Stratified least fixpoint logic

Kevin J. Compton

Rapports de Recherche

1992

25ème anniversaire

N° 1674

Programme 2
Calcul Symbolique, Programmation et Génie logiciel

STRATIFIED LEAST FIXPOINT LOGIC

Kevin COMPTON

Mai 1992

RR_1674

Stratified Least Fixpoint Logic

Kevin Compton

Abstract. *Stratified least fixpoint logic*, or *SLFP*, characterizes the expressibility of stratified logic programs and, in a different formulation, has been used as a logic of imperative programs. These two formulations of *SLFP* are proved to be equivalent and a complete sequent calculus for *SLFP* is presented. It is argued that *SLFP* is the most appropriate assertion language for program verification. In particular, it is shown that traditional approaches using first-order logic as an assertion language only restrict to interpretations where first-order logic has the same expressibility as *SLFP*.

Logique de Plus Petit Point Fixe Stratifiée

Résumé. *La logique de plus petit point fixe stratifiée*, ou *SLFP*, caractérise l'expressibilité des programmes logiques stratifiés et, dans une formulation différente, a été utilisée comme logique de programmes impératifs. Nous prouvons que les deux formulations de la *SLFP* sont équivalentes et nous présentons un calcul des séquents complet pour la *SLFP*. Nous soutenons de point de vue que la *SLFP* est le langage assertionnel le plus approprié pour la vérification de programmes. En particulier, nous montrons que les approches traditionnelles utilisant la logique du premier ordre comme langage assertionnel se réduisent seulement à des interprétations là où la logique du premier ordre a la même expressibilité que la *SLFP*.

Stratified Least Fixpoint Logic

Kevin J. Compton
INRIA, Rocquencourt
78153 Le Chesnay Cedex
FRANCE
compton@margaux.inria.fr

10 April 1992

Abstract

Stratified least fixpoint logic, or *SLFP*, characterizes the expressibility of stratified logic programs and, in a different formulation, has been used as a logic of imperative programs. These two formulations of *SLFP* are proved to be equivalent and a complete sequent calculus for *SLFP* is presented. It is argued that *SLFP* is the most appropriate assertion language for program verification. In particular, it is shown that traditional approaches using first-order logic as an assertion language only restrict to interpretations where first-order logic has the same expressibility as *SLFP*.

1 Introduction

Although the logical foundations for both logic programming and program verification have been widely studied (see Apt [4], Kanellakis [26], Cousot [15], and Kozen and Tiuryn [29]), there is a close connection between the two that has generally gone unnoticed. We shall study a logic that was introduced independently by researchers in these two areas for quite different reasons. In logic programming, this logic characterizes the expressibility of stratified logic programs. For that reason we will call it *stratified least fixpoint logic* or *SLFP*. This logic is equivalent to the formally continuous μ-calculus introduced by Park [38]. We prove this equivalence, which is not immediately obvious, and then present a sound and complete sequent calculus (or Gentzen-style deductive system) for *SLFP*. From this we will derive some implications for program verification. We then argue that *SLFP*, not first-order logic, is the most appropriate assertion language for program verification. Finally, we prove some results about the expressibility of *SLFP* showing that a widely used approach to the difficulties of using first-order logic as an assertion language really just restricts to structures where first-order logic has the same expressibility as *SLFP*.

Stratified logic programming was devised as a means to introduce a limited form of negation in logic programming. The idea was first discovered by Chandra and Harel [12] and has been investigated, and sometimes rediscovered, by many others (see Apt and Blair [5], Apt, Blair, and Walker [6], Barbuti and Martelli [9], Naish [36], Przymusiński [39], and Van Gelder [44]).

For Chandra and Harel, the idea arose naturally from consideration of a logic they called *YE*. We will call it *existential least fixpoint logic* or *ELFP*. This logic expresses the queries of (unstratified) logic programs. It has a least fixpoint operator, but allows only existential quantification. Also, negation may be applied only to atomic formulas containing no relation variables. In logic programming, this corresponds to forbidding negation of intensional symbols. (Definitions pertaining to logic programming are given in section 2.) No problems arise with the queries expressed by such programs because intensional relations are defined by a least fixpoint construction from extensional relations and their complements.

An obvious generalization is to consider programs whose relation symbols may divided into *strata* so that the intensional relations in one stratum are defined by a least fixpoint construction from relations and complements of relations defined in lower strata. This kind of reasoning led Chandra and Harel to the notion of stratified logic programming. They did not go on to formulate a logic that corresponds to stratified logic programs as *ELFP* corresponds to logic programs without negation of intensional symbols, but it is straightforward to do so from their paper. (They mistakenly asserted that stratified logic programs have the same expressive power as least fixpoint logic; Dahlhaus [16] and Kolaitis [28] gave a counterexample to this assertion.)

Park [38] formulated the formally continuous μ-calculus for entirely different reasons than Chandra and Harel. Rather than extending the expressibility of a more limited logic, such as *ELFP*, he sought to restrict the expressibility of a more general logic, the μ-calculus or least fixpoint logic. This logic is obtained by adding to first-order logic the capability to describe least fixpoints or inductive definitions. Park had used the μ-calculus earlier [37] as a formalism to express induction principles for program proving. Later Aho and Ullman [2] rediscovered this logic and proposed that it be used as a database query language. In Park's formulation, this logic has relation variables which interpret inductively defined relations: one may specify the least relation $P(\vec{x})$ holding whenever $\vartheta(P, \vec{x})$ holds. To guarantee the existence of such a relation, P is required to occur only positively in ϑ (i.e., always within the scope of an even number of negations). This is a sufficient condition for the function $F_\vartheta(P)$, which maps P to the set of values $\vec{a}$ satisfying $\vartheta(P, \vec{a})$, to be monotone.

The least fixpoint construction justifies the term *inductive definition*: the least fixpoint of F_ϑ results from repeated application of F_ϑ starting from the empty relation. It may be necessary to apply F_ϑ a transfinite number of times, taking unions at limit ordinals. If F_ϑ happens to be *continuous* (i.e., $F_\vartheta(\bigcup_{n\in\omega} P_n) = \bigcup_{n\in\omega} F_\vartheta(P_n)$ whenever $P_0 \subseteq P_1 \subseteq \cdots$) then this construction converges by stage ω. This is often desirable from a computational viewpoint. Park's idea was to further restrict the syntax of the μ-calculus so that F_ϑ will always be continuous, not just monotone. He required that negation be applied only to formulas with no free relation variables. This gives a logic between *ELFP* and least fixpoint logic. De Roever described a similar logic around the same time and made the observation that the sentences of his logic were "syntactically continuous" (see [17]).

It is not difficult to see that *ELFP* is strictly less expressive than *SLFP* even on finite structures. Blass and Gurevich [11], for example, show that *ELFP* sentences are preserved by extensions. But since *SLFP* contains first-order logic, there are *SLFP* sentences that are not preserved by extensions. Dahlhaus [16] and Kolaitis [28] proved that *SLFP* is strictly less expressive than least fixpoint logic on finite structures. In their proofs they considered the "existential fragment" of least fixpoint logic. This fragment is equivalent in expressive power to *SLFP*.

Kolaitis also showed that *SLFP* is strictly less expressive than least fixpoint logic on infinite structures. We will give another proof of this in section 5 as a corollary to a result on the expressive power of *SLFP*. This result is analogous to a theorem of Aczel [1] on systems of positive existential inductive definitions. These are equivalent to *ELFP* formulas containing a single simultaneous inductive definition. Chandra and Harel showed that every *ELFP* formula is equivalent to a formula with just one such inductive definition, so Aczel's result may be viewed as a result about *ELFP* definability. From this perspective, it says that in *existentially acceptable* structures, the *ELFP* definable sets are precisely the Σ_1^0 sets. A structure is existentially acceptable if contains an existentially definable copy of the natural numbers and an existentially definable relation coding all finite sequences of elements. We will show that on existentially acceptable structures the sets definable by *SLFP* sentences corresponding to programs with n strata are the Σ_n^0 sets.

Aczel's result was inspired by a result of Moschovakis [35] stating that on *acceptable structures* the inductively definable sets are the Π_1^1 sets. Acceptable structures (sometimes also called *arithmetical structures* in program verification) are defined in the same way as existentially acceptable structures except that the condition of existential definability is relaxed to first-order definability. We will show that on the acceptable structures the *SLFP* definable sets are the first-order definable sets. This, combined with a result of Blass and Gurevich [11] showing that weakest preconditions and strongest postconditions for a programming language with recursive procedures are expressible in *ELFP*, explains why acceptable structures often arise in program verification (see, e.g., Cook [14] and Harel [23]).

Hoare [24, 25] originally used first-order logic as the assertion language for program verification. Attempts to find a complete Hoare logic for program verification uncovered a variety of problems. Cook [14] found a way around some of these problems by showing that if interpretations (structures on which programs operate) are required to be *expressible* (i.e., strongest postconditions are first-order definable), then Hoare logic is complete for proving partial correctness. Unfortunately, Lipton [32] showed that expressible interpretations are quite restricted. One approach to this difficulty has been to use logics other than first-order logic as the assertion language. In this direction Stavely [41] considered monadic logic with second-order quantification, Back [7, 8] considered $L_{\omega_1\omega}$, and Leivant [31] considered full second-order logic. These have certain theoretical advantages, but are unsuitable for practical program verification. Monadic logic is expressively meager and second-order logic does not have a complete deductive system. Sentences of $L_{\omega_1\omega}$ may not even be recursively enumerable, let alone finite.

It is natural, in light the expressibility result of Blass and Gurevich, to ask if *ELFP* is a reasonable assertion language. *ELFP* seems at first to hold promise as an assertion language since it has a deductive system, although, as with $L_{\omega_1\omega}$, an infinitary one. However, besides the obvious drawbacks of an assertion language with no universal quantification, *ELFP* has a very conspicuous deficiency: program correctness is not a logical property with respect to *ELFP*. By this we mean that a pair of structures may satisfy precisely the same *ELFP* sentences, but still there may be a an asserted program true in one and false in the other. When we consider the modifications needed to rectify this, we discover that we must be able to negate formulas with no free relation variables. This leads directly to *SLFP*. Both partial correctness and total correctness are logical notions with respect to *SLFP*. This demonstrates the superiority of *SLFP* over first-order logic as an assertion language. Partial correctness is a logical notion with respect to first-order logic, but total correctness is not.

The infinitary nature of the deductive system for *SLFP* cannot be avoided. Neither *ELFP* nor *SLFP* is compact (see Compton [13]), so neither has a finitary deductive system. The sequent calculi we present for these logics contain just one infinitary rule. Since they do not contain the *cut rule* (see Takeuti [42]) it will follow that the infinitary rule can sometimes be avoided. As an example, we show rather easily that there is a finitary deductive system for *total correctness* proofs of programs with first-order assertions.

The idea of using infinitary rules for programming logics and, in particular, of embedding the logics in $L_{\omega_1\omega}$, has a long history. Engeler [20, 21] was the first to do this in formulating an extension of first-order logic in which algorithmic properties could be expressed. Salwicki [40] took up and extended these ideas. Infinitary rules for programming logics have been used extensively since that time (see the summaries in Harel [23] and Kozen and Tiuryn [29]). We will suggest ways of dealing with infinitary rules.

2 Description of the Logic.

To describe *SLFP*, let us first look at a standard textbook example: a Datalog query about membership of a pair (c, d) in the reflexive, transitive closure of a binary relation E. (Datalog is pure Prolog with no function symbols.)

```
P(x,y) ← x=y.
P(x,y) ← E(x,y).
P(x,y) ← P(x,z),E(z,y).
-? P(c,d).
```

This program consists of three rules which constitute an inductive definition of a relation P, followed by a query about membership in P. In *ELFP* we would write

$$[P(x,y) \equiv x = y \vee E(x,y) \vee \exists z\,(P(x,z) \wedge E(z,y))]\, P(c,d).$$

The part of the formula within the square brackets is an inductive definition of the relation P. This definition is used in the formula that follows. (Blass and Gurevich use the notation LET $\cdots$ THEN rather than $[\cdots]$.) Notice that an inductive definition binds variables just as a quantifier does, so it goes before the formula.

Now suppose that we wish to make a query as to whether there are at least three components. We would like to add the following to the program above.

```
Q() ← ¬P(x,y),¬P(y,z),¬P(z,x).
-? Q().
```

This would not be allowed in Datalog because negation is forbidden. This restriction avoids problems of convergence in examples such as

```
P(x,y) ← ¬P(x,y).
```

One solution to this problem is to divide the rules defining relations into strata. Within each rule the only symbols that may be negated are those defined in lower strata. Thus, the query about three components would be allowed since the definition of Q may occupy a higher stratum than the definition of P, but the program where P appears negated within its own definition is not allowed. This is the essential idea behind stratified logic programs.

Let us make this precise. Fix a vocabulary V of constant, function, and relation symbols. The symbols in V are the *extensional symbols.* Also fix a set of relation symbols, disjoint from V. These are the *relation variables* or, in database parlance, the *intensional symbols*. Element variables will be specified by lower case letters such as x, y, z, x_1, x_2, while relation variables will be specified by upper case letters such as P and Q. Each relation variable P has a specified arity and we assume that we have a potentially infinite number of relation variables of each arity. We now form *terms* in the usual way using function and constant symbols in V and element variables. We form *atomic formulas* by applying either intensional or extensional relation symbols to tuples of terms, or by equating two terms.

Definition. A *rule* is an expression of the form $P(\vec{x}) \leftarrow \alpha_1, \ldots, \alpha_k$ where P is an intensional symbol, the elements of $\vec{x}$ are distinct, and each α_i is an atomic or negated atomic formula. $P(\vec{x})$ is the *head* of the rule and the formulas α_i form the *body* of the rule.

This definition may appear to be more restrictive than the usual definition in logic programming where arbitrary terms rather distinct variables may occur in the head of a rule. However, since we allow equality, it can be shown that a rule of the form $P(t_1, \ldots, t_j) \leftarrow \alpha_1, \ldots, \alpha_k$ may be replaced with

$$P(x_1, \ldots, x_j) \leftarrow x_1 = t_1, \ldots, x_j = t_j, \alpha_1, \ldots, \alpha_k.$$

Definition. A *general program* is a finite set of rules in which every intensional symbol that occurs appears at least once at the head of a rule.

The *dependency graph* of a general program is a directed graph whose vertices are the relation variables of the program, with (P, Q) as an edge whenever there is a rule in the program with P at the head and Q somewhere in the body. An edge (P, Q) is *negative* if there is a rule in the program with P at the head and Q negated somewhere in the body. P is *dependent* on Q if there is a path from P to Q in the dependency graph and *negatively dependent* if there is a path from P to Q containing a negative edge. A *logic program* is a general program in which no occurrence of an intensional symbol is negated. This is equivalent to saying that the dependency graph contains no negative edges. A *stratified logic program* is a general program such that no cycle of its dependency graph contains a negative edge; i.e., no relation symbol is negatively dependent on itself.

A *query* is a pair $(\mathcal{S}, P(\vec{x}))$ where $\mathcal{S}$ is a logic program, P is intensional symbol occurring in $\mathcal{S}$, and $\vec{x}$ is a sequence of distinct element variables. A *stratified query* is defined in the same way except that $\mathcal{S}$ is a stratified logic program.

We can now give the semantics for a stratified logic program $\mathcal{S}$. The intensional symbols of $\mathcal{S}$ (and thus, using their head symbols, the rules in $\mathcal{S}$) may be stratified (or partitioned into a linearly ordered set of classes) so that a relation variable in a particular stratum depends only on variables in its own or lower strata, and depends negatively only on variables in lower strata. The interpretations of relation variables are then given by the usual least fixpoint construction beginning at the lowest stratum and working upward. Apt, Blair, and Walker [6] show that the resulting interpretations of intensional symbols are independent of the particular stratification used. The stratified query $(\mathcal{S}, P(\vec{x}))$ is interpreted by the set of tuples satisfying $P(\vec{x})$ when P is interpreted according to $\mathcal{S}$.

It is useful to have a *canonical stratification* for a stratified logic program $\mathcal{S}$. Let V_{n+1} contain the intensional symbols P such that in the dependency graph the maximum number of negative edges along any directed path beginning at R is n. It is not difficult to see that the canonical stratification is of minimal size. The *depth* of a stratified query $(\mathcal{S}, P)$ is the number of elements in the canonical stratification of $\mathcal{S}$.

Now let us define the *ELFP* and *SLFP* formulas. As before, we assume that we have a fixed vocabulary V and a set of relation variables.

Definition. The set $\mathcal{F}$ of *ELFP* formulas φ over V is the least set containing the atomic formulas and satisfying the following conditions.

(i) If ψ is a formula in $\mathcal{F}$ containing no relation variables or quantifiers, then $(\neg\psi)$ is in $\mathcal{F}$.

(ii) If ψ and ϑ are in $\mathcal{F}$, so are $(\psi \vee \vartheta)$ and $(\psi \wedge \vartheta)$.

(iii) If ψ is in $\mathcal{F}$ and x is an element variable, then $(\exists x\, \psi)$ is in $\mathcal{F}$.

(iv) If ψ and ϑ are in $\mathcal{F}$, P is a relation variable of arity k, and $\vec{x} = (x_1, \ldots, x_k)$ is a sequence of distinct element variables, then $([P(\vec{x}) \equiv \vartheta]\, \psi)$ is in $\mathcal{F}$. The initial part of the formula, viz., $[P(\vec{x}) \equiv \vartheta]$, is called an *inductive definition.*

We follow the usual conventions for deleting parentheses in formulas.

Definition. For each *ELFP* formula φ define *free*(φ), the set of free variables in φ, and the *free occurrences* of variables in φ. When φ is atomic, *free*(φ) is the set of element and relation variables in φ; all occurrences of variables in φ are free. Free variables in formulas constructed using logical connectives and quantifiers are handled in the usual way. Finally,

$$\mathit{free}([P(\vec{x}) \equiv \vartheta]\, \psi) = \Big((\mathit{free}(\vartheta) - \{x_1, \ldots, x_j\}) \cup \mathit{free}(\psi)\Big) - \{P\}.$$

The free occurrences of variables in $[P(\vec{x}) \equiv \vartheta]\, \psi$ are the free occurrences of variables of $\mathit{free}(\vartheta) - \{P, x_1, \ldots, x_j\}$ in ϑ and the free occurrences of variables from $\mathit{free}(\psi) - \{P\}$ in ψ. As usual, a *sentence* is a formula with no free variables.

Let us now give the analogous definitions for *SLFP*. Strictly speaking, the notions of formula and free variable should be defined by simultaneous induction.

Definition. Inductively define the set $\mathcal{F}$ of *SLFP* formulas by making two modifications in the definition of *ELFP* formulas above. First, condition (i) is replaced with the following.

(i′) If ψ is a formula in $\mathcal{F}$ containing no free relation variables, then $(\neg\psi)$ is in $\mathcal{F}$.

In addition, it is convenient (though it does not increase expressive power) for formulas to contain universal quantifiers. We add the following condition.

(v′) If ψ is a formula in $\mathcal{F}$ containing no free relation symbols and x is an element variable, then $(\forall x\, \psi)$ is in $\mathcal{F}$.

To define the notion of a free variable and a free occurrence of a variable in an *SLFP* formula, add the obvious condition to cover universal quantification.

When we write $\varphi(x/t)$ we mean that term t has been substituted for all free occurrences of the element variable x in φ. All uses of this notation are subject to the proviso that occurrences of variables in t be free wherever t is substituted. In the case where t is just a single variable y we often write $\varphi(y)$ rather than $\varphi(x/y)$. The notation $\neg\varphi$ is defined only when φ contains no free relation variables. The notation $\varphi(P/\rho)$ means that all subformulas of φ containing free occurrences of the relation variable P are replaced by formula ρ. (To be precise, we should specify a sequence of k distinct element variables in ρ, where k is the arity of P; the correspondence between element variables of P and element variables of ρ will always be clear from context.) All uses of this notation are subject to the proviso that free occurrences of variables in ρ remain free wherever ρ is substituted.

We now give the semantics of *ELFP* and *SLFP* formulas. As usual, we define by induction on φ the relation $\mathfrak{A} \models \varphi[\alpha]$ ($\mathfrak{A}$ *satisfies* φ *at* α), where α is an assignment in $\mathfrak{A}$. More precisely, suppose φ has free relation variables $P_1, \ldots, P_k$, with respective arities $j_1, \ldots, j_k$, and free element variables $x_1, x_2, \ldots, x_l$. Fix a structure $\mathfrak{A}$. An assignment α for φ can be represented as a sequence $(R_1, R_2, \ldots, R_k, a_1, \ldots, a_l)$, where each R_i is a j_i-ary relation on $\mathfrak{A}$ and each a_i is an element of $\mathfrak{A}$. With φ we will associate a function F_φ mapping sequences $(R_1, R_2, \ldots, R_k)$ to l-ary relations on $\mathfrak{A}$:

$$F_\varphi(R_1, R_2, \ldots, R_k) = \{(a_1, \ldots, a_l) \mid \mathfrak{A} \models \varphi[R_1, R_2, \ldots, R_k, a_1, \ldots, a_l]\}.$$

Simultaneously with our definition of satisfaction, we also show that F_φ is *continuous*; i.e., that

$$\bigcup_{\alpha<\lambda} F_\varphi(R_{1\alpha}, \ldots, R_{k\alpha}) = F_\varphi(\bigcup_{\alpha<\lambda} R_{1\alpha}, \ldots, \bigcup_{\alpha<\lambda} R_{k\alpha})$$

for all chains $(R_{i\alpha} \mid \alpha < \lambda)$ of j_i-ary relations. Notice that if F_φ is continuous, it is *monotone* as well; i.e., $F_\varphi(R_1, \ldots, R_k) \subseteq F_\varphi(R'_1, \ldots, R'_k)$ whenever $R_1 \subseteq R'_1, \ldots, R_k \subseteq R'_k$. By a continuous (or monotone) formula, we mean a formula φ such that F_φ is continuous (or monotone).

If φ is atomic, $\mathfrak{A} \models \varphi[\alpha]$ is defined in the usual way and F_φ is clearly continuous. Also, if φ is a disjunction, conjunction, negation, or quantified formula, $\mathfrak{A} \models \varphi[\alpha]$ is again defined in the usual way, and it is not difficult to see that φ is continuous. (Notice, however, that it is crucial that negations are not applied to formulas with free relation variables.)

Let us define $\mathfrak{A} \models \varphi[\alpha]$ when φ is of the form $[P(\vec{x}) \equiv \vartheta]\,\psi$ assuming that ϑ and ψ are continuous and their truth values have been defined for the assignment α. Let the assignments to variables other than P and $\vec{x} = (x_1, \ldots, x_k)$ be given by the assignment α. We thereby obtain from F_ϑ a continuous mapping G from k-ary relations to k-ary relations. G is monotone and hence has a least fixpoint by the Least Fixpoint Theorem (or at least by one of the theorems that go by this name; see Lassez, Nguyen, and Sonenberg [30]).

The well known construction of the least fixpoint of a monotone function is as follows. Let $G^0(R) = R$, $G^{\beta+1}(R) = G(G^\beta(R))$ and if β is a limit ordinal, $G^\beta(R) = \bigcup_{\gamma<\beta} G^\gamma(R)$. By induction $G^\beta(\emptyset) \subseteq G^\gamma(\emptyset)$ whenever $\beta < \gamma$. On each structure there is a smallest ordinal κ (called the *closure ordinal* of the inductive definition $[P(\vec{x}) \equiv \vartheta]$) such that $G^\beta(\emptyset) = G^\kappa(\emptyset)$ whenever $\beta \geq \kappa$. $G^\kappa(\emptyset)$ is the least fixpoint of G. Since G is continuous, it follows that $\kappa \leq \omega$ (see [30]). Thus, $\mathfrak{A} \models \varphi[\alpha]$ holds in case $\mathfrak{A} \models \psi[\alpha']$, where α' is identical to α except that it assigns $G^\omega(\emptyset)$ to P.

It will be useful to define, for each nonnegative integer m, the formula

$$[P(\vec{x}) \equiv \vartheta]_m\,\psi.$$

$\mathfrak{A} \models [P(\vec{x}) \equiv \vartheta]_m \psi[\alpha]$ holds just in case $\mathfrak{A} \models \psi[\alpha'']$, where α'' is identical to α except that it assigns $G^m(\emptyset)$ to P. We regard this formula as an abbreviation. Construct a sequence of formulas $\rho_0, \rho_1, \rho_2, \ldots$, where ρ_0 is the formula $\exists x\,(\neg x{=}x)$ and ρ_{m+1} is the formula $\vartheta(P/\rho_m)$. Then $[P(\vec{x}) \equiv \vartheta]_m \psi$ is an abbreviation for $\psi(P/\rho_m)$. Call this formula φ_m.

Since continuity is preserved by composition, each of the functions F_{φ_m} is continuous. Moreover, the sequence $F_{\varphi_0}, F_{\varphi_1}, F_{\varphi_2}, \ldots$ is a chain (in the partial order of function dominance) with supremum F_φ. Since the supremum of a chain of continuous functions is continuous, F_φ is continuous. (See Theorem 4.18 of Loeckx and Sieber [33].) It follows that $[P(\vec{x}) \equiv \vartheta]\,\psi$ is equivalent to the infinite disjunction

$$\bigvee_{m\in\omega} [P(\vec{x}) \equiv \vartheta]_m \psi.$$

We summarize our observations in the following theorem.

Theorem 2.1 *The following hold for* ELFP *and* SLFP.

(i) All formulas are continuous (and hence monotone).

(ii) The closure ordinal of any inductive definition is at most ω.

(iii) $[P(\vec{x}) \equiv \vartheta]\,\psi$ is equivalent to $\bigvee_{m\in\omega}[P(\vec{x}) \equiv \vartheta]_m \psi$. Thus every sentence is equivalent to a sentence of $L_{\omega_1\omega}$.

This theorem is due to Park [38]. Part (ii) of this theorem was observed by Aczel [1] for systems of existential inductive definitions. Blass and Gurevich [11] observed that (ii) is true for *ELFP* formulas.

In practice we extend the definitions of *ELFP* and *SLFP* to cover simultaneous inductive definitions, as Blass and Gurevich did in their definition of *ELFP*. By this we mean that rather than a single relation variable P and formula ϑ, we allow multiple relation variables and formulas in inductive definitions. Thus, we allow formulas of the form

$$[P_1(\vec{x}_1) \equiv \vartheta_1; \cdots; P_k(\vec{x}_k) \equiv \vartheta_k]\,\psi$$

where we make the obvious modifications to define several relations simultaneously. This does not change the expressive power of the logic, nor any of results above. A formula with simultaneous inductive definitions may always be transformed into an equivalent formula with only simple inductive definitions. This was first proved by Chandra and Harel [12]; their proof was based on a similar result of Moschovakis [35] for inductive definitions. Exactly the same construction works for *SLFP*. We also define the formula

$$[P_1(\vec{x}_1) \equiv \vartheta_1; \cdots; P_k(\vec{x}_k) \equiv \vartheta_k]_m\,\psi$$

analogously to the formula $[P(\vec{x}) \equiv \vartheta]_m \psi$.

Definition. The *negation rank* of an *SLFP* formula is defined as follows. The negation rank of a formula containing no quantifiers or inductive definitions is 1. The negation rank of $\varphi \vee \psi$ and $\varphi \wedge \psi$ is the maximum of the negation ranks of φ and ψ. The negation rank of

$$[P_1(\vec{x}_1) \equiv \vartheta_1; \cdots; P_k(\vec{x}_k) \equiv \vartheta_k]\,\psi$$

is the maximum of the negation ranks of $\vartheta_1, \ldots, \vartheta_k$ and ψ. The negation rank of $\exists x \, \varphi$ is the negation rank of φ. If φ contains a quantifier or inductive definition, the negation rank of $\neg\varphi$ is one more than the negation rank of φ.

Now we show that *SLFP* formulas have the same expressibility as stratified queries.

Theorem 2.2 *Let n be a positive integer. For every stratified query $(\mathcal{S}, P)$ of depth n, there is an equivalent* SLFP *formula ψ of negation rank n. Conversely, for every* SLFP *formula of negation rank n without free relation variables, there is an equivalent stratified query of depth n.*

Proof. The first half of the theorem is proved by induction on n. The base case $n = 1$ is easy. It was essentially proved by Chandra and Harel [12].

Suppose that $(\mathcal{S}, P(\vec{x}))$ is a stratified query of depth n and that the theorem is true for all stratified queries of smaller depth. We we must produce an *SLFP* formula $\psi_P(\vec{x})$ of negation rank at most n equivalent to $(\mathcal{S}, P(\vec{x}))$.

Let $V_1, \ldots, V_n$ be the canonical stratification of the intensional symbols in $\mathcal{S}$ and $\mathcal{S}_i$ be the set of rules in $\mathcal{S}$ whose heads are in V_i. Notice that $\mathcal{S}' = \mathcal{S}_1 \cup \cdots \cup \mathcal{S}_{n-1}$ is a stratified logic program of depth $n-1$ and that any relation symbol in $V' = V_1 \cup \cdots \cup V_{n-1}$ has the same interpretation in $\mathcal{S}'$ as in $\mathcal{S}$. Thus, if P is in V', we know by the induction hypothesis that there is an *SLFP* formula ψ_P equivalent to the query $(\mathcal{S}', P)$. Note that ψ_P has no free relation variables and has depth at most $n-1$.

Now consider the case where P is in V_n. Let $P_1, \ldots, P_k$ be the relation variables in V_n where P is P_1, say. Without loss of generality we may suppose that the heads of all rules where P_i appears are of the form $P_i(\vec{x}_i)$ for fixed sequences of variables $\vec{x}_i$. Consider one such rule. It has a sequence of atomic and negated atomic formulas in its body. In each formula of the body that mentions a relation symbol Q from V', replace Q with the formula ψ_Q described in the previous paragraph, then take the conjunction of the resulting sequence of formulas and existentially quantify all element variables not appearing in $\vec{x}_i$. For each rule with P_i at the head this gives an *SLFP* formula. Its negation rank is at most n since we have applied negation at most once to formulas of depth at most $n-1$. The free relation variables in each such formula are included in $P_1, \ldots, P_k$. Now take the disjunction of all such formulas over rules with P_i at the head to form a formula ϑ_i of depth at most n. The formula

$$[P_1(\vec{x}_1) \equiv \vartheta_1; \cdots; P_k(\vec{x}_k) \equiv \vartheta_k] \, P_1(\vec{x}_1)$$

is of depth at most n and is equivalent to $P(\vec{x})$.

Now we need to prove the other half of the theorem. We show by induction on formula complexity that every *SLFP* formula of negation rank at most n with free relation variables in V' is equivalent to a stratified query of depth at most n with extensional symbols in $V \cup V'$ and with no intensional symbol negatively dependent on a relation symbol in V'. This is clear for atomic formulas.

Let ψ_1 and ψ_2 be *SLFP* formulas equivalent to stratified queries $(\mathcal{S}_1, P_1(\vec{x}_1))$ and $(\mathcal{S}_2, P_2(\vec{x}_2))$ respectively. We consider the various operations for building *SLFP* formulas from ψ_1 and ψ_2.

The formula $\psi_1 \vee \psi_2$ is equivalent to $(\mathcal{S}, P(\vec{x}))$, where $\vec{x}$ contains all the variables in $\vec{x}_1$ and $\vec{x}_2$, and $\mathcal{S}$ contains all the rules in $\mathcal{S}_1$ and $\mathcal{S}_2$ and, in addition, the rules $P(\vec{x}) \leftarrow P_1(\vec{x}_1)$ and $P(\vec{x}) \leftarrow P_2(\vec{x}_2)$. We suppose here that $\mathcal{S}_1$ and $\mathcal{S}_2$ have disjoint sets of intensional variables. To obtain a stratified program for $\psi_1 \wedge \psi_2$ we do the same thing except that we instead add the rule $P(\vec{x}) \leftarrow P_1(\vec{x}_1), P_2(\vec{x}_2)$. Notice that in both these cases the depth of $\mathcal{S}$ is the maximum of the depths of $\mathcal{S}_1$ and $\mathcal{S}_2$. Also, no new negative dependencies arise in constructing $\mathcal{S}$.

Formula $\exists y\, \psi_1$ is equivalent to $(\mathcal{S}, P(\vec{x}))$, where $\vec{x}$ contains all the variables in $\vec{x}_1$ except y and $\mathcal{S}$ contains the rules in $\mathcal{S}_1$ and the rule $P(\vec{x}) \leftarrow P_1(\vec{x}_1)$. Here depth is unchanged and no new negative deper dencies arise in the construction of $\mathcal{S}$.

Now $\neg\psi_1$ is defined only if ψ_1 has no free relation variables. Thus, it is equivalent to $(\mathcal{S}, P(\vec{x}_1))$, where $\mathcal{S}$ contains the rules in $\mathcal{S}_1$ and the rule $P(\vec{x}) \leftarrow \neg P_1(\vec{x}_1)$. Here depth increases by one, as does the negation rank of the formula. Also, no intensional symbol in $\mathcal{S}$ can be negatively dependent on a free relation variable in $\neg\psi_1$ because there are none.

Finally, consider the formula $[P(\vec{x}) \equiv \psi_1]\, \psi_2$. To avoid trivialities we may suppose that P is free in both ψ_1 and ψ_2. We would like to do something like the following. By the induction hypothesis, ψ_1 and ψ_2 are equivalent to the stratified queries $(\mathcal{S}_1, P_1(\vec{x}_1))$ and $(\mathcal{S}_2, P_2(\vec{x}_2))$, respectively, where the extensional variables in $\mathcal{S}_i$ not in V are free in ψ_i. P is an extensional variable in both $\mathcal{S}_1$ and $\mathcal{S}_2$. Form $\mathcal{S}$ by taking the union of $\mathcal{S}_1$ and $\mathcal{S}_2$ and adding the rule $P(\vec{x}) \leftarrow P_1(\vec{x}_1)$. The problem with this is that there may be variables in $\vec{x}_1$ not in $\vec{x}$. These extra variables are like global variables (in an imperative programming language) whose values may affect the return value of a procedure P even though they are not parameters of P. Logic programs do not have global variables: as we have seen, extra variables are existentially quantified.

We use a standard technique for eliminating global variables. Let $\vec{y}$ be the sequence of variables of $\vec{x}_1$ not in $\vec{x}$. Increase the arity of P by an amount equal to the length of $\vec{y}$ and add the variables $\vec{y}$ to the variable list of each occurrence of P in ψ_1 and ψ_2. (It may be necessary to change some of the bound variables in ψ_1 and ψ_2 to avoid conflicts.) We thereby obtain formulas ψ_1' and ψ_2' having the same negation ranks as ψ_1 and ψ_2. By the the induction hypothesis, there are stratified queries $(\mathcal{S}_1', P_1'(\vec{x}_1))$ and $(\mathcal{S}_2', P_2'(\vec{x}_2))$ equivalent to ψ_1' and ψ_2'. We may suppose that $\mathcal{S}_1'$ and $\mathcal{S}_2'$ have disjoint sets of intensional variables. Also, by the induction hypothesis, we suppose that in $\mathcal{S}_1'$ symbol P_1' is not negatively dependent on P.

We now form $\mathcal{S}$ by taking the union of $\mathcal{S}_1'$ and $\mathcal{S}_2'$ and adding the rule $P(\vec{x}, \vec{y}) \leftarrow P_1(\vec{x}_1)$. What we have done, in effect, is add new parameters $\vec{y}$ to the definition of P (in the theory of programming languages, these are the *formal* parameters), and also new parameters (the *actual* parameters) wherever P is invoked. It should be clear that $(\mathcal{S}, P_2)$ is equivalent to $[P(\vec{x}) \equiv \psi_1]\, \psi_2$ and that no intensional symbol in $\mathcal{S}$ is negatively dependent on any free variable in $[P(\vec{x}) \equiv \psi_1]\, \psi_2$. □

3 A Deductive System for *SLFP*.

We now present a sequent calculus, which we call LS, for *SLFP*. The rules of this calculus are not difficult to formulate once we have Theorem 2.1 showing that *SLFP* formulas may be easily translated into $L_{\omega_1\omega}$ formulas. We need only make suitable modifications of a deductive system for $L_{\omega_1\omega}$. C. Karp [27] was the first to prove the completeness of a deductive system for $L_{\omega_1\omega}$. Our system is based on a sequent calculus for $L_{\omega_1\omega}$ due to Lopez-Escobar [34]. One notable feature of this calculus is a proof rule for equality that circumvents some of the usual problems with equality in cut-free sequent calculi. Lopez-Escobar attributes this rule to Maehara and Takeuti.

We observe the following conventions. Lower case Greek letters denote *SLFP* formulas. Upper case Greek letters denote *sets* of *SLFP* formulas. Γ, Δ denotes $\Gamma \cup \Delta$. Γ, φ denotes $\Gamma \cup \{\varphi\}$. A *sequent* is an expression of the form $\Gamma \vdash \Delta$. In general, a formula φ occurring as part of a sequent denotes the set $\{\varphi\}$. Finally, $t_1 \doteq t_2$ indicates that either $t_1 = t_2$ or $t_2 = t_1$ may be used.

We may regard the rules of the calculus as inductively defining a binary relation $\vdash$ holding between sets of *SLFP* formulas: the *lower sequent* (located below the line) holds if the *upper sequents* (above the line) hold. The *axioms* of the calculus are the base cases for the induction. Gentzen's sequent calculus **LK** used *sequences* of formulas rather than sets. By working with sets we may ignore two of the so-called "weak" rules of inference, viz. the rules of contraction and exchange. (See Takeuti [42].)

The rule ([ ] $\vdash$) in our calculus is infinitary: it has countably many upper sequents. In Compton [13] it is shown that *ELFP* is not compact, so *SLFP* is also not compact. It follows that we must have some sort of infinitary rule in any complete sequent calculus for *SLFP*.

As usual, $\Gamma \models \varphi$ will mean that every model of Γ satisfies φ and $\Gamma \models \Delta$ will mean that every model of Γ satisfies some formula in Δ. (When Δ is empty, this is interpreted to mean that Γ has no models).

Definition. The *axioms* of **LS** are sequents of the form $\varphi \vdash \varphi$, where φ is a formula of *ELFP*, and $\emptyset \vdash t{=}t$, where t is a term.

Definition. The *rules* for **LS** are as follows.

$$(* \vdash) \quad \frac{\Gamma \vdash \Delta}{\Gamma, \Sigma \vdash \Delta} \qquad\qquad (\vdash *) \quad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \Sigma}$$

$$(S \vdash) \quad \frac{\Gamma, \varphi(x/t_1) \vdash \Delta}{\Gamma, t_1 \doteq t_2, \varphi(x/t_2) \vdash \Delta} \qquad\qquad (\vdash S) \quad \frac{\Gamma \vdash \Delta, \varphi(x/t_1)}{\Gamma, t_1 \doteq t_2 \vdash \Delta, \varphi(x/t_2)}$$

$$(\neg \vdash) \quad \frac{\Gamma \vdash \Delta, \psi}{\Gamma, \neg\psi \vdash \Delta} \qquad\qquad (\vdash \neg) \quad \frac{\Gamma, \psi \vdash \Delta}{\Gamma \vdash \Delta, \neg\psi}$$

$$(\vee \vdash) \quad \frac{\Gamma, \psi \vdash \Delta \quad \Gamma, \vartheta \vdash \Delta}{\Gamma, \psi \vee \vartheta \vdash \Delta} \qquad\qquad (\vdash \vee) \quad \frac{\Gamma \vdash \Delta, \psi, \vartheta}{\Gamma \vdash \Delta, \psi \vee \vartheta}$$

$$(\wedge \vdash) \quad \frac{\Gamma, \psi, \vartheta \vdash \Delta}{\Gamma, \psi \wedge \vartheta \vdash \Delta} \qquad\qquad (\vdash \wedge) \quad \frac{\Gamma \vdash \Delta, \psi \quad \Gamma \vdash \Delta, \vartheta}{\Gamma \vdash \Delta, \psi \wedge \vartheta}$$

$$(\exists \vdash) \quad \frac{\Gamma, \psi(x) \vdash \Delta}{\Gamma, \exists y\, \psi(y) \vdash \Delta} \quad x \notin \mathit{free}(\Gamma \cup \Delta) \qquad (\vdash \exists) \quad \frac{\Gamma \vdash \Delta, \psi(x/t)}{\Gamma \vdash \Delta, \exists y\, \psi(y)}$$

$$(\forall \vdash) \quad \frac{\Gamma, \psi(x/t) \vdash \Delta}{\Gamma, \forall y\, \psi(y) \vdash \Delta} \qquad\qquad (\vdash \forall) \quad \frac{\Gamma \vdash \Delta, \psi(x)}{\Gamma \vdash \Delta, \forall y\, \psi(y)} \quad x \notin \mathit{free}(\Gamma \cup \Delta)$$

$$([\,]\vdash)\ \frac{\Gamma, [P(\vec{x}) \equiv \vartheta]_m \psi \vdash \Delta \quad (m \in \omega)}{\Gamma, [P(\vec{x}) \equiv \vartheta]\, \psi \vdash \Delta} \qquad (\vdash [\,])\ \frac{\Gamma \vdash \Delta, [P(\vec{x}) \equiv \vartheta]_m \psi}{\Gamma \vdash \Delta, [P(\vec{x}) \equiv \vartheta]\, \psi}$$

Rules $(* \vdash)$ and $(\vdash *)$ are, respectively, the left and right *weakening rules.* Rules $(S \vdash)$ and $(\vdash S)$ are the left and right *substitution rules.* The other rules introduce the various operations on formulas on the left and right sides of sequents. Notice that in the rule $(\vdash [\,])$ there is just one upper sequent: m is a fixed nonnegative integer. We have stated the rules $([\,] \vdash)$ and $(\vdash [\,])$ for formulas with simple inductive definitions, but we intend the rules to apply also to formulas with simultaneous inductive definitions.

Definition. The sequent calculus **LE** for *ELFP* is defined exactly as above except that the rules $(\forall \vdash)$ and $(\vdash \forall)$ are deleted. The set of *theorems* of **LS** is the least set of *SLFP* sequents containing the axioms and closed under the rules of inference of **LS** and similarly for **LE.**

We have not included the familiar *cut rule*

$$\frac{\Gamma, \varphi \vdash \Delta \quad \Gamma \vdash \Delta, \varphi}{\Gamma \vdash \Delta}$$

in either **LE** or **LS**. In Compton [13], we prove completeness of **LE** without the cut rule. The same proof works for **LS**. We have the following results.

Theorem 3.1 (Soundness and Completeness Theorem for LE and LS)

(i) Suppose Γ *and* Δ *are sets of* ELFP *sentences. Then* $\Gamma \vdash \Delta$ *is a theorem of* LE *if and only if* $\Gamma \models \Delta$.

(ii) Suppose Γ *and* Δ *are sets of* SLFP *sentences. Then* $\Gamma \vdash \Delta$ *is a theorem of* LS *if and only if* $\Gamma \models \Delta$.

4 A Logic for Program Verification?

In this section we will look at verification of programs written in an imperative language. The term *program* will no longer mean logic program or general program as it did in earlier sections. Let us first review some of the basics of program verification.

The state of a program is represented by an assignment α of elements from a structure $\mathfrak{A}$ to the program variables. In the literature of program verification, an assignment is called a *state* and a structure is called an *interpretation.* Program execution changes the state: it assigns new values to the program variables. A logic such as first-order logic or *ELFP* serves as an *assertion language*; it is used to make assertions about states. Verification is a matter of showing that if certain assertions hold of the initial state a program, then other assertions hold of the final state.

Definition. Let $\varphi(\vec{x})$ and $\psi(\vec{x})$ be formulas and S be a program with variables $\vec{x}$. An *asserted program* is an expression of the form $\{\varphi\}\,S\,\{\psi\}$.

By $\mathfrak{A} \models \{\varphi\}\,S\,\{\psi\}$ we mean that if S begins in state α, $\mathfrak{A} \models \varphi[\alpha]$, and S halts in state β, then $\mathfrak{A} \models \psi[\beta]$. Notice that if S does not halt, then $\mathfrak{A} \models \{\varphi\}\,S\,\{\psi\}$ is true by default. This defines the notion of *partial correctness* of an asserted program in an interpretation $\mathfrak{A}$. By $\models \{\varphi\}\,S\,\{\psi\}$ we mean that $\mathfrak{A} \models \{\varphi\}\,S\,\{\psi\}$ holds for every $\mathfrak{A}$.

By $\mathfrak{A} \mid\!\models \{\varphi\}\,S\,\{\psi\}$ we mean that if S begins in a state α and $\mathfrak{A} \models \varphi[\alpha]$, then S halts in a state β such that $\mathfrak{A} \models \psi[\beta]$. This defines the notion of *total correctness* of an asserted program in an interpretation $\mathfrak{A}$. By $\mid\!\models \{\varphi\}\,S\,\{\psi\}$ we mean that $\mathfrak{A} \mid\!\models \{\varphi\}\,S\,\{\psi\}$ holds for every $\mathfrak{A}$.

Fix an ordering of the variables of S so that a state may be represented as a sequence $\vec{a}$ of elements from $\mathfrak{A}$. The *state transformer* of a program S under an interpretation $\mathfrak{A}$ is the relation consisting of all pairs $(\vec{a}, \vec{b})$ where S halts in state $\vec{b}$ whenever it begins in state $\vec{a}$. Blass and Gurevich [11] showed that the state transformers for programs written in a while-language with recursive procedures can be defined in *ELFP*. That is, for every program S, there is an *ELFP* formula $\tau_S(\vec{x}, \vec{y})$ that defines the state transformer of S on every interpretation. *ELFP* is well suited for defining the state transformers of programming languages with continuous semantics, including languages with recursive procedures, and even some nondeterministic, parallel, and distributed languages. Fixpoint constructions are fundamental in defining the semantics of these languages, and in most cases *ELFP* suffices to describe these constructions. We note, however, that the example given by Clarke [19] of a programming language whose halting problem is undecidable on finite interpretations is not amenable to this approach.

We illustrate these ideas using the simple while-language in section 2 of Apt [3]. A *program* consists either of a single assignment statement $x_i{:=}t$, where t is a term, or is built from simpler programs according to the following rules.

(i) If S and T are programs, then so is $S; T$.

(ii) If β is a first-order quantifier free formula and S and T are programs, then so is **if** β **then** S **else** T **fi**.

(iii) If β is a first-order quantifier free formula and S is a program, then so is **while** β **do** S **od**.

The results in the remainder of this section will be respect to this programming language, but there is no difficulty in extending to more general languages for which Hoare logics have been worked out.

It is a simple matter to define the state transformers $\tau_S(x_1, \ldots, x_k, y_1, \ldots, y_k)$ for programs in this language. The state transformer for $x_i{:=}t$ is

$$\bigwedge_{j \neq i} x_j = y_j \wedge y_i = t.$$

The state transformer for $S; T$ is

$$\exists \vec{z}(\tau_S(\vec{x}, \vec{z}) \wedge \tau_T(\vec{z}, \vec{y})).$$

The state transformer for **if** $\beta(\vec{x})$ **then** S **else** T **fi** is

$$(\beta(\vec{x}) \wedge \tau_S(\vec{x}, \vec{y})) \vee (\neg\beta(\vec{x}) \wedge \tau_T(\vec{x}, \vec{y})).$$

Finally, the state transformer of while β do $\mathcal{S}$ od is

$$\neg\beta(\vec{y}) \wedge [P(\vec{z}) \equiv (\vec{x} = \vec{z}) \vee \exists\vec{w}\,(\beta(\vec{w}) \wedge P(\vec{w}) \wedge \tau_S(\vec{w},\vec{z}))]\, P(\vec{y}).$$

Here the inductively defined relation P "collects" the states the program is in whenever the Boolean expression β is evaluated. Notice the similarity to the logic program for computing reflexive, transitive closure at the beginning of section 2. Notice also that if Boolean expressions in if-statements and while-statements were allowed to contain quantifiers then it would be necessary to use *SLFP* to express the state transformer.

It is well known that partial and total correctness may be expressed in terms of state transformers (see Tucker and Zucker [43]).

The statement $\mathfrak{A} \models \{\varphi\}\,\mathcal{S}\,\{\psi\}$ is equivalent to the statement that if $\mathfrak{A} \models \exists\vec{x}\,(\varphi(\vec{x}) \wedge \tau_S(\vec{x},\vec{b}))$, then $\mathfrak{A} \models \psi(\vec{b})$. It follows by the Completeness Theorem for **LE** (or **LS**) that whenever φ and ψ are *ELFP* (or *SLFP*) sentences then $\models \{\varphi\}\,\mathcal{S}\,\{\psi\}$ is equivalent to

$$\exists\vec{x}\,(\varphi(\vec{x}) \wedge \tau_S(\vec{x},\vec{y})) \vdash \psi(\vec{y}).$$

The formula $\exists\vec{x}\,(\varphi(\vec{x}) \wedge \tau_S(\vec{x},\vec{y}))$ is called the *strongest postcondition* of φ and $\mathcal{S}$.

Similarly, the statement $\mathfrak{A} \mathrel{|}\!\models \{\varphi\}\,\mathcal{S}\,\{\psi\}$ is equivalent to the statement that if $\mathfrak{A} \models \varphi(\vec{a})$, then $\mathfrak{A} \models \exists\vec{y}\,(\psi(\vec{y}) \wedge \tau_S(\vec{a},\vec{y}))$. Again by the Completeness Theorem for **LE** (or **LS**), whenever φ and ψ are *ELFP* (or *SLFP*) sentences then $\mathrel{|}\!\models \{\varphi\}\,\mathcal{S}\,\{\psi\}$ is equivalent to

$$\varphi(\vec{x}) \vdash \exists\vec{y}\,(\psi(\vec{y}) \wedge \tau_S(\vec{x},\vec{y})).$$

The formula $\exists\vec{y}\,(\psi(\vec{y}) \wedge \tau_S(\vec{x},\vec{y}))$ is called the *weakest precondition* of ψ and $\mathcal{S}$.

It might seem that we do not need *SLFP* because we can verify programs by translating partial or total correctness statements into *ELFP*. We encounter difficulties if we do this. First, it is quite likely that we would want to make assertions containing universal quantifiers. Also, we are usually interested in verification for a particular interpretation, such as the natural numbers, or for a restricted class of interpretations. In the case of a particular interpretation $\mathfrak{A}$ it is customary to take $Th(\mathfrak{A})$, the set of sentences of the assertion language true in $\mathfrak{A}$, as given. For partial correctness, then, we would want to establish something like

$$Th(\mathfrak{A}), \exists\vec{x}\,(\varphi(\vec{x}) \wedge \tau_S(\vec{x},\vec{y})) \vdash \psi(\vec{y}).$$

The problem with this is that partial correctness is not a logical notion with respect to *ELFP*. By this we mean that two interpretations $\mathfrak{A}$ and $\mathfrak{B}$ may satisfy precisely the same *ELFP* sentences, but differ as to partial correctness of some asserted program.

Here is a simple example. Let $\mathfrak{A}$ be the set of rational numbers in the open interval $(0,1)$ with the usual order. Let $\mathfrak{B}$ be the set of rational numbers in the closed interval $[0,1]$ with the usual order. Note that $\mathfrak{A}$ and $\mathfrak{B}$ embed into each other. Blass and Gurevich [11] showed that *ELFP* sentences are preserved by embeddings, so it follows that $\mathfrak{A}$ and $\mathfrak{B}$ satisfy the same *ELFP* sentences. Let φ be the formula $x = x$, ψ be the formula $\exists y\,(y < x)$, and $\mathcal{S}$ be the program $x{:=}x$. Then $\mathfrak{A} \models \{\varphi\}\,\mathcal{S}\,\{\psi\}$ but this is not the case for $\mathfrak{B}$.

One solution to this problem is to modify the definition of proof for **LE** to allow the introduction of *sequents* true in $\mathfrak{A}$. Unfortunately, for practical applications this will not work. We should not expect to have all sequents true in $\mathfrak{A}$ at our disposal. More realistically, we would have a small set of sequents from which other sequents might be derived. The difficulty is that we would have to add the cut rule to **LE** to have a complete deductive system. From a proof theoretic point of view this is undesirable. In some cases we might then be able to prove a cut elimination theorem, but in general this is difficult.

A better approach is to extend the assertion language. It is reasonable to suppose that assertions contain no free relation variables. Now $\mathfrak{A} \models \{\varphi\}\, S\, \{\psi\}$ is equivalent to

$$\mathfrak{A} \models \neg(\exists \vec{x}\,(\varphi(\vec{x}) \wedge \tau_S(\vec{x}, \vec{y}))) \vee \psi(\vec{y})$$

and $\mathfrak{A} \mid\!\models \{\varphi\}\, S\, \{\psi\}$ is equivalent to

$$\mathfrak{A} \models \neg\varphi(\vec{x}) \vee \exists \vec{y}\,(\psi(\vec{y} \wedge \tau_S(\vec{x}, \vec{y})).$$

We see that we need to be able to negate formulas without free relation variables. In other words, we need *SLFP*. From our discussion we have the following result.

Proposition 4.1 *Partial correctness and total correctness are logical notions with respect to* SLFP, *provided that assertions contain no free relation variables.*

Partial and total correctness of asserted programs can be proved by translating to *SLFP*. Of course, we then have the problem of dealing with an infinitary rule. We will say more about this in the next section.

It is interesting to contrast Proposition 4.1 with the situation for first-order logic as an assertion language. Partial correctness is a logical notion with respect to first-order logic (see Lemma 8.7 of Loeckx and Sieber [33]), but total correctness is not, as this example from the proof of Theorem 3 in Apt [3] shows. Let φ and ψ be tautologies and S be the program **while** $x > 0$ **do** $x := x - 1$ **od**. We are using $x - 1$ as a notation for predecessor of x. Then $\mathfrak{A} \mid\!\models \{\varphi\}\, S\, \{\psi\}$, where $\mathfrak{A}$ is the standard model of Peano arithmetic, but this is not the case for any nonstandard model elementarily equivalent to $\mathfrak{A}$.

The difficulties researchers have encountered with total correctness arise precisely because total correctness is not a logical notion with respect to first-order logic. We believe that this is a strong argument for *SLFP* as an assertion language. The the most widely accepted approach to total correctness when first-order logic is the assertion language is due to Harel [22]. It assumes that we work over a class of structures in which the natural numbers are first-order definable. The proof rule for while-statements then takes advantage of the well-foundedness of the natural numbers. This may seem to be a reasonable approach, especially since it does not introduce an infinitary rule, but as we shall see in the next section *SLFP* has the same expressibility as first-order logic on acceptable structures, which, by definition, are structures on which the natural numbers and finite sequences are first-order definable. Thus, if we make a similar restriction to Harel's, we can dispense with the infinitary rule in **LS**.

The lack of a cut rule in **LS** has a rather surprising consequence in the classical total correctness framework where first-order logic is the assertion language. We show that total correctness over all interpretations can be proved in a finitary deductive system.

Theorem 4.2 *Let φ and ψ be first-order formulas and S be a program whose state transformer is expressible in* ELFP. *Then $\mid\!\models \{\varphi\}\, S\, \{\psi\}$ if and only if*

$$\varphi(\vec{x}) \vdash \exists \vec{y}\,(\psi(\vec{y}) \wedge \tau_S(\vec{x}, \vec{y}))$$

can be proved in **LS** *without the infinitary rule* $([\,]\vdash)$.

Proof. The forward direction is a direct consequence of having no cut rule in **LS**. Since φ is a first-order formula it contains no inductive definitions. Since τ_S is an *ELFP* formula, it contains no inductive definition within the scope of a negation. Thus, in the **LS** proof we do not use ([ ] $\vdash$). The converse direction is immediate. □

This theorem may appear to be good news, but in fact is shows how little can be said about total correctness over all interpretations. Monotonicity is the only property of inductive definitions used. This theorem may be viewed as a generalization of an early theorem of Engeler [20] showing that total correctness of simple while-programs can be determined by bounding the number of loop iterations in a program. This is essentially what the rule ($\vdash$ [ ]) does.

We close this section with a results on Hoare logic for partial correctness. Hoare logic [24, 25] is a deductive system for inferring correctness of asserted programs. Cook [14] showed that a Hoare logic similar to the one presented below is complete for proving partial correctness on *expressible interpretations* (interpretations where strongest postconditions are first-order definable) when first-order logic is the assertion language. Lipton [32] showed that expressible structures are either finite or satisfy a very strong condition, viz., that the natural numbers with addition and multiplication be first-order definable. This kind of problem with expressibility led Blass and Gurevich [11] to search for a logic in which strongest postconditions are definable; they found *ELFP*. As we have seen, it is not possible to go further and use *ELFP* as an assertion language. However, it is natural to ask if *SLFP* is a good assertion language for Hoare logic. The answer, we believe, is yes.

Consider the following Hoare logic for the programming language introduced in this section. (This is based on the presentation in Apt [3].) It has one axiom $\{\varphi(x/t)\}\, x{:=}t\, \{\varphi(x)\}$ and four rules of inference:

$$\frac{\{\varphi\}\, S\, \{\psi\} \quad \{\psi\}\, T\, \{\vartheta\}}{\{\varphi\}\, S; T\, \{\vartheta\}},$$

$$\frac{\{\varphi \wedge \beta\}\, S\, \{\psi\} \quad \{\varphi \wedge \neg\beta\}\, T\, \{\psi\}}{\{\varphi\}\ \mathbf{if}\ \beta\ \mathbf{then}\ S\ \mathbf{else}\ T\ \mathbf{fi}\ \{\psi\}},$$

$$\frac{\{\varphi \wedge \beta\}\, S\, \{\varphi\}}{\{\varphi\}\ \mathbf{while}\ \beta\ \mathbf{do}\ S\ \mathbf{od}\ \{\varphi \wedge \neg\beta\}},$$

$$\frac{\varphi \vdash \varphi' \quad \{\varphi'\}S\{\psi'\} \quad \psi' \vdash \psi}{\{\varphi\}S\{\psi\}}.$$

In the last rule, known as the *consequence rule* it is customary to have formulas $\varphi \rightarrow \varphi'$ and $\psi' \rightarrow \psi$ rather than sequents $\varphi \vdash \varphi'$ and $\psi' \vdash \psi$. The reason for this is in partial correctness proofs for a particular interpretation or class of interpretations we may assume that the formulas $\varphi \rightarrow \varphi'$ and $\psi' \rightarrow \psi$ are accepted by an oracle that decides validity in this interpretation or class. We take the point of view here that program verification should not assume an oracle to determine validity of formulas: formulas (or sequents) should be proved. We can fix a set of sentences Γ (possibly the set of sentences true in a particular interpretation or class, or possibly a much smaller set of sentences) and obtain the following *modified consequence rule.*

$$\frac{\Gamma, \varphi \vdash \varphi' \quad \{\varphi'\}S\{\psi'\} \quad \Gamma, \psi' \vdash \psi}{\{\varphi\}S\{\psi\}}.$$

Let us call the deductive system consisting of the rules and axioms of **LS** together with the rules of Hoare logic above (with the modified consequence rule) $\mathbf{H}_\Gamma$.

Theorem 4.3 (Completeness Theorem for Hoare Logic) $\mathbf{H}_\Gamma$ *is a complete deductive system for proving partial correctness of asserted programs on interpretations satisfying* Γ *provided assertions have no free relation variables.*

Proof. The proof is very much like Cook's proof [14] except that we use identities between *SLFP* formulas rather than Cook's expressiveness hypothesis. The idea of the proof is to show by induction on the structure of S that if $\mathfrak{A} \models \{\varphi(\vec{x})\}S\{\psi(\vec{x})\}$ for all $\mathfrak{A}$ satisfying Γ, then $\{\varphi\}S\{\psi\}$ is a theorem of $\mathbf{H}_\Gamma$. We follow the presentation of this proof in section 2.8 of Apt [3] for the simple programming language presented in this section. Cook's original proof for a more general programming language, and proofs for programming languages given in later sections of Apt's paper, can be treated similarly.

We consider only the case where S is a program of the form **while** β **do** S' **od**, the other cases being straightforward. Suppose that $\mathfrak{A} \models \{\varphi(\vec{x})\}S\{\psi(\vec{x})\}$ for all $\mathfrak{A}$ satisfying Γ. We claim that it is enough to show that there is a *loop invariant* $\rho(\vec{x})$ such that these three conditions hold:

$$\Gamma, \varphi(\vec{x}) \vdash \rho(\vec{x}),$$
$$\Gamma, \rho(\vec{x}), \neg\beta(\vec{x}) \vdash \psi(\vec{x}),$$
$$\Gamma, \rho(\vec{x}), \beta(\vec{x}), \tau_{S'}(\vec{x}, \vec{y}) \vdash \rho(\vec{x}).$$

The last condition is equivalent to saying that $\mathfrak{A} \models \{\rho(\vec{x}) \wedge \beta(\vec{x})\}S'\{\rho(\vec{x})\}$ for all $\mathfrak{A}$ satisfying Γ. By the induction hypothesis, $\{\rho(\vec{x}) \wedge \beta(\vec{x})\}S'\{\rho(\vec{x})\}$ is a theorem of $\mathbf{H}_\Gamma$ and thus, by the while-rule, so is $\{\rho(\vec{x})\}S\{\rho(\vec{x}) \wedge \neg\beta(\vec{x})\}$. The first two conditions and the modified consequence rule imply that $\{\varphi(\vec{x})\}S\{\psi(\vec{x})\}$ is a theorem of $\mathbf{H}_\Gamma$.

How do we construct $\rho(\vec{x})$? Regard the first and third conditions above as parts of an inductive definition: we would like $\rho(\vec{x})$ to hold if either $\varphi(\vec{x})$ or $\rho(\vec{x}) \wedge \beta(\vec{x}) \wedge \tau_{S'}(\vec{x}, \vec{y})$ hold. Hence the first and third conditions are satisfied if we take $\rho(\vec{x})$ to be

$$[P(\vec{z}) \equiv \varphi(\vec{z}) \vee \exists\vec{w}\,(P(\vec{w}) \wedge \beta(\vec{w}) \wedge \tau_{S'}(\vec{w}, \vec{z}))]\, P(\vec{x}).$$

By hypothesis,

$$\Gamma, \varphi(\vec{x}), \tau_S(\vec{x}, \vec{y}) \models \psi(\vec{y}),$$

so by the definition of the state transformer for S we have

$$\Gamma, \varphi(\vec{x}), \neg\beta(\vec{y}), [P(\vec{z}) \equiv \vec{z} = \vec{x} \vee \exists\vec{w}\,(P(\vec{w}) \wedge \beta(\vec{w}) \wedge \tau_{S'}(\vec{w}, \vec{z}))]\, P(\vec{x}) \models \psi(\vec{y}).$$

This is equivalent to

$$\Gamma, \neg\beta(\vec{y}), [P(\vec{z}) \equiv \varphi(\vec{z}) \vee \exists\vec{w}\,(P(\vec{w}) \wedge \beta(\vec{w}) \wedge \tau_{S'}(\vec{w}, \vec{z}))]\, P(\vec{y}) \models \psi(\vec{y}),$$

which implies to the second condition above. □

The close connection between inductive definitions and loop invariants in this proof is not surprising. It is well known in the literature of program verification that invariants are fixpoints (see Clarke [18]).

5 Expressibility on Acceptable Structures.

We have argued that *SLFP* is the appropriate assertion language for program verification. The difficulty with *SLFP* is in finding ways to deal with the infinitary rule $([\ ]\vdash)$. We saw in the last section that the lack of a cut rule in LS sometimes allows us to show that $([\ ]\vdash)$ is unnecessary. In this section we show that the most widely used restriction to deal with the problems of first-order logic as an assertion language also eliminates the need for an infinitary rule in *SLFP*.

We mentioned in the last section that to handle total correctness Harel [22] suggested restricting to interpretations in which the natural numbers are definable. He proposed a similar restriction to handle the problem of expressibility of strongest postconditions for partial correctness proofs. He called interpretations satisfying this restriction *arithmetical.* Moschovakis [35] had earlier shown that the same restriction is a sufficient condition for the inductively definable sets on a structure to be precisely the the Π^1_1 sets. He called structures satisfying this condition *acceptable.* An acceptable structure $\mathfrak{A}$ is one on which the natural numbers with addition and multiplication are first-order definable and also there is a first-order formula $\beta(x, y, n)$ defining all finite sequences on $\mathfrak{A}$. Intuitively, $\beta(x, y, n)$ says that x codes a finite sequence whose nth element is y. Here n is in the copy of the natural numbers defined on $\mathfrak{A}$.

Aczel [1] later defined the notion of an *existentially acceptable* structure by making the further restriction that the formulas in the definition of acceptability be first-order existential. He showed that on existentially acceptable structures sets definable by *positive existential induction* are precisely the Σ^0_1 sets. Sets definable by positive existential induction are those definable by *ELFP* formulas of the form

$$[P_1(\vec{x}_1) \equiv \vartheta_1; \cdots; P_k(\vec{x}_k) \equiv \vartheta_k]\,\psi$$

where $\vartheta_1, \ldots, \vartheta_k$ and ψ are existential first-order formulas. Chandra and Harel [12] showed that every *ELFP* formula is equivalent to a formula of this form so on existentially acceptable structures, the *ELFP* definable sets are precisely the Σ^0_1 sets. The *ELFP* formulas are the *SLFP* formulas of negation rank 1. We state a generalization of Aczel's theorem.

Theorem 5.1 *On existentially acceptable structures, the sets definable by* SLFP *formulas of negation rank n are precisely the Σ^0_n sets (i.e., sets in the nth level of the arithmetic hierarchy).*

Proof. The theorem follows by induction on n. The case $n = 1$ is Aczel's theorem. Let n be greater than 1. We know by Theorem 2.2 that *SLFP* formulas of negation rank n are equivalent to stratified queries of depth n. But if we take the canonical stratification we see that a stratified query of depth n is equivalent to a logic query (i.e., a stratified query of depth 1) applied to negations queries of depth $n - 1$. By the induction hypothesis this shows that sets definable by *SLFP* formulas of negation rank n are precisely sets that are Σ^0_1 over complements of Σ^0_{n-1} sets; i.e., they are the Σ^0_n sets. □

If a structure is just expressive, rather than existentially expressive, we can still carry out Aczel's proof but of course we lose the correspondence between levels of the arithmetic hierarchy and the negation ranks of sentences. We obtain the following theorem.

Theorem 5.2 *On acceptable structures,* SLFP *and first-order logic have the same expressibility.*

A consequence of this is the following result of Kolaitis [28].

Corollary 5.3 SLFP *is strictly less expressive than least fixpoint logic on infinite structures.*

Proof. Moschovakis showed that the inductively definable sets are the Π^1_1 sets. These sets are definable in least fixpoint logic. But on the natural numbers, Π^1_1 strictly contains the arithmetic hierarchy, so *SLFP* is strictly less expressive than least fixpoint logic. □

Another consequence is that we do not need an infinitary deductive system to reason about *SLFP* definable sets on acceptable structures since we can use first-order logic instead.

6 Conclusion.

There are still many connections between stratified logic programming and program verification left to be explored. It would be interesting to develop a verification system that relies on the evaluation of stratified logic programs.

The biggest problem of program verification is handling the infinitary proof rule ([ ] ⊢). We have seen that having no cut rule helps in some cases. Also, on certain kinds of structures the need for an infinitary rule disappears. There are other avenues still to be explored.

One direction is to replace the infinitary rule in *SLFP* with a weaker finitary induction rule. The resulting deductive systems will be incomplete if we work on all structures, but many significant mathematical theories are incomplete. Here is an example of a rule that might replace ([ ] ⊢):

$$\frac{\Gamma, \vartheta(P/\rho) \vdash \Delta, \rho \qquad \Gamma, \psi(P/\rho) \vdash \Delta}{\Gamma, [P(\vec{x}) \equiv \vartheta]\, \psi \vdash \Delta.}$$

The rule is useful when ρ defines a relation containing the inductively defined relation P (so the upper left sequent true), but ρ is a close enough approximation to P to be used in place of P in the sequent we are trying to prove (so the upper right sequent is true). It is easy to show that the rule is sound. Notice that the rule could be used in cases where an inductive definition can be replaced with a first-order definition.

Another direction is to develop a system for actually working with infinitary rules. Harel [23] notes that many programming logics embed in $L^{\mathrm{CK}}_{\omega_1\omega}$, which is a restriction of $L_{\omega_1\omega}$ in which conjunctions and disjunctions are recursively enumerable. In the realm of infinitary logics, $L^{\mathrm{CK}}_{\omega_1\omega}$ is considered one of the tamest logics after first-order logic because all mathematical objects associated with the logic — formulas, proofs, and structures needed to prove completeness — exist below the level of the first nonconstructible ordinal ω_1^{CK}. *SLFP* is a sublogic of $L^{\mathrm{CK}}_{\omega_1\omega}$ and is even tamer. At this level, an infinitary proof rule may not be so hard to deal with. Here it may be useful to use techniques from the study of admissible sets (see Barwise [10]).

References

[1] P. Aczel, *Introduction to inductive definitions*, in Handbook of Mathematical Logic, J. Barwise, ed., North-Holland, Amsterdam, 1977, pp. 739–782.

[2] A. V. Aho and J. D. Ullman, *Universality of data retrieval languages*, in Proc. 6th ACM Symp. on Principles of Programming Languages, New York, 1979, Association for Computing Machinery, pp. 110–117.

[3] K. R. Apt, *Ten years of Hoare's logic: A survey — part I*, ACM Trans. Prog. Lang. Syst., 3 (1981), pp. 431–483.

[4] ——, *Logic programming*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., vol. B, North-Holland, Amsterdam, 1990, pp. 493–574.

[5] K. R. Apt and H. A. Blair, *Arithmetic classification of perfect models of stratified programs*, in Proc. 5rd Internat. Conf. on Logic Programming, Cambridge, MA, 1988, M.I.T. Press, pp. 765–779.

[6] K. R. Apt, H. A. Blair, and A. Walker, *Towards a theory of declarative knowledge*, in Foundations of Deductive Databases and Logic Programming, J. Minker, ed., Los Altos, CA, 1988, Morgan Kaufmann, pp. 89–148.

[7] R. J. R. Back, *Correctness Preserving Program Refinements: Proof Theory and Applications*, vol. 131 of Mathematical Centre Tracts, Mathematische Centrum, Amsterdam, 1980.

[8] ——, *Proving total correctness of programs in infinitary logic*, Acta Inform., 15 (1981), pp. 233–249.

[9] R. Barbuti and M. Martelli, *Completeness of the SLDNF-resolution for a class of logic programs*, in Proc. 3rd Internat. Conf. on Logic Programming, vol. 225 of Lecture Notes in Computer Science, New York, 1986, Springer-Verlag, pp. 600–614.

[10] J. Barwise, *Admissible Sets and Structures*, Springer-Verlag, New York, 1975.

[11] A. Blass and Y. Gurevich, *Existential fixed-point logic*, in Computation Theory and Logic, E. Börger, ed., vol. 270 of Lecture Notes in Computer Science, New York, 1987, Springer-Verlag, pp. 20–36.

[12] A. Chandra and D. Harel, *Horn clause queries and generalizations*, J. Logic Programming, 1 (1985), pp. 1–15.

[13] K. J. Compton, *A deductive system for existential least fixpoint logic.* Submitted.

[14] S. A. Cook, *Soundness and completeness of an axiom system for program verification*, SIAM J. Comput., 78 (1978), pp. 70–90.

[15] P. Cousot, *Methods and logics for proving programs*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., vol. B, North-Holland, Amsterdam, 1990, pp. 841–994.

[16] E. DAHLHAUS, *Skolem normal forms concerning the least fixpoint*, in Computation Theory and Logic, E. Börger, ed., vol. 270 of Lecture Notes in Computer Science, New York, 1986, Springer-Verlag, pp. 101–106.

[17] W. P. DE ROEVER, *Recursive Program Schemes: Semantics and Proof Theory*, vol. 70 of Mathematical Centre Tracts, Mathematische Centrum, Amsterdam, 1976.

[18] J. E. M. CLARKE, *Program invariants as fixedpoints*, Computing, 21 (1979), pp. 273–294.

[19] ——, *Programming language constructs for which it is impossible to obtain good Hoare axiom systems*, J. Assoc. Comput. Mach., 26 (1979), pp. 129–147.

[20] E. ENGELER, *Algorithmic properties of structures*, Math. Systems Theory, 1 (1967), pp. 183–195.

[21] ——, *Algorithmic logic*, in Foundations of Computer Science, J. W. D. Bakker, ed., vol. 63 of Mathematical Centre Tracts, Mathematische Centrum, Amsterdam, 1975, pp. 57–85.

[22] D. HAREL, *First-Order Dynamic Logic*, vol. 68 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1979.

[23] ——, *Dynamic logic*, in Handbook of Philosophical Logic, D. M. Gabbay and F. Guenthner, eds., vol. II, Reidel, Boston, 1984, pp. 497–604.

[24] C. A. R. HOARE, *An axiomatic basis for computer programming*, Comm. ACM, 12 (1969), pp. 576–580.

[25] ——, *Procedures and parameters: An axiomatic approach*, in Symp. on Semantics of Algorithmic Languages, E. Engeler, ed., vol. 188 of Lecture Notes in Mathematics, Berlin, 1971, Springer-Verlag, pp. 112–116.

[26] P. C. KANELLAKIS, *Elements of relational database theory*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., vol. B, North-Holland, Amsterdam, 1990, pp. 1073–1156.

[27] C. KARP, *Languages with Expressions of Infinite Length*, North-Holland, Amsterdam, 1964.

[28] P. G. KOLAITIS, *The expressive power of stratified logic programs*, Inform. and Comput., 90 (1991), pp. 50–66.

[29] D. KOZEN AND J. TIURYN, *Logics of programs*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., vol. B, North-Holland, Amsterdam, 1990, pp. 789–840.

[30] J.-L. LASSEZ, V. NGUYEN, AND E. SONENBERG, *Fixed point theorems and semantics: A folk tale*, Information Processing Lett., 14 (1982), pp. 112–116.

[31] D. LEIVANT, *Logical and mathematical reasoning about imperative programs*, in Proc. 11th ACM Symp. on Principles of Programming Languages, New York, 1984, Association for Computing Machinery, pp. 132–140.

[32] R. J. LIPTON, *A necessary and sufficient condition for the existence of Hoare logics*, in Proc. 18th IEEE Symp. on Foundations of Computer Science, Los Angeles, 1977, IEEE Computer Society Press, pp. 1–6.

[33] J. LOECKX AND K. SIEBER, *Foundations of Program Verification*, Wiley, New York, 1984.

[34] E. G. K. LOPEZ-ESCOBAR, *An interpolation theorem for denumerably long sentences*, Fund. Math., 57 (1965), pp. 253–272.

[35] Y. N. MOSCHOVAKIS, *Elementary Induction on Abstract Structures*, North-Holland, Amsterdam, 1974.

[36] L. NAISH, *Negation and Control in PROLOG*, vol. 238 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1986.

[37] D. PARK, *Fixpoint induction and proofs of program properties*, in Machine Intelligence, B. Meltzer and D. Michie, eds., vol. 5, Edinburgh University Press, Edinburgh, 1969, pp. 59–77.

[38] ——, *Finiteness is μ-ineffable*, Theoret. Comput. Sci., 3 (1976), pp. 173–181.

[39] T. PRZYMUSIŃSKI, *On the declarative semantics of deductive databases and logic programs*, in Foundations of Deductive Databases and Logic Programming, J. Minker, ed., Los Altos, CA, 1988, Morgan Kaufmann, pp. 193–216.

[40] A. SALWICKI, *Formalised algorithmic languages*, Bull. Acad. Pol. Sci., Ser. Sci. Math. Astron. Phy., 18 (1981), pp. 227–232.

[41] A. STAVELY, *Proving Programs Correct Using Abstract, High-Level Logic*, PhD thesis, University of Michigan, Ann Arbor, MI, 1977.

[42] G. TAKEUTI, *Proof Theory*, North-Holland, Amsterdam, second ed., 1987.

[43] J. V. TUCKER AND J. I. ZUCKER, *Program Correctness over Abstract Data Types*, vol. 6 of CWI Monographs, North-Holland, Amsterdam, 1988.

[44] A. VAN GELDER, *Negation as failure using tight derivations for general logic programs*, in Proc. 3rd IEEE Conf. on Logic Programming, Los Angeles, 1986, IEEE Computer Society Press, pp. 127–139.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399