The Logic in Computer Science Column

by

Yuri Gurevich

Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
gurevich@microsoft.com

Two Forms of One Useful Logic: Existential Fixed Point Logic and Liberal Datalog

Andreas Blass[*] Yuri Gurevich

Abstract

A natural liberalization of Datalog is used in the Distributed Knowledge Authorization Language (DKAL). We show that the expressive power of this liberal Datalog is that of existential fixed-point logic. The exposition is self-contained.

1 Prologue

Existential fixed point logic (EFPL) differs from first-order logic by prohibiting universal quantification (while allowing existential quantification) and by allowing the "least fixed point" operator for positive inductive definitions. A precise definition is given below.

[*] Partially supported by NSF grant DMS-0653696. Address: Mathematics Department, University of Michigan, Ann Arbor, MI 48109–1043, U.SA. Email: ablass@umich.edu

Our original motivation for developing EFPL in [1] was its appropriateness for formulating pre- and post-conditions in Hoare's logic of asserted programs [6]. In particular, the expressivity hypothesis needed for Cook's completeness theorem [3] in the context of first-order logic is automatically satisfied in the context of EFPL.

But it turned out that EFPL has many other interesting properties.

1. EFPL captures polynomial time computability on the class of structures of the form $\{0, 1, \ldots, n\}$ with (at least) the successor relation and names for the endpoints.

2. The set of logically valid EFPL formulas is a complete recursively enumerable set.

3. The set of satisfiable EFPL formulas is a complete recursively enumerable set.

4. The set of EFPL formulas that hold in all finite structures is a complete co-r.e. set.

5. When an EFPL formula is satisfied by a tuple of elements in a structure, this fact depends on only a finite part of the structure.

6. No transfinite iteration is needed when evaluating EFPL formulas, by the natural iterative process, in any structure.[1]

7. EFPL can define (given appropriate syntactic apparatus) truth of EFPL formulas.

8. Truth of EFPL formulas is preserved by homomorphisms.

9. If an EFPL formula and a first-order formula are equivalent, then they are equivalent to an existential[2] first-order formula.

Except for (7), which will be proved elsewhere, all these results are in [1]. The combination of (2) and (3) is surprising; it is possible because EFPL is not closed under negation. That is also why (7) doesn't contradict Tarski's theorem on undefinability of truth.

[1] This means that the closure ordinal of each of the iterations is at most ω, the first infinite ordinal. Contrast this with what happens when the least fixed point operator is added to full first-order logic or just to its universal fragment. As shown in the appendix to [1], arbitrarily large closure ordinals are possible there.

[2] We could nearly say "existential positive" here. Negative occurrences are needed in the existential formula only for those predicate symbols that are negative in the vocabulary of the EFPL formula.

Recently, EFPL has found a new application as the logical underpinning of the distributed knowledge authorization language DKAL [4, 5]. For this application, it was useful to recast EFPL in a form that looks similar to Datalog; it was called liberal Datalog in [5]. The purpose of the present note is to show exactly how the logic programs of liberal Datalog correspond to the formulas of traditional EFPL.

2 Introduction

Quisani: Let's return to existential fixed-point logic. We discussed it once [2], yet something bothers me about the definition.
Authors:[3] Before we get to what's bothering you, let's be sure you have the correct definition from [1].

Q: I think I know the definition all right, but to be safe let me check it with you: After making the convention that predicate symbols are classified as *positive* or *negatable*, one defines terms and atomic formulas just as in first-order logic. Compound formulas are built by

- negation, applied only to atomic formulas whose predicate symbol is negatable,
- conjunction and disjunction,
- existential quantification, and
- the LET-THEN construction.

All but the last of these have their traditional meanings as in first-order logic. The LET-THEN construction produces formulas of the form

$$\text{LET } P_1(\vec{x}^1) \leftarrow \delta_1, \dots, P_k(\vec{x}^k) \leftarrow \delta_k \text{ THEN } \psi$$

where the P_i's are distinct, new, positive predicate symbols and the δ_i and ψ are EFPL formulas in the vocabulary expanded by addition of these P_i's. Semantically, this formula means to use the δ_i's to define a monotone operator on k-tuples of predicates; given a tuple of (interpretations of) the P_i's, see which tuples $\vec{x}^i$ satisfy the δ_i's, and use those sets of tuples as the new interpretation of the P_i's. Repeat this operation until you reach a fixed point. Finally, use this least fixed point of the operator to interpret the P_i's in ψ.

A: That's right. You tacitly assumed a specific vocabulary Υ when you said that the P_i's are new, meaning they're not in Υ. In other words, although they occur in

[3] Not necessarily speaking in unison.

the LET-THEN formula you mentioned, they don't count as part of the vocabulary of that formula.

Q: Right. I think of them as bound predicate variables. They could, for example, be renamed without affecting the meaning of the formula (as long as there are no clashes).

A: Indeed, bound predicate variables are just what the P_i's become when the formula is translated into second-order logic (as in Theorem 5 of [1]). That reminds us of another comment about your description of the semantics. You repeatedly applied the operator defined by the δ_i's, until a fixed point is reached. But the semantics merely requires the least fixed point; it doesn't care about its explicit construction.

Q: OK. I guess I was giving a sort of operational semantics, whereas the logic is defined in a purely denotational way.

But I've found it convenient to think about EFPL operationally, especially when trying to write EFPL formulas to express particular properties. For example, I once checked that, in the standard model of arithmetic, $\mathfrak{N} = \langle N, 0, 1, +, \cdot, < \rangle$, the property of being a prime number is expressible by an EFPL formula.

A: By Matiyasevich's solution of Hilbert's tenth problem, this property — indeed any recursively enumerable property of natural numbers — is expressible in $\mathfrak{N}$ by an existential first-order formula; you don't need the fixed point operator.

Q: I know, but I was looking for a formula that directly expresses what it means to be prime, without detouring through clever Diophantine tricks. The formula I constructed was actually fairly complicated, mainly because of the need to simulate two bounded universal quantifiers. A natural definition of "x is prime" is that no $u < x$ divides x unless $u = 1$, and a natural definition of "u doesn't divide x" (as long as $1 \neq u < x$) is that there is no $v < x$ such that $u \cdot v = x$. These are the two bounded universal quantifications. In [1], you showed[4] how to replace some cases of universal quantification with EFPL descriptions of searches through the domains of quantification. For example, in the case of arithmetic, $(\forall w < x)P(w)$ can be replaced with

$$\text{LET } Y(u) \leftarrow u = 0 \lor \exists w\,[u = w + 1 \land Y(w) \land P(w)] \text{ THEN } Y(x).$$

I applied this idea, replacing each of the two universal quantifiers in the definition of "prime" by a search. Here's what I came up with, assuming that equality is

[4] See the proof of Theorem 3 in [1].

negatable.

$$\begin{array}{l} \text{LET } Y(u,v) \leftarrow \\ \qquad u = 0 \lor \exists w\,[u = w+1 \land Y(w,v) \land (w\cdot v \neq x \lor w = 1 \lor w = x) \\ \text{THEN} \\ \qquad \text{LET } X(u,v) \leftarrow \\ \qquad\qquad v = 0 \lor \exists w\,[v = w+1 \land X(u,w) \land Y(u,w)] \\ \qquad \text{THEN } 1 < x \land X(x,x). \end{array}$$

A: That looks correct. The first LET-clause makes $Y(u,v)$ express "x is not the product of anything $< u$ with v, except for trivial products $1 \cdot x$ and $x \cdot 1$," and then the second LET-clause makes $X(u,v)$ express "x is not a nontrivial product of anything $< u$ and anything $< v$." So, as you said, the two LET-clauses replace the two universal quantifiers in the natural first-order definition of "prime." By the way, you don't really need that equality is negatable, since you can replace $w \cdot v \neq x$ with $(w \cdot v < x) \lor (x < w \cdot v)$.

Q: That's right. And I don't really need < since $a < b$ is equivalent to $\exists y\,(a+y+1 = b)$. But I wasn't trying to minimize the vocabulary; I just wanted to make sure I see how to formalize things in EFPL.

A: OK. Did you notice that, although your formulas define the desired predicates on all of $\mathbb{N}$, you could have cut off the searches at x, for example by adding a conjunct $w < x$ for each $\exists w$? The resulting finite searches would still define "prime" correctly.

Q: I thought of that, but I decided the formula was long enough already.

A: In any case, it's clear that you know what EFPL is; so what's the problem with the definition?

Q: I thought I knew EFPL until I saw your extended abstract about the distributed knowledge authorization language, DKAL [5].

A: That abstract isn't by the two of us; it's by one of us and Itay Neeman.

Q: I know, but Itay isn't here just now and you are, so I hope you can clarify the situation for me.

A: We'll try. What exactly needs to be clarified?

Q: Section 2 of the extended abstract claims to be about existential fixed-point logic (EFPL), but it looks quite different from the logic that I learned from your paper [1] and described to you here. In particular, that section of the abstract hardly mentions quantifiers at all and makes no distinction between existential and universal quantifiers, whereas that distinction was crucial in [1].

So I decided to look at the full tech report [4]. Its Section 2 is very similar to that of the extended abstract. Its Appendix A.3 contains a quick, prose description of EFPL as defined in [1] but then ignores that and talks about logic programs and queries instead, just as Section 2 did.

As a result, I'm wondering about the connection between the "logic programs plus queries" picture in [5, 4] and the traditional picture of EFPL in [1].

A: The traditional picture in [1] corresponds exactly to the logic programs aspect described in the DKAL paper [4]. The queries in the latter paper are outside EFPL, because they include universal quantification, at least in certain circumstances.

Q: Can you show me in more detail how logic programs correspond to traditional EFPL formulas?

A: Sure. First, let's state the correspondence precisely. We deal with structures X for a vocabulary in which all predicate symbols are negatable. (So positive predicate symbols will arise only as the P's in LET-clauses.)

Theorem 1. *The relations definable, in a structure X, by EFPL formulas are the same as the superstrate relations obtained, over the substrate X, by logic programs.*

The proof is in two parts, namely translations in both directions between the two formalisms. Furthermore, the translations are uniform; that is, they do not depend on the structure X. Incorporating this uniformity into the statement of the theorem, we have the following more complete formulation.

Theorem 2. *For every EFPL formula φ with free variables among $x_1, \dots, x_n$, there is a logic program Π with a distinguished n-ary superstrate relation P such that, in every structure X, the interpretation of P defined by Π consists of exactly the n-tuples that satisfy φ. Conversely, given a logic program Π and a distinguished superstrate predicate symbol P, there is an EFPL formula φ defining, in every structure X, the set of n-tuples that Π produces as the interpretation of P.*

Q: You said "every structure X" but surely you intended some restriction on the vocabulary of X.

A: You're right. The vocabulary of X should the same as that of φ. That's also the substrate vocabulary of Π, while the full vocabulary of the program Π includes, in addition, P and (possibly) other superstrate predicate symbols.

3 From Logic Programs to Formulas

A: Let's begin by considering a logic program of the sort described in [4]. To recapitulate that description, we have a vocabulary Υ divided into a *substrate* part (which may contain relation and function symbols) and a *superstrate* part (containing only relation symbols). Substrate (resp. superstrate) formulas are those whose relation symbols are all in the substrate (resp. superstrate) part of Υ; note that a superstrate formula is allowed to contain substrate function symbols.

Q: What's the intuition behind substrate and superstrate?

A: The idea is that the substrate relations are given to us and the superstrate relations are computed by means of the program. That intuition is reflected in the semantics, which we'll review in a moment, but first let's finish the description of the syntax of logic programs.

A *logic rule* in the vocabulary Υ has the form $H \leftarrow B$, where the *head* H is an atomic superstrate formula and the *body* B is a conjunction of atomic superstrate formulas and possibly a quantifier-free substrate formula. A *logic program* is a finite set of logic rules.

Semantically, a logic program is to be interpreted in a given structure X for the substrate vocabulary. It defines interpretations for the superstrate relations as the least fixed point of all the rules in the program. That is, interpret each rule

$$R(t_1, \dots, t_n) \leftarrow B$$

in the program as an instruction to increase the current interpretation of R by adding all those tuples $(a_1, \dots, a_n)$ of elements of X such that some assignment of values (in X) to the variables makes B true and gives each t_i the value a_i. Formally, this amounts to an operator Γ on tuples of relations regarded as interpreting all the superstrate relations (or, equivalently, on Υ-structures whose reduct to the substrate is X). Repeatedly apply this operator until a fixed point is reached. The desired interpretations of the superstrate relations constitute the least (with respect to componentwise inclusion) fixed point of Γ. We'll see that the superstrate relations obtained in this way can be defined in X by EFPL formulas in the sense of [1]. In fact, we'll obtain the required formulas in a simple, explicit manner from the given program Π.

As a first step, we can rewrite Π so that the head of each rule involves no function symbols, i.e., each head looks like $R(x_1, \dots, x_n)$ where the x_i are variables.

Q: This was already explained in [4] in the context of an example, but the method is clearly general. Given a rule of the form $R(t_1, \dots, t_n) \leftarrow B$, where the t_i are terms that need not be variables, replace it with

$$R(x_1, \dots, x_n) \leftarrow B \wedge \bigwedge_{i=1}^{n} (x_i = t_i)$$

where the x_i are distinct, fresh variables. This modification of Π has no effect on the operator Γ that it defines, so the superstrate relations are unchanged.

A: Right. When making these modifications to Π, you can also arrange that, if the same relation symbol R occurs in the head of several rules, then the same tuple of variables $x_1, \dots, x_n$ is used for all these occurrences.

At this point, we'll gradually move from the syntax of logic rules to the syntax of EFPL. Specifically, we'll modify the rules of our program some more, and the resulting bodies will no longer have the form required in logic rules but rather will be EFPL formulas.

If several rules in the program have begin with the same superstrate symbol R and therefore, by the preceding normalization, have the same head H, then we combine these rules $H \leftarrow B_1, \dots, H \leftarrow B_k$ into a single rule

$$H \leftarrow (B_1 \vee \cdots \vee B_k).$$

Q: This use of disjunction was allowed in [4, Appendix A.2.2].

A: Yes, but there it was regarded as syntactic sugar, a mere abbreviation of the k separate rules $H \leftarrow B_j$. Now, we want to regard it as a single rule in its own right.

Next, if the body of a rule contains variables other than those in the head, quantify them existentially. That is, replace $R(x_1, \dots, x_n) \leftarrow B$ with

$$R(x_1, \dots, x_n) \leftarrow (\exists y_1) \dots (\exists y_r)\, B$$

where $y_1, \dots, y_r$ are all the variables in B other than $x_1, \dots, x_n$. Is it clear that this change doesn't affect the superstrate relations?

Q: Yes. In fact, it doesn't change the operator Γ used to define those relations. The essential point is that the definition of Γ was already in terms of "some assignment."

A: Good. Notice also that the bodies of our rules, after these modifications, are still EFPL formulas.

Q: Sure. In fact, they're in the existential fragment of first-order logic.

A: Right. Let Π' be the current, modified program[5], let R be a superstrate relation, say n-ary, and consider the EFPL formula $\varphi(x_1, \dots, x_n)$ given by

$$\text{LET } \Pi' \text{ THEN } R(x_1, \dots, x_n).$$

[5] To agree exactly with the syntax of [1], Π' should be regarded as a sequence of rules, rather than a set, by ordering its rules arbitrarily. This pedantry is required because in [1] we defined the fixed-point construction "LET … THEN …" using sequences between the LET and the THEN. Sequences have the advantage that formulas are strings of symbols; sets would have the advantage of mathematical elegance, since the ordering in the sequence never matters. The same comments apply to logic programs. It is curious that the directly writable, sequence convention is used in the mathematically oriented paper [1] while the more elegant, abstract, set convention is used in the application-oriented paper [4].

Q: Are the variables x_i after THEN the same ones that were used with all occurrences of R in head position in the program?

A: They might as well be, but it doesn't matter. All variables in the Π' part of φ are bound, either by the quantifiers that we explicitly introduced into the bodies of rules or by the "LET ... THEN" construction. So it doesn't matter which variables they are. The only reason we insisted on having the same variables for all occurrences, in heads, of the same relation symbol is to be able to combine those rules into a single rule. Thus in Π' each superstrate relation symbol occurs in the head of exactly one rule, as required by the syntax of EFPL.

Q: Wait a minute. I see why each superstrate relation symbol occurs in the head of at most one rule in Π'; if it was originally in more than one, then you combined those rules using disjunction. But why is it in exactly one? What if some superstrate symbol doesn't occur at all?

A: We ignored that situation because such a symbol would be interpreted as the empty relation over any substrate, so there's really no point in including it in the superstrate. But, to be accurate, we should cover this case as well, and the disjunction idea still works. So if the original program had no rules starting with the superstrate symbol R, then Π' would have one such rule, $R(x_1, \dots, x_n) \leftarrow B$, where B is the disjunction of no formulas (the bodies of all the 0 rules with R in the head). Since the disjunction of no formulas is, by the only reasonable convention, `false`, we get the rule $R(x_1, \dots, x_n) \leftarrow$ `false`, which has the right semantical effect.

Q: That's a pretty sneaky answer.

A: It was a pretty sneaky question.

Coming back to the EFPL formula φ defined above, is it clear that the relation it defines is exactly the interpretation of the superstrate relation R that is produced by the original logic program Π?

Q: Almost. It's clear that, if we interpreted Π', where it occurs in φ, in the same way that logic programs are interpreted, then it would produce the same superstrate relations as Π. But the interpretation of rules, and specifically the operator Γ, is not quite the same in logic programs as in EFPL. In the semantics of logic programs, the set of tuples described by a rule is *added* to the current interpretation of the relevant superstrate relation symbol. In the semantics of EFPL, the same set of tuples *alone* constitutes the new interpretation of that symbol. In other words, the Γ operator for logic programs [4] is explicitly designed to be inflationary; that of EFPL [1] need not be inflationary.

A: That is true, but it doesn't matter. A not-necessarily-inflationary operator Δ and its explicitly inflationary variant Γ defined by $\Gamma(A) = \Delta(A) \cup A$ have the same closed points.

Q: What do you mean by closed points?

A: A closed point of Γ is an A such that $\Gamma(A) \subseteq A$. It's fairly common terminology to say that a set is closed under an operator. We say "closed point" rather than "closed set" because, when there are several superstrate predicate symbols, our operators act on tuples of relations, not on single sets.

Coming back to the situation of an operator Δ and its inflationary variant Γ, it's clear that they have the same closed points. For any monotone operator, the least fixed point is also the least closed point. And our operators are monotone, because superstrate relations occur only positively in the bodies of logic programs (even after we modify the programs as above). So Γ and Δ have the same least fixed point.

Q: OK. Actually, I now see another reason why we can ignore the inflationary aspect of the Γ in [4]. If we think of the least fixed point of a monotone operator Δ as being constructed by iterating Δ, then the sequence of iterates is non-decreasing. So, for every A in this sequence, $\Delta(A) = \Gamma(A)$.

A: Right. So this completes the proof that the superstrate relations defined, over a given substrate structure X, by a logic program Π, as in [4], are also defined over X by EFPL formulas φ. Furthermore, the transformation of Π into φ is uniform, i.e., the same for all X.

Q: Yes. In fact it proves a bit more, namely that any logic program can be translated into EFPL formulas of a rather special form: A single LET ... THEN construct, where the part after THEN is an atomic formula consisting of a relation symbol followed by variables. Furthermore, the bodies of the inductive definitions (rules) after LET are existential first-order formulas.

That worries me, because for the other half of the equivalence between logic programs and EFPL, you'll have to find logic programs equivalent to arbitrary EFPL formulas, not just those of this special form.

A: That's right, but you needn't worry. Every EFPL formula is equivalent to one in this special form, and the equivalence to logic programs is one way to prove this.

4 From Formulas to Logic Programs

A: Given an arbitrary EFPL formula $\varphi(u_1, \dots, u_n)$ with free variables among those indicated, we shall transform it into a logic program Π such that a particular n-ary superstrate relation, as defined by Π over any substrate X, is the set of n-tuples that satisfy φ in X. That will complete the proof that logic programs and EFPL formulas are equivalent; they can be regarded as two ways of presenting the same logic.

As a first step, we'll show that every EFPL formula is logically equivalent to one of the special form

$$\text{LET } P_1(\vec{x}^1) \leftarrow \delta_1, \dots, P_k(\vec{x}^k) \leftarrow \delta_k \text{ THEN } \psi$$

where all the formulas δ_i and ψ are existential first-order formulas and where the free variables of any δ_i are among the variables $\vec{x}^i$ serving as the arguments of the corresponding P_i in the definition $P_i(\vec{x}^i) \leftarrow \delta_i$.

Q: Since "LET ... THEN" binds these $\vec{x}^i$s, the only free variables in such a formula are those in ψ.

A: Right; that will simplify part of the proof. We should also mention that $k = 0$ is allowed; then the formula above amounts to just ψ.

Q: I bet your proof that all EFPL formulas are equivalent to ones of this special form is an induction on formulas, and by allowing $k = 0$ you've made the cases of atomic formulas and of negated atomic formulas trivial (whereas otherwise they would only have been obvious).

A: Right on both counts. Conjunction and disjunction are also easy. Given two formulas in the desired form, rename the bound predicate variables of the LET-clause — the P_i's in the notation above — in one of them so as to be distinct from those of the other formula. Then just combine their LET-clauses and form the conjunction or disjunction of the THEN-clauses.

Existential quantification is even easier. Leave the LET-clause alone and quantify the THEN-clause.

Q: This simple argument for the existential quantifier case makes use of your convention that the only free variables in δ_i are among the $\vec{x}^i$. Without this convention, you'd have to consider the possibility that some other variable free in some δ_i is being quantified.

But, in keeping with the principle that there's no free lunch, it seems that you'll have to pay for this convention in the one remaining case of your induction. Given a LET-THEN formula, you'll have to get rid of any extraneous free variables in its LET-clause.

A: You're right, but in this case the lunch is fairly cheap.

Suppose we're given a formula LET $P(x) \leftarrow \delta$ THEN ψ where both δ and ψ are of the desired form.

Q: Wait a minute. Are you assuming that there's only one constituent $P(x) \leftarrow \delta$ in the LET part and that P is unary?

A: Yes, but this is only for notational simplicity.[6] The general case would involve a lot of subscripts but no new ideas. Notice, in particular, that if we have several P's with their corresponding δ's, and if each δ has a LET-clause with several constituents, defining predicates Q, then each of those Q's needs two subscripts — the first to tell which δ it's in and the second to tell where it is in that δ's LET-clause — and the range of the second subscript depends on the first. Subscript-juggling in such a case tends to obscure the proof.

Q: OK, go ahead with your "one unary predicate" proof. Maybe afterward I'll figure out all the subscripts for the general case on my own.

A: Let's start by taking our formula,

$$\text{LET } P(x) \leftarrow \delta \text{ THEN } \psi,$$

and showing how to eliminate any extraneous free variables from δ. Continuing to avoid uninformative subscripts, let's suppose y is the only variable other than x that is free in δ. Then we claim our formula is equivalent to

$$\text{LET } P'(x, y) \leftarrow \delta' \text{ THEN } \psi',$$

where δ' and ψ' are obtained from δ and ψ by replacing each atomic subformula of the form $P(t)$ with $P'(t, y)$. (Of course, we assume that bound variables in δ and ψ have been renamed if necessary so that the y's introduced here don't become accidentally bound.)

To see that the new formula is equivalent to the original, consider the binary relation obtained as (the interpretation of) P' from the recursion $P'(x, y) \leftarrow \delta'$. If you fix any particular value $b \in X$ for y, then the resulting unary relation, $P'(x, b)$ is exactly the relation defined by the original clause, $P(x) \leftarrow \delta$ with y assigned the value b.

Q: Yes, that's easy to see if you think of the iterative process leading to the fixed points that interpret P and P'. In the new clause, $P'(x, y) \leftarrow \delta'$, y behaves simply as a parameter. So, as the binary relation defined by this clause grows, from $\varnothing$ toward the fixed point P', its unary section obtained by fixing the second argument as b grows exactly according to the original clause $P(x) \leftarrow \delta$ with y denoting b. In particular, the agreement between P and a section of P' occurs not only for the final fixed points but stage by stage during the iteration.

A: That's right. But one can also verify the final agreement directly in terms of least fixed points without referring to the iteration. If P' is the least fixed point of

[6] It is also known that, at least in the presence of two constant symbols, simultaneous positive recursions can be reduced to a single positive recursion, albeit for a relation of higher arity. See [7, Theorem 1C.1]. The context in [7] is positive recursion over full first-order logic, but universal quantification isn't used for this theorem.

the new iteration, then each of its sections, say at b, is the least fixed point of the old iteration with y denoting b.

Q: I see that the section is a fixed point of the old operator, simply because P' itself is a fixed point of the new one, but why is it the least fixed point?

A: If you had a smaller fixed point P^- for the old operator, then you could replace the b section of P' with P^- while leaving all the other sections unchanged. The result would be a smaller fixed point than P' for the new operator. The point here is that we can modify a single section independently of the others because the new operator works on each section separately.

Q: I see; this is what I expressed earlier by saying that y behaves simply as a parameter.

OK, so you can get rid of extraneous free variables in δ. But your unary P has become a binary P'.

A: It's still the case that higher arities (or more P's) contribute only notational complications. So, if you don't mind, we'll revert to the unary notation and we'll drop the primes on P, δ, and σ. In other words, we'll return to the original notation LET $P(x) \leftarrow \delta$ THEN ψ but with the assumption that only x is free in δ.

Q: OK; I never was a big fan of subscripts.

A: Good. So let's consider this formula φ:

$$\text{LET } P(x) \leftarrow \delta \text{ THEN } \psi.$$

By induction hypothesis, we know that δ and ψ are already of the desired form, say δ is

$$\text{LET } R(y) \leftarrow \rho \text{ THEN } \pi,$$

and ψ is

$$\text{LET } S(z) \leftarrow \sigma \text{ THEN } \theta,$$

where $\rho, \pi, \sigma, \theta$ are existential first-order formulas, and their free variables are among those indicated here:

$$\rho(y), \quad \pi(x), \quad \sigma(z), \quad \theta(u).$$

Q: What's this u? It wasn't in any of the previous formulas.

A: u represents whatever variables are free in the whole formula φ. They were called $u_1, \dots, u_n$ at the beginning of this half of the proof, but, as usual, we now pretend, for notational simplicity, that there's only one such variable.

Q: OK. All your other limitations on free variables are based on the fact that, by induction hypothesis and by the preceding discussion, none of the three LET-clauses have extraneous free variables. In particular, any variable free in π would also be free in δ and therefore can only be x.

A: That's right. Let's write out φ in detail, exhibiting not only the free variables in each part (as above) but also the predicate variables, P, R, S, that could occur in each part. So φ looks like

$$\begin{aligned}&\text{LET } P(x) \leftarrow [\text{LET } R(y) \leftarrow \rho(P, R, y) \text{ THEN } \pi(P, R, x)]\\ &\qquad\qquad\qquad\qquad \text{THEN } [\text{LET } S(z) \leftarrow \sigma(P, S, z) \text{ THEN } \theta(P, S, u)].\end{aligned}$$

Q: Please wait a minute while I check your claims about which predicates can occur where. ... OK, I agree with what you wrote. The point is that the predicate variable introduced before a ← in a LET-clause is allowed to occur at the right of that ← in that LET-clause and also in the associated THEN-clause but not elsewhere.

A: Right. Now we claim that φ is equivalent to the following formula φ':

$$\begin{aligned}&\text{LET } P(x) \leftarrow \pi(P, R, x),\ R(y) \leftarrow \rho(P, R, y),\ S(z) \leftarrow \sigma(P, S, z)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } \theta(P, S, u).\end{aligned}$$

Q: Essentially, you've just lumped together all the LET-clauses in φ, ignoring the nesting of the clause for R inside the clause for P, and made one big LET-clause out of all of them. Not very subtle.

A: But it works. The first step toward the proof that it works is setting up some notation that is neither cumbersome nor ambiguous. (Either problem alone is easily avoided.) We propose the following.

Fix a structure and a value for the free variable u of φ. Since they're fixed, we won't mention them in our notation. To further simplify the notation, we'll generally use the same symbols for syntactic entities (like the predicates P, R, S and the variables x, y, z) and possible semantic interpretations of them in our structure.

Now let's look at our formula φ and set up some notation for the various fixed points that occur in it. We begin with the LET-clause $R(y) \leftarrow \rho(P, R, y)$ that defines the fixed-point interpretation for R.

Q: Why not start with the first LET-clause, the one for P?

A: The defining formula δ in that clause involves the fixed point for the R clause, so it's useful to settle the R part of the notation first.

For any particular interpretation of the predicate P — and, as indicated above, we'll use the same symbol P for the interpretation — ρ defines a least fixed point that we'll call $R^\infty(P)$.

Next, the LET-clause for P amounts to using the definition $\pi(P, R, x)$ but with R interpreted as $R^\infty(P)$. (Indeed, that's the semantics of "LET $R(y) \leftarrow \rho(P, R, y)$ THEN $\pi(P, R, x)$.") Note carefully that the monotone operator described by this clause,

$$P \mapsto \{x : \pi(P, R^\infty(P), x)\}$$

uses its input P twice — first in the first argument of π and again via the dependence on P of the second argument $R^\infty(P)$. We denote the least fixed point of this operator by P^∞.

Similarly, the LET-clause for S uses the definition $\sigma(P, S, z)$ with P interpreted as P^∞. We write S^∞ for the least fixed point of this operator.

Next, we need notation for the three predicates obtained as the least fixed point of the simultaneous recursion in φ'. Having already used superscripts ∞ for another purpose, we'll use stars instead for this triple of fixed points, calling them P^*, Q^*, R^*.

In connection with all these fixed points, it is useful to remember that the least fixed point is also the least closed point. Thus, for example, P^*, Q^*, R^* can be characterized as the smallest relations that simultaneously satisfy the implications

$$\forall x\, [\pi(P^*, R^*, x) \implies P^*(x)] \tag{1}$$
$$\forall y\, [\rho(P^*, R^*, y) \implies R^*(y)] \tag{2}$$
$$\forall z\, [\sigma(P^*, S^*, z) \implies S^*(z)] \tag{3}$$

(whereas for fixed points we'd write bi-implications).

The essence of the proof is to show that the fixed points arising from φ and from φ' match as follows.

$$P^* = P^\infty, \quad R^* = R^\infty(P^*), \quad \text{and} \quad S^* = S^\infty.$$

Q: It's clear that, once you establish these equations, you're done. After all the truth value of φ is, by definition, obtained by evaluating θ with P and S interpreted as P^∞ and S^∞, while the truth value of φ' is obtained by evaluating the same θ (in the same structure with the same value for u, as fixed earlier) with P and S interpreted as P^* and S^*. In fact, this last part of the proof won't need the equation for R, with its curious mixture of ∞ and $*$ on the right side; presumably that equation is needed as an intermediate step in the proof of the other two equations.

So please go ahead and prove the $* = \infty$ equations.

A: OK. We'll start by showing that $R^* = R^\infty(P^*)$. Formula (2) says exactly that R^* is a closed point of the operator $R \mapsto \{y : \rho(P^*, R, y)\}$. According to the definition of $R^\infty(P)$, applied with P instantiated as P^*, the least closed point of this operator is $R^\infty(P^*)$. So we immediately have that $R^\infty(P^*) \subseteq R^*$.

Furthermore, all three of the formulas (1), (2), and (3) remain true if we replace R^* by $R^\infty(P^*)$. For (1), this follows from the fact that R is a positive predicate symbol (otherwise it couldn't have occurred on the left of $\leftarrow$ in φ) and so π is monotone with respect to R. When we replace R^* by $R^\infty(P^*)$, the interpretation of R can only decrease. That strengthens the antecedent in (1) and thus preserves the truth of (1). The argument for (2) is easier; the result of replacing R^* by $R^\infty(P^*)$ there is just the fact that $R^\infty(P^*)$ is by definition closed under the operator defined by π with P interpreted as P^*. Finally, the case of (3) is trivial, as R^* doesn't occur there.

Thus, the triple $P^*, R^\infty(P^*), S^*$ is closed under the simultaneous recursive operator whose least closed point is P^*, R^*, S^*. Therefore $R^* \subseteq R^\infty(P^*)$, and the $* = \infty$ equation for R is proved.

Let's turn next to the equation for P. In view of what we've already proved, formula (1) can be written as

$$\forall x\,[\pi(P^*, R^\infty(P^*), x) \implies P^*(x)],$$

which says that P^* is closed under the operator $P \mapsto \{x : \pi(P, R^\infty(P), x)\}$, whose least closed point is P^∞. So we have $P^\infty \subseteq P^*$.

Furthermore, all three of the formulas (1), (2), and (3) remain true if we replace P^* by P^∞ and replace R^* by $R^\infty(P^\infty)$. In the case of (1), this is just the closure condition defining P^∞. In the case of (2), it's the closure condition defining $R^\infty(P^\infty)$. And in the case of (3), it follows from the facts that σ is monotone with respect to P and $P^\infty \subseteq P^*$.

Thus, the triple $P^\infty, R^\infty(P^\infty), S^*$ is closed under the simultaneous recursive operator whose least closed point is P^*, R^*, S^*. Therefore $P^* \subseteq P^\infty$, and the $* = \infty$ equation for P is proved.

Finally, we turn to S. In view of the equations already proved, formula (3) is equivalent to

$$\forall z\,[\sigma(P^\infty, S^*, z) \implies S^*(z)],$$

which says that S^* is closed under the operator whose least closed point is S^∞. Therefore, $S^\infty \subseteq S^*$.

Furthermore, all three of the formulas (1), (2), and (3) remain true if we replace P^* by P^∞, R^* by $R^\infty(P^\infty)$, and S^* by S^∞. Since S^* doesn't occur in (1) and (2), the argument given for them above still applies. As for (3), the formula we get is just the closure requirement in the definition of S^∞.

Thus, the triple $P^\infty, R^\infty(P^\infty), S^\infty$ is closed under the simultaneous recursive operator whose least closed point is P^*, R^*, S^*. Therefore, $S^* \subseteq S^\infty$, and so the proof is complete.

Q: You mean the proof of the $* = \infty$ equations. So you've completed the inductive proof that every EFPL formula can be put into the normal form LET $P(x) \leftarrow \delta$ THEN ψ, with δ and ψ in existential first-order logic. And you have the additional normalization that δ has no free variables but x (except that you might really have lots of P's of arbitrary arities). You still have to convert this into a logic program of the form used in [4].

A: Right, but the rest is fairly easy.

First, we can arrange that the formula ψ after "THEN" is an atomic formula $Q(\vec{u})$ where $\vec{u}$ is a tuple of distinct variables (not compound terms). Indeed, if we let $\vec{u}$ list all the variables free in the given formula LET $P(x) \leftarrow \delta$ THEN ψ (hence free in ψ) and if we let Q be a new, positive predicate symbol of the right arity, then this given formula is easily equivalent to

$$\text{LET } P(x) \leftarrow \delta,\ Q(\vec{u}) \leftarrow \psi \text{ THEN } Q(\vec{u}).$$

Q: I see: Since Q doesn't occur in ψ, the "recursive" definition $Q(\vec{u}) \leftarrow \psi$ isn't really recursive. The relevant iteration takes just one step (once P has reached its fixed point) and, in effect, makes $Q(\vec{u})$ an alias for ψ.

A: Right. Furthermore, as Q doesn't occur in δ, nothing has changed in the recursive definition of P.

So our formula has been equivalently rewritten as

$$\text{LET } P_1(\vec{x}^1) \leftarrow \delta_1,\ \ldots,\ P_k(\vec{x}^k) \leftarrow \delta_k \text{ THEN } P_k(\vec{u}).$$

Q: You must really be getting near the end of the proof, since you've restored the multiple P's and their subscripts.

A: Yes. What remains is to transform each of the existential first-order formulas δ_i as follows.

We can put each δ_i into prenex form and then put its quantifier-free matrix into disjunctive normal form. So δ_i now looks like

$$\exists \vec{y}^i \bigvee_r \bigwedge_s \alpha_{i,r,s}(\vec{x}^i, \vec{y}^i),$$

where the α's are atomic or negated atomic formulas.

Then the required logic program Π consists of the rules

$$P_i(\vec{x}^i) \leftarrow \bigwedge_s \alpha_{i,r,s}(\vec{x}^i, \vec{y}^i),$$

one for each i and r.

Q: This is very similar to what happened in the first part of the proof, translating logic programs into EFPL formulas. The monotone operators defined by the transformed EFPL formula and by the logic program are identical as operators. So they certainly have the same least fixed point. In particular, the P_k component of that least fixed point, which is one of the superstrate relations defined by the program, is the interpretation, in the substrate, of the original EFPL formula.

Remark 3. Although, as mentioned in a footnote earlier, [4] is directed toward applications and [1] is more theoretical, EFPL does have one advantage over liberal Datalog from the programming point of view. In a liberal Datalog program, all variables are global, but in an EFPL formula, the LET-THEN construction provides local variables with scopes. The latter can be important for large-scale programming, by making it easy to assemble small modules into a large program (or formula).

References

[1] Andreas Blass and Yuri Gurevich, "Existential fixed-point logic," in *Computation Theory and Logic,* ed. E. Börger, Springer-Verlag, Lecture Notes in Computer Science 270 (1987) 20–36.

[2] Andreas Blass and Yuri Gurevich, "The underlying logic of Hoare Logic," *Bull. Eur. Assoc. Theor. Comput. Sci.,* 70 (2000) 82–110, reprinted in *Current Trends in Theoretical Computer Science: Entering the 21st Century,* eds. G. Paun, G. Rozenberg, and A. Salomaa, World Scientific (2001) 409–436.

[3] Stephen Cook, "Soundness and completeness of an axiom system for program verification," *SIAM J. Computing* 7 (1978) 70–90.

[4] Yuri Gurevich and Itay Neeman, "DKAL: Distributed knowledge authorization language," Microsoft Research Technical Report MSR-TR-2007-116, revised version MSR-TR-2008-09.

[5] Yuri Gurevich and Itay Neeman, "DKAL: Distributed knowledge authorization language," extended abstract of [4], in 21st IEEE Computer Security Foundations Symposium (CSF 2008).

[6] C. A. R. Hoare, "An axiomatic basis for computer programming," *Comm. ACM* 12 (1969) 576–580, 583.

[7] Yiannis N. Moschovakis, *Elementary Induction on Abstract Structures,* Studies in Logic and the Foundations of Mathematics 77, North-Holland (1974).