extension The componentwise Taylor interval extension is more precise than the standard Taylor interval extension in the case of multidimensional constraints. We evaluate this improvement in the next benchmark. We use a set of ten constraints in ten variables, where each constraint contains four different variables. The example can be found in [12], Example 7.6.

In the table below, the first column indicates the interval extension being used. The second column indicates whether consistency preservation is done incrementally after each new constraint or once after all constraints have been entered. The third column shows the runtime spent on consistency during the insertion of the constraints and the fourth column shows the runtime of the search phase. The last column gives the total runtime.

Interval Extension	Mode	$T_{insertion}$(s)	T_{search}(s)	T_{total}(s)
Natural	incremental	3.12	0.01	3.13
	block	0.82	0.01	0.83
Standard Taylor	incremental	0.15	1.00	1.15
	block	0.09	1.01	1.10
Componentwise Taylor	incremental	3.85	0.01	3.86
	block	0.78	0.01	0.79

The two versions of the Taylor interval extension give quite different results because the componentwise Taylor interval extension is able to reach a higher level of consistency than the standard Taylor interval extension. This reduces the search time but is expensive when constraints are entered incrementally. The natural interval extension performs very well because there is no dependency problem in the example.

7 Conclusions

The work presented in this paper is a continuation of the work done in [17]. In the introduction, we presented three main problems of existing solvers: the precision problems of pure floating point based solvers, the limited expressive power of linear constraint solvers and the lack of openness and availability of (interval-based) nonlinear systems. INCLP($\mathbb{R}$) overcomes all of these problems. The design focus can now shift towards a more efficient and robust system.

We created a system supporting nonlinear constraints in multiple dimensions. Although only polynomial constraints are currently supported, the only property that we rely on at this point is that the constraint functions are smooth. We made a port to hProlog which proves the claim that INCLP($\mathbb{R}$) is highly portable. Experimental evaluation has shown that INCLP($\mathbb{R}$) already is an improvement over existing software for specific kinds of constraints. This strengthens our belief that we can build a competitive system.

INCLP($\mathbb{R}$) is written using Constraint Handling Rules and is the first nonlinear CLP system based on that technology. The entire system consists of less than 100kb of code, with about one fourth of it concerning interval arithmetic. This illustrates the power of CHR for writing complex and efficient constraint solvers. More information on INCLP($\mathbb{R}$), including a link to download the system, a short manual and the described benchmarks, is available at the author's website at http://www.cs.kuleuven.be/~leslie/INCLPR/.

7.1 Future Work

As future work we plan to improve many of the crucial components of the system. We will implement other interval extensions like a slope based interval extension [25,24] and will improve the ones that are already in the system. In particular, we will add a monotonicity test for the Taylor interval extension to create sharper enclosures. The often expensive box-consistency will be complemented with hull-consistency for variables which occur only once in a constraint [2]. We also plan to extend the functionality by allowing non-polynomial constraints.

Our second benchmark already showed the advantages of processing constraints in blocks in comparison with an incremental approach. However, incremental constraint solving is often necessary in a CLP context because of choice points. Therefore, it is important to investigate how to make incremental constraint solving more efficient. Termination criteria play a crucial role in this matter.

The integration of interval based constraint programming in the logic programming context is another point that deserves our attention. There are many semantical issues that arise in a finite precision context. Also the exact semantics of a consistent constraint set is important. When these issues are sorted out, it may become advantageous to compile programs that are written using INCLP($\mathbb{R}$) so that the most optimal implementation for a given problem can be chosen.

Finally, a more thorough evaluation is definitely needed, in particular comparing with other systems as well as evaluating different design choices. It is important to note here that we do not plan to make the difference on the level of constant factors, but rather on the more fundamental level of time complexity.

References

1. Götz Alefeld and Günter Mayer. Interval analysis: theory and applications. *Journal of Computational and Applied Mathematics*, 121:421–464, Sep 2000.

2. Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-Francois Puget. Revising hull and box consistency. In *ICLP*, pages 230–244, 1999.
3. Frédéric Benhamou, David A. McAllester, and Pascal Van Hentenryck. CLP(Intervals) revisited. In *SLP*, pages 124–138, 1994.
4. Bart Demoen. hProlog. http://www.cs.kuleuven.be/~bmd/hProlog/.
5. Bart Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Technical Report CW 350, K.U.Leuven, 2002.
6. Gregory J. Duck, Peter J. Stuckey, Maria J. García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In Bart Demoen and Vladimir Lifschitz, editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2004.
7. Thom W. Frühwirth. Constraint Handling Rules. In Andreas Podelski, editor, *Constraint Programming*, volume 910 of *Lecture Notes in Computer Science*, pages 90–107. Springer, 1994.
8. Chao-Yang Gau and Mark A. Stadtherr. Nonlinear parameter estimation using interval analysis. *AIChE Symp. Ser.*, 94(304):445–450, 1999.
9. Laurent Granvilliers, Frédéric Goualard, and Frédéric Benhamou. Box consistency through weak box consistency. In *ICTAI*, pages 373–380, 1999.
10. Pascal Van Hentenryck. Numerica: a modeling language for global optimization. In *IJCAI*, pages 1642–1650, 1997.
11. Pascal Van Hentenryck, Laurent Michel, and Frédéric Benhamou. Newton - constraint programming over nonlinear constraints. *Sci. Comput. Program.*, 30(1-2):83–118, 1998.
12. Stefan Herbort and Dietmar Ratz. Improving the efficiency of a nonlinear-system-solver using a componentwise Newton method. Technical Report 2, Universität Karlsruhe, 1997.
13. Timothy J. Hickey, Qun Ju, and Maarten H. van Emden. Interval arithmetic: from principles to implementation. *J. ACM*, 48(5):1038–1068, 2001.
14. Christian Holzbaur. OFAI CLP(Q,R) manual. Technical Report TR-95-09, Austria Research Institute for Artificial Intelligence, 1995.
15. IC-Parc. ECL^{i}PS^e 3.4 knowledge base user manual, Sep 1994.
16. Ralph Baker Kearfott, Mitsuhiro T. Nakao, Arnold Neumaier, Siegfried M. Rump, Sergey P. Shary, and Pascal Van Hentenryck. Standardized notation in interval analysis. *Reliable Computing*, to appear.
17. Leslie De Koninck. Constraint solvers for SWI-Prolog. Master's thesis, K.U.Leuven, Belgium, 2005. in Dutch.
18. Laurent Michel and Pascal Van Hentenryck. Helios: a modeling language for global optimization and its implementation in Newton. *Theor. Comput. Sci.*, 173(1):3–48, 1997.
19. Don P. Mitchell. Robust ray intersection with interval arithmetic. In *Proceedings on Graphics interface '90*, pages 68–74, Toronto, Ont., Canada, Canada, 1990. Canadian Information Processing Society.
20. Guy Alain Narboni. From Prolog III to Prolog IV: the logic of constraint programming revisited. *Constraints*, 4(4):313–335, 1999.
21. Maurice Obstfeld and Alan M. Taylor. Nonlinear aspects of goods-market arbitrage and adjustment: Heckscher's commodity points revisited. *Journal of the Japanese and International Economies*, 11(4):441–479, 1997.
22. Swedish Institute of Computer Science. SICStus Prolog user's manual. Technical report, Oct 2001.
23. Jean-Francois Puget. Applications et évolutions d'ILOG SOLVER. In Jean-Jacques Chabrier, editor, *JFPLC*, page 429, 1995.
24. Dietmar Ratz. A nonsmooth global optimization technique using slopes | the one dimensional case, 1999.
25. Siegfried M. Rump. Expansion and estimation of the range of nonlinear functions. *Math. Comput.*, 65(216):1503–1512, 1996.
26. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In *First workshop on constraint handling rules: selected contributions*, pages 1–5, 2004.
27. Jan Wielemaker. SWI-Prolog 5.3 reference manual, Jun 2004.