Dynamical Properties of Answer Set Programs
Extended Abstract

Howard A. Blair
EECS Dept., 2-177 SciTech
Syracuse University
Syracuse, NY 13210 USA

Introduction

Solutions to problems are often not unique. A representation of a problem as a theory in some logical formalism often admits a number of models representing solutions (Marek & Truszczyński, 1999). The solution-representing models are perhaps required to come from a particular class.

The typical strategy is to represent a class of problem instances E, for example the problem of determining a Hamiltonian circuit in a digraph if there is one, as a theory T_E, and a specific instance I of E, e.g. a specific digraph, as a theory T_I in such a way that certain kinds of models of the combined theory $T_E + T_I$ represent the solutions, i.e. in the example, the required Hamiltonian circuits in I, as the result of a mapping from answer sets (the models) to solutions of I.

As a specific formalism for answer set programming, DATALOG programs with negation (Ceri, Gottlob, & Tanka, 1990) and their stable models have received a large amount of attention e.g. (Niemelä, 1998; Marek & Truszczyński, 1999; Remmel & Marek, 2001). While more recent work e.g. (East & Truszczyński, 2000) has built upon this formalism, for our purposes here we shall stay with a formalism more closely tied directly to DATALOG$^\neg$ programs. It will be seen that extensions of the basic formalism can be profitably incorporated in our dynamical systems approach.

This work identifies a certain class of DATALOG$^\neg$ programs which we call call *fully normalized.*

Definition 1: A *Fully Normalized Program* (FNP) is a DATALOG$^\neg$ program together with an Herbrand base such that every ground instance of a program clause is either a unit clause or has the form $A \leftarrow \neg B$ where A and B are atoms. An FNP P is a *2FNP* if for every ground atom A, either A is a unit clause in ground P, or A occurs in no clause head, or otherwise there are exactly two clauses in P in which A occurs occurs in the head.

In general, neither FNPs and 2FNP's are positive programs.

FNPs are representative of DATALOG$^\neg$ programs in the sense that every ground DATALOG$^\neg$ program is uniformly conservatively extendible in linear time to a ground fully normalized program. We will discuss these extensions below.

There are several reasons for focusing on FNPs. Answer set programming accords with the models-as-fixed-points idea widespread in the developments to date in nonmonotonic logics. It is a well-known easy exercise to show that for the class of programs we have called fully normalized, every supported model (i.e. fixed point) is stable. Also, it is easy to show that every propositional formula is conservatively extendible to the Clark completion of a 2FNP. In other words, the supported model semantics of FNPs (2FNPs in particular) is just the ordinary semantics of propositional logic. The search for stable models of FNPs can therefore be directly aided by SAT solvers.

Moreover, the one-step consequence operator associated with an FNP is clearly a dynamical system. Less clear, but still true, is that these dynamical systems need not be taken only in the restricted sense of discrete, symbolic dynamical systems. The connection with dynamical systems allows us import the heavy artillery of modern continuous mathematics to assist in finding and organizing the solutions to combinatorial search problems. The purpose of answer-set programming with FNPs is to provide a handy tool for representing search and decision problems as stable fixed-point finding for dynamical systems. There are several notions of stability in the dynamical systems context that accord with ordinary stable models semantics, but also extend to a notion of stability for entire execution traces. However, a focus on stable fixed points permits us to avoid thorny issues of timing in the dynamical systems.

It must however be said that the ability to take advantage of nontrivial continuous mathematics in a serious way in connection with FNPs dwarfs any considerations of their utility for everyday programming. Our view is that the interplay between the logical and dynamical properties of FNPs distinguishes them as an important *underlying* formalism to guide the development of practical programming languages and systems. We believe that familiarity with FNPs amounts to one among many devices in the programmer's conceptual toolbox, but one of an unusual kind.

Nevertheless, FNPs do have the virtue of being a practical *tool* for everyday programming when used in conjunction with other answer-set programming formalisms and techniques. We do not take the position that they should be regarded as a complete programming formalism. We will illustrate their use for crafting dynamic datastructures, below. We make no claims that we have codified a methodology of programming *exclusively* based on FNPs; indeed, we reject the idea that anyone should strive at this stage to develop such methodologies, as one, for example, should not strive to develop methodologies of programming exclusively with artificial neural networks. We do not yet know enough to do that well.

Reducing to FNPs

Remmel and Marek (Remmel & Marek, 2001) have shown that all search problems in NP are reducible to finding the set of stable models of a DATALOG$^{\neg}$ program. The strategy of Remmel and Marek's proof resembles that of Cook (Cook, 1971) in proving the NP-completeness of SAT. Programs constructed in Remmel and Marek's proof belong to the class of programs all of whose fixed-point models are stable. In this section we will partially tighten a nonuniform version of their result to show that the problem of the existence of (and the finding of) fixed-point models of 2FNPs is NP-complete. Let SSAT(2FNP) be the problem of finding stable models of 2FNP programs.

Proposition 0.1: SAT is reducible in linear time to SSAT(2FNP).

Proof: Consider the phrase structure tree of a given propositional formula φ. Obtain a directed acyclic phrase structure graph (dag) by replacing the connectives at every interior node in linear time by the connective NAND (i.e. A NAND B is $\neg(A \wedge B)$. We denote A NAND B by the more succinct $A \mid B$. For example, subformulas of the form $\neg A$ become $A \mid A$, and subformulas of the form $A \supset B$ become $A \mid (A \mid B)$. This does not necessitate the copying of subformulas, which would lead to exponential increase in the size of the resulting structure. To translate the resulting dag back to a proposition in which the only connective that occurs is NAND, we label the interior nodes of the graph with auxiliary atoms. The dag is then a representation of a finite set of equivalences, each of the form $p \leftrightarrow (q \mid r)$ where p, q and r are atoms. Each exterior node p of the dag is represented by the two equivalences $p \leftrightarrow (p' \mid p')$ and $p' \leftrightarrow (p \mid p)$. We need to ensure that the atom, call it a, that labels the root of the dag is true in any model of the equivalences. Join two additional equivalences $a' \leftrightarrow (a' \mid a'')$ and $a'' \leftrightarrow (a \mid a)$ to the collection of equivalences obtained thus far. Every model of φ is the projection of a model of the set of equivalences thus obtained and every model of the equivalences projects to a model of φ. It is easy to show that the number of equivalences, each containing three atom occurrences, is at most $11 \cdot \mathrm{size}(\varphi)$. Replace the $\leftrightarrow$ connective with $\leftarrow$. Each program clause now has the form $p \leftarrow (q \mid r)$ and each atom occurs in exactly one clause head. Finally, replace each clause $p \leftarrow (q \mid r)$ with the two clauses $p \leftarrow \neg q$ and $p \leftarrow \neg r$. The result is a 2FNP whose fixed point models project onto the models of φ. ∎

Crafting datastructures with FNPs

In this section we will assume that programs are partitionable into an intentional database (IDB) and an extensional database (EDB).

What are FNPs good for? They are good for crafting datastructures that can be changed in a uniform way using EDBs. Hence these datastructures can be naturally thought of as inputs to programs in the form of complex objects.

One can form fully normalized programs that embody familiar *dynamic* pointer data structures (e.g. linked lists, search trees, DAGs, queues, ...) within stable models of those programs. It is clearly possible to do that by encoding such structures into the call graph of a program. What is less clear is that the clauses added to the basic program systematically grow and shrink these data structures, and the various answer sets in effect embody the result of searching these data structures.

Fundamentally there are only three basic kinds of data structures: primitive data types, arrays, and pointer structures. Of course these basic kinds of structures generate a host of hybrid structures. We do not treat arrays.

The basic idea for crafting FNPs is that ground facts added to the program in the manner of the extensional database enable (i.e. "turn on") pointers between *links*. The set of all potential ground links in the program is representable as a few nonground clauses. The available links change as the signature grows.

There are three predicates, car, cdr and link used in the construction of links. We will conclude the description with an example of a linked list. Pointers come in three species: cdr-pointers and two species of car-pointers. All cdr-pointers will point from atoms of the form $\mathsf{cdr}(x)$ to atoms of the form $\mathsf{link}(y)$. car-pointers pointers point from atoms of the form $\mathsf{car}(x)$ to either atoms of the form $\mathsf{item}(y)$ or of the form $\mathsf{link}(y)$. There are multiple species of links. We give all of the clauses for the pointers at once, and then explain them. The collection of clauses representing at once, all cdr-pointers is

$$
\begin{array}{lrcl}
(1) & a(X,Y) & \leftarrow & a(X,Y) \mid b(X,Y) \\
(2) & b(X,Y) & \leftarrow & b'(X,Y) \mid b'(X,Y) \\
(3) & b'(X,Y) & \leftarrow & \mathsf{dptr}(X,Y) \mid b''(X,Y) \\
(4) & b''(X,Y) & \leftarrow & c(X,Y) \mid d(X,Y) \\
(5) & c(X,Y) & \leftarrow & e(X,Y) \mid \mathsf{cdr}(X) \\
(6) & d(X,Y) & \leftarrow & e(X,Y) \mid \mathsf{link}(Y) \\
(7) & e(X,Y) & \leftarrow & \mathsf{cdr}(X,Y) \mid \mathsf{link}(Y)
\end{array}
$$

There are two groups of clauses for car-pointers. The first group is obtained from the above seven clauses by replacing the predicate symbol cdr by the predicate symbol car and dptr by aptr. The second group is obtained from the above seven clauses by replacing cdr by car, dptr by aptr, *and* link by item.

When crafting a program using this methodology we require that the predicates, a, b, b', b'', c, d, and e not occur in any other clause head in allowable programs. With this restriction, every ground instance of these clauses satisfies the conditions for yielding fully normalized programs. This does not mean that FNPs cannot violate this requirement. It does mean that those FNPs which do violate the requirement are not constructed in accord with the methodology. Call FNPs that adhere to the requirement, *well-crafted*.

In any fixed-point model of a well-crafted FNP containing these clauses, (represented here by replacing variables by lower-case identifiers) the clauses read as equivalences. Hence, clause (1) ensures that $a(x,y)$ is true and $b(x,y)$ is false. Clause (2) then ensures $b'(x,y)$ is true. Then, if $\mathsf{dptr}(x,y)$ is true then by clause (3) $b''(x,y)$ is false, and if $\mathsf{dptr}(x,y)$ is false, then $b''(x,y)$ is undetermined. Suppose $\mathsf{dptr}(x,y)$ is true. Then by clause (4), both $c(x,y)$ and $d(x,y)$ are true. $e(x,y)$ is either true or false. If true, then by clauses (5) and (6), $\mathsf{cdr}(X)$ and $\mathsf{link}(Y)$ are both false. If $e(x,y)$ is false, then by clause (7) $\mathsf{cdr}(X)$ and $\mathsf{link}(Y)$ are both true.

Just as an informal intuition: a link should be thought of as having two fields, a car field which contains either a car-pointer or an item, and a next field which contains a cdr-pointer. All links are represented by the schema of clauses

$$
\begin{array}{lrcl}
(1) & \mathsf{link}(X) & \leftarrow & \mathsf{car'}(X) \mid \mathsf{cdr'}(X) \\
(2) & \mathsf{car'}(X) & \leftarrow & u(X) \mid \mathsf{car}(X) \\
(3) & \mathsf{cdr'}(X) & \leftarrow & v(X) \mid \mathsf{cdr}(X)
\end{array}
$$

Now, let P be a program containing the car-pointer and cdr-pointer and link clauses. Suppose we want to represent the linked list data structure $[\mathsf{item}(e_1), \mathsf{item}(e_2), \mathsf{item}(e_3)]$. We will need three links, two cdr-pointers, and three car pointers. We put the following ground facts in the extensional database: $\mathsf{item}(e_1)$, $\mathsf{item}(e_2)$, $\mathsf{item}(e_3)$, $\mathsf{dptr}(l_1,l_2)$, $\mathsf{dptr}(l_2,l_3)$, $\mathsf{aptr}(l_1,e_1)$, $\mathsf{aptr}(l_2,e_2)$, and $\mathsf{aptr}(l_3,e_3)$. We also include the ground facts $v(l_1)$, $v(l_2$, and $u(l_3)$. Then in any fixed-point model of P, the truth-value of the atom $\mathsf{link}(l_1)$ is the same as the value of atom $\mathsf{item}(e_3)$. In other words, the extensional database forces the selection of the third item in the list. By changing the extensional database to contain the ground facts, $v(l_1)$, $u(l_2)$, and $v(l_3)$ we force the selection of item e_2, for example.

The pointer and link constructions allow the construction in the extensional database of any finite pointer data structure. While the necessary clauses seem complicated, it should be noted that there only three kinds of groups of clauses. Moreover, the ability to construct these data structures obviates the need for nonunary predicates, other than those occurring in the pointer and link clause groups. Each ground atom $p(e)$ other than those reserved for pointer and links should be thought of as a node of type p labeled by e. Tuples of labels can simply be replaced by pointers to other data structures. Lastly, the entire approach is facilitated by a handy block diagram system.

The conditions for being an FNP entail that every variable that occurs in a clause body occurs in a clause head. If we relax this condition then we can both reduce the number of clauses for pointers, as well as enable the implementation of pointers in the context of continuous-valued logic. (We will continue to call such programs FNP's, reserving the term *localized* for FNPs satisfying the more stringent requirement.) In particular, in any supported model in continuous-valued logic of a well-crafted FNP that contains the following clauses

$$\begin{array}{llcl}
(1) & \mathsf{car}(X) & \leftarrow & \mathsf{car'}(X) \mid \mathsf{link'}(Y) \\
(2) & \mathsf{car'}(X) & \leftarrow & \mathsf{aptr'}(X,Y) \mid \mathsf{car}(X) \\
(3) & \mathsf{aptr'}(X) & \leftarrow & \mathsf{aptr}(X,Y) \mid \mathsf{aptr}(X,Y) \\
(4) & \mathsf{link'}(Y) & \leftarrow & \mathsf{aptr}(X,Y) \mid \mathsf{link}(Y)
\end{array}$$

if $\mathsf{aptr}(X,Y)$ has value corresponding to *true* (cf. the next section) then the values of $\mathsf{car}(X)$ and $\mathsf{link}(Y)$ are equal, and as a dynamical system, if $\mathsf{aptr}(X,Y)$ has value corresponding to *false* then the value of $\mathsf{car}(X)$ is stationary.

FNPs as dynamical systems

Iterating the one-step consequence operator T_P associated with an FNP P views the FNP as a dynamical system whose fixed points are stable models. Of course a much wider class of programs besides FNPs have this property, but FNPs are a convenience from the point of view of dynamical systems analysis, and also practical for programming as we have argued above.

Fitting (Fitting, 1994) introduced nonmonotonic logic programs that could, under a suitable pseudo-metrization of the Herbrand base of the program, be seen as a contraction mapping having a unique fixed point by the Banach contraction mapping theorem. The dynamical system idea of Fitting need not be tied to contraction mappings however.

In (Blair *et al.*, 1999) it was shown how to pass from discrete to continuous-valued programs. The sixteen arity-2 Boolean functions are representable as polynomials of the form $\lambda_1 + \lambda_2 x + \lambda_3 y + \lambda_4 xy$. Familiar examples are conjunction (xy), disjunction $(x+y-xy)$, and not-both $(1-xy)$. What is not so familiar is that any two members of an algebraic field can be chosen as truth value representatives, and all of the arity-2 Boolean functions are *uniquely* representable as polynomials of degree 1 in each of two variables for suitable choices of the λ_i. A choice of truth-value representatives amounts to a choice of a basis for the 4-d vector space of representing polynomials. In particular we can choose any two real or even complex numbers. Let us in the sequel suppose for simplicity's sake that we do use real numbers as continuous truth-values and that 0 represents the underlying truth value *false*, and similarly 1 represents *true*. The clause connective $\mid$ in an FNP is then represented by the polynomial $1-xy$.

If we compose the representing polynomials with a function f which is continuously differentiable on a compact subset with nonempty interior containing the truth values, and such that the derivative vanishes at the truth values, then any trajectory of continuous valuations of the program P' obtained by grounding P and iterating from a $0,1$-valuation produces a trajectory of $0,1$-valuations. $\frac{1}{2}(1-\cos(\pi \mathrm{x}))$ gives such a function. Moreover, there is a cylindrical neighborhood around any such trajectory which has non-zero radius (even in an infinite number of dimensions, which will be relevant in extensions of the formalism to programs with an infinite Herbrand base) such that within that trajectory all trajectories converge to the central $0,1$-valued trajectory. In effect, this property "digitizes" discrete-valued FNPs within their continuous re-interpretation. The vanishing derivative at the truth-values characterize so-called super-attractors. The central trajectory is converged upon with an accuracy with doubly increasing orders of magnitude with each iteration. The importance of this issue is that upon continualization we do not want trajectories that begin with previously discrete states to diverge from the previously discrete ones. In that circumstance we would not have achieved an *embedding* of the discrete dynamical system in the continuous one.

One can obtain an adequate concept of superattractors with a few iterations performed on a pocket calculator: Consider the functions $\mathsf{cos}(x)$ and x^2. Let x have the initial value 0.5. As one iterates $\mathsf{cos}(x)$ the value of x will be *attracted* to *the* fixed point of cosine which is approximately 0.739085133 (radians). It will be observed that the attraction is strong in the sense that at most 11 iterations are sufficient to pin down each decimal place. (That is an observation, not a theorem. Its status as a theorem is analogous to the question of whether seven consecutive 7's occur in the decimal expansion of π.) The attraction to 0.739085133 proceeds at a linear rate. Now repeat the procedure with x^2. The value of x is attracted to 0, one of two fixed points of this function, and the number of decimal places fixed by each iteration is doubled with each each iteration. The attraction to 0 occurs at an exponential rate because the derivative vanishes continuously at 0. 0 is a superattractor with respect to x^2. Conceptually, basins of superattractors are implosive.

Some salient properties of continuous-valued programs are

- FNPs crafted for datastructure preserve the datastructures when continualized. Using truth values to determine the value of a node permits the datastructures to include continuous primitive datatypes if desired.
- methods of mathematical analysis are applicable to analyzing, as well as computing the stable models of, continuous-valued programs;
- continuous-valued programs have basins of attraction around fixed point models thereby giving a probability assignment to a fixed-point model with respect to a measure introduced on the space of continuous valuations of the Herbrand base; lower bounds on the probability are given by identifying measurable subsets of the basin;
- there are hidden *homotopic* connections between the stable models of differing ordinary ground fully normalized programs; the connections are revealed through continualization;
- neural network training methods are applicable to the problem of synthesizing continuous-valued programs;
- there are Wolfram class 4 (Wolfram, 1994) emergent trajectories obtainable by tuning continuous-valued programs; (we will explain this point in greater detail below.
- discrete-valued programs, continuous-valued programs, and differential equations are unifiable, in a sense made precise by employing the device of *convergence spaces* (Wyler, 1974; Heckmann, 2000).
- there are connections between modeling systems such as *Numerica* (Van Hentenryck, Laurent, & Deville, 1997) enabling the formulation and solution of continuous nonlinear optimization problems.

Converging to fixed-point models

There are a variety of ways that stable models of continuous-valued FNP's can be found, all of which are based on an initial guess, but which is nevertheless very robust since the guess can be clumsy. The least robust approach among them is to simply iterate the program straightforwardly by iterating the one-step consequence operator. A much better approach is to apply a variant of Newton's method for finding roots that avoids having to do a matrix inversion. The basins of attraction of fixed points are, by experiment, typically large. A proper analysis of the power of this approach nevertheless remains to be carried out. However, with respect to either method it should be noted that for FNP programs that one actually crafts that contain small groups of clauses such as the link and pointer groups, these groups of clauses are rapidly convergent and in turn drive the convergence to a fixed point of the remainder of the clauses.

A third method for finding stable models is to begin with one program with its stable models and subject it to a deformation that results in the target program. The fixed points are generally observed to move under the deformation revealing path connections between the fixed points of the former program and the fixed points of the target program. Another variant of Newton's method can track the deformations of the fixed points. Moreover, Kozen and Stefansson (Kozen & Stefansson, 1997) have developed an efficient algebraic algorithm based on cell-decomposition to compute the boundaries of basins of attraction of roots of complex polynomials with respect Newton flows. Application to the higher dimensional systems arising with FNPs makes it desirable to avoid matrix inversions, and that can be accomplished by amalgamating Newton's methods with gradient descent methods as stationary locations are approached.

Emergent structures

Continuous state FNPs permit us to have continuously tunable logical connectives. In the proof of the proposition that SSAT(2FNP) is NP-complete we took advantage of the fact that every 2FNP program was equivalent to program for which every clause is either a unit clause or has the form $p \leftrightarrow (q \mid r)$ and each atom occurs in at most one clause head. We regard programs given in this form as 2FNPs also, and assume that all 2FNP programs are given in this form. Think of the connective | as a continuously tunable connective. Now consider the call graph of a 2FNP program. Call such a call graph a 2FNP *rule system* and let the set of all 2FNP programs with that call graph be the *rule space* of that system. 2FNP rule systems essentially constitute a nice generalization of cellular automata.

There are quantitative measures of the dynamics of rule systems that allow for attaching geometrically richly structured vector fields to rule spaces. These fields, which have been considered in some of the chaos and fractals literature (Markus & Tammames, 1996) as *Lyapunov spaces*, or *fat fractals*, realize a modest reformulation of Langton's edge-of-chaos idea (La90) for tuning a rule towards complex nonchaotic behavior. The fields organize the rule space into regions of rules with similar Lyapunov stability. The boundaries of the regions are sharply identifiable and near the boundaries one finds a plentiful supply of rules with complex nonchaotic behavior characteristic of Wolfram's class 4 cellular automata. The self-organizing emergent structure associated with these rules is often striking and suggestive of artificial life that has co-evolved with its environment. The speed with which the organization takes place is remarkably rapid in some experiments. These systems seem useful for exploring connections to evolutionary programming (Vitanyi, 2000).

Discrete programs with "edge-of-chaos" dynamics are extractable from continuous ones due to the fact that the dynamical behavior of a continuous-state rule with small differential Lyapunov stability is largely preserved by discrete approximations.

A common approach to discrete and continuous rule systems.

A common theory of computation for discrete and continuous rule systems involves hybrid systems that interface discrete and continuous components, and involves passing back and forth between continuous rule systems and their discrete approximations. There is a construction that generalizes the notion of a topological space, called a *convergence space* (Heckmann, 2000) within which computation can be characterized by continuity in a manner similar to domain theory. The characterization is enabled by the observation that the neighborhoods of points in topological spaces viewed as convergence spaces are topological neighborhoods, while the neighborhoods of points, i.e. nodes, in directed graphs viewed as convergence spaces are graph-theoretic neighborhoods. The fundamental idea is that discrete and continuous programs are the same things but have different kinds of models depending on the underlying convergence structure with which the Herbrand base and call graph is equipped. For example, a single program can be given that represents a Turing machine or represents an ordinary differential equation, depending on the underlying convergence structure.

References

Blair, H.A., Dushin, F., Jakel, D.W., Rivera, D.J., and Sezgin, M. 1999. Continuous models of computation for logic programs: importing continuous mathematics into logic programming's algorithmic foundations, in *The Logic Programming Paradigm*, K.R. Apt, V.W.Marek, M. Truszczyński, and D.S.Warren (eds), pp.231–255. Springer.

Blair, H.A. 2000. The Differential Scheme. *First Irish Conference on the Mathematical Foundations of Computer Science and Information Technology (MFCSIT 2000)* July 2000 (Paper submitted).

Blair, H.A. 2000. Locating self-organization at the edge of chaos. *International Conference on Complex Systems.* (May 2000) (Paper submitted).

Ceri, S., G. Gottlob and L. Tanka. 1990. *Logic Programming and Databases*, Springer-Verlag.

Clark, K.L. 1978. Negation as failure in *Logic and Data Bases.* H. Gallaire and J. Minker (eds.) Plenum Press, pp. 293–322.

Cook, S.A. 1971. The complexity of theorem-proving procedures. In *Proc. 3rd Annual ACM Symposium on the Theory of Computing*, ACM, pp. 151–158.

East, D., and Truszczyński, M. 2000. Datalog with constraints. In *Proccedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, 163–168.

Fitting, M. 1994. "Metric methods, three examples and a theorem" *Journal of Logic Programming* volume 21, 1994, pp 113–127.

Heckmann, R. A. 2000. Non-topological view of dcpo's as convergence spaces, *First Irish*

Conference on the Mathematical Foundations of Computer Science and Information Technology (MFCSIT 2000) July 2000.

Kozen, D. and K Stefansson. 1997. Computing the Newtonian graph. In *J. Symbolic Computation* 24(1997), 125-136.

La90. Computation at the edge of chaos, *Physica D*(42) pp. 12–37, 1990.

Lifschitz, V. 1999. Answer set planning. In *Proceedings of the 1999 International Conference on Logic Programming*, 1999, pp. 23-37.

Marek, V.W. and M. Truszczyński. 1999. Stable models and alternative logic programming paradigm. In K.R. Apt, W. Marek, M. Truszczyński and D.S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*, pages375-398, Springer-Verlag, 1999.

Markus, M. and J. Tammames. 1996. Fat fractals in Lyapunov space. In C. Pickover, *Fractal Horizons*, St. Martins, 1996.

Remmel, J.B. and V.W. Marek. 2001. On the foundations of answer-set programming. *this volume.*

Niemelä, I. 1998. Logic programs with stable model semantics as a constraint programming paradigm. In *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pages 72 – 79, 1998.

Van Hentenryck, P., Laurent, M. and Deville, Y. 1997. *Numerica: A Modeling Language for Global Optimization*, MITPRess, 1997.

Vitanyi, P. 2000. A discipline of evolutionary programming, *Theoretical Computer Science* 241(2000), pp. 3–23.

Wolfram, Stephen. 1994. *Cellular Automata and Complexity.* Addison Wesley, 1994.

Wyler, O. 1974. Filter space monads, regularity, completions. *General Topology and its Applications, Lecture Notes in Mathematics*, vol. 378 591–637 (1974).