STEADFAST LOGIC PROGRAMS

KUNG-KIU LAU, MARIO ORNAGHI AND
STEN-ÅKE TÄRNLUND

▷ We present the notion of steadfastness that at once embodies modularity, reusability, and formal correctness. A steadfast program is an open program with parameters, which is parametrically correct in the sense that it will always be correct with respect to its (open) specification whenever its parameters are computed correctly. Thus, a steadfast program is *correct*, *modular* and *reusable*. Therefore, steadfastness provides a basis for hierarchical construction of correct reusable modules. We first introduce the idea of steadfastness in an informal manner. Then we give a model-theoretic characterisation of steadfastness, followed by an operational semantics based on the (open) completion of an open program. Finally, we apply our general results to program correctness with respect to parametric specifications. ◁

1. INTRODUCTION

In (both standard and constraint) logic programming, we have developed a formal approach (see e.g. [17, 19]) to constructing programs in which we can define and reason about the correctness and reusability of not only parametric programs but also generic classes (see [20, 15, 16]). The cornerstone of our formalisation of reusability and correctness is the notion of *steadfastness*. In model-theoretic terms, steadfastness corresponds to parametric correctness in a class of interpretations.

Address correspondence to Kung-Kiu Lau, Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, United Kingdom, email: kung-kiu@cs.man.ac.uk, Mario Ornaghi, Dipartimento di Scienze dell'Informazione, Universita' degli studi di Milano, Via Comelico 39/41, 20135 Milano, Italy, email: ornaghi@dsi.unimi.it, or Sten-Åke Tärnlund, Computing Science Department, Uppsala University, P.O. Box 520, S-751 20 Uppsala, Sweden, email: stenake@csd.uu.se

The first author was partially supported by the European Union HCM project on Logic Program Synthesis and Transformation, contract no. CHRXCT930414, and by a grant from Consiglio Nazionale delle Ricerche (CNR), Italy, and the British Council, Rome. The third author has been supported by a TMR fellowship from the European Commission, while at the Departamento de Informatica, Universidade Nova de Lisboa, Portugal.

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

It allows us to formalise correctness with respect to parametric specifications, in such a way that correct programs are preserved through inheritance hierarchies of (generic) classes. Steadfast programs thus make ideal units in a library from which correct software can be composed. In this paper, we shall introduce, formally define, and discuss steadfast logic programs.[1]

Steadfastness is a model-theoretic notion of correctness for open or parametric programs. In an open program P, predicates that do not occur as the head of any clause have parametric interpretations. We call them *open predicates*, while the predicates that do appear in clause heads are called *defined predicates*. If we consider a class $\mathcal{I}$ of possible interpretations for the signature Σ of the program P, then we say that P is *steadfast* in $\mathcal{I}$ if, for every interpretation $\mathsf{i} \in \mathcal{I}$, the minimum i-model of P, namely the minimum of the models of P that interpret the open predicates as i, coincides with i.

The class $\mathcal{I}$ of interpretations in which steadfastness is defined therefore plays a key role by providing the context for steadfastness. For a model-theoretic treatment of steadfastness, what $\mathcal{I}$ interprets is not relevant. However for practical program development, $\mathcal{I}$ typically interprets an axiomatisation $\mathcal{F}$ of the problem domain in question. We call $\mathcal{F}$ a *specification framework*, or just framework for short. From a model-theoretic point of view, of course, an open framework $\mathcal{F}$ simply axiomatises a class $\mathcal{I}$ of (intended) models. If $\mathcal{I}$ contains only one model (or an isomorphism class), then $\mathcal{F}$ is *closed*. On the other hand, an open framework $\mathcal{F}$ has many non-isomorphic (intended) models.

We can define steadfastness formally in terms of models of open programs (Section 4), by adapting the completion as well as the minimum model semantics of standard (closed) logic programs. Moreover, this model-theoretic characterisation enables us to show that steadfast programs enjoy properties such as compositionality, reusability and inheritance, that are of crucial importance in modular program development.

In order to use our model-theoretic semantics to prove that an arbitrary, given open program is steadfast, we also define steadfastness in terms of computations of open programs (Section 5), by using variants of SLD derivations and trees, and relating them to provability from the (open) completion *Ocomp* of open programs. That is, we extend the kernel of the theory of (closed) logic programs ([37, 7, 25, 1]) to open logic programs. Based on this extension, we introduce (parametric) existential termination in a class $\mathcal{I}$ of interpretations. In this way, we establish soundness and completeness results for open logic programs. This leads to a major result: if a program P existentially terminates in $\mathcal{I}$, then P is steadfast in $\mathcal{I}$ if and only if its open completion, $Ocomp(P)$, is true in $\mathcal{I}$. $Ocomp(P)$ contains only the completed definitions of the defined predicates of P, but not those of the open ones. This corresponds to the idea that the open predicates are parameters, or place-holders, by allowing them to have open interpretations. Similarly, existential termination in $\mathcal{I}$ corresponds to termination with respect to every interpretation $\mathsf{i} \in \mathcal{I}$ of the open predicates: it means that P terminates under the assumption that open symbols are answered, during a computation, by some program Q that computes according to i.

[1] Note that what we call a 'steadfast logic program' is completely different from what O'Keefe [32, p. 96] calls a 'steadfast predicate', although we also (need to) use 'steadfast' in the sense of 'unchanging'.

Steadfastness provides the model-theoretic basis for parametric correctness, i.e., correctness with respect to parametric specifications. To define and reason about such specifications and parametric correctness (Section 6), we make use of axiomatised frameworks of the form $\langle \Sigma, Ax \cup Th, \mathcal{I} \rangle$, where Σ is a signature, Ax is a decidable set of first-order axioms, Th a set of (already proven) theorems, and $\mathcal{I}$ a class of intended models of Ax. A framework typically embodies an axiomatisation of the problem domain, that provides an unambiguous semantic underpinning for specifications and programs, as well as the correctness relationship between them (see [22]). Axioms and theorems should codify *all* the relevant knowledge on the domain.

In an (open) axiomatised framework $\mathcal{F} = \langle \Sigma, Ax \cup Th, \mathcal{I} \rangle$, a parametric specification of r is a set S_r of formulas, that define r in terms of the symbols of Σ. Thus S_r contains symbols from Σ and the new symbol r. In terms of models, S_r defines an expansion $\mathcal{C}_r$ of $\mathcal{I}$, namely a class of models such that, for every $\mathsf{i} \in \mathcal{I}$, $\mathcal{C}_r$ contains one expanded model i_r that coincides with i over the signature Σ and interprets r according to S_r. We will show that, in the more general setting for program development, we can define correctness of a program with respect to S_r as steadfastness in a class of expansions defined by S_r.

Correctness defined in terms of steadfastness inherits the compositionality, reusability and inheritance properties of steadfastness. Thus steadfastness provides a formal basis for correct program reuse in particular, and for modular development of correct programs in general.

Furthermore, the theorems Th in a framework $\mathcal{F}$ embody knowledge already proved in $\mathcal{F}$. Steadfastness in $\mathcal{F}$ thus enables reuse of not only (correct) programs, but also such (verified) knowledge.

The paper is organised as follows. First, we informally introduce frameworks and the idea of steadfastness in a framework (Section 2). Then we give a formal account of frameworks (Section 3), followed by a model-theoretic characterisation of steadfastness (Section 4), and a computational or proof-theoretic interpretation of steadfastness (Section 5), which together constitute a self-contained formal treatment of steadfastness. A brief discussion of the relationship between steadfastness and program correctness with respect to specifications (Section 6) then rounds off the paper, showing the relevance of steadfastness to practical modular development of correct open programs. Finally, we briefly compare steadfastness with related work on modularity, reuse and correctness.

2. THE IDEA OF STEADFASTNESS

The notion of steadfastness is defined in the context of a problem domain, or more precisely its characterisation. If our problem domain is library information systems, for example, then we need data types like strings and lists of strings to represent book titles and lists of authors, a suitable type for book codes, and so on. We will also need relations like $author(A, C)$, $title(T, C)$, etc. We call the characterisation of a problem domain a *specification framework* $\mathcal{F}$, or framework for short. In this section, we will introduce and illustrate the idea of steadfastness by using this informal description of frameworks. Then in Section 3, we will define frameworks formally.

Example 2.1. Suppose we want to iterate n times a binary operation $\circ$ on some domain D. A *specification framework* for our problem has to contain at least the following:

(i) a (generic) domain D, equipped with a binary operation $\circ$ and a distinguished element e, used in (iii) below;

(ii) the usual structure of natural numbers, to express the number n;

(iii) the iteration operation $\times(a,n) = e \circ a \underbrace{\circ \cdots \circ}_{(n \text{ times})} a$.

In this framework, which we will call $\mathcal{ITER}$, we can specify the relation *iterate* that we want to compute, by the following specification:

$$S_{iterate}(D,\times): \qquad iterate(x,n,z) \leftrightarrow z = \times(a,n) \tag{2.1}$$

$S_{iterate}(D,\times)$ is a *parametric specification*, that is, its meaning depends on the interpretation of the 'parameters' D and $\times$. Natural numbers are not parameters, since they have only one interpretation in the framework. We can derive the following program $P_{iterate}$:

$$\begin{array}{ll} iterate(a,0,v) & \leftarrow unit(v) \\ iterate(a,s(n),v) & \leftarrow iterate(a,n,w),\, op(w,a,v) \end{array}$$

where s is the successor function for natural numbers, and the predicates *unit* and *op* are defined in $\mathcal{ITER}$ by the parametric specifications $S_{unit}(D,e)$ and $S_{op}(D,\circ)$:

$$\begin{array}{lll} S_{unit}(D,e): & unit(u) & \leftrightarrow u = e \\ S_{op}(D,\circ)\ : & op(x,y,z) & \leftrightarrow z = x \circ y \end{array} \tag{2.2}$$

Now $P_{iterate}$ is an *open* program, since it does not contain clauses for computing *unit* and *op*. Thus *unit* and *op* are *open* in $P_{iterate}$. On the other hand, *iterate* is a *defined symbol* in $P_{iterate}$, since we have clauses for computing it.

The question is 'How do we define the correctness of such an open program with respect to a parametric specification, and how do we reason about such correctness?' We will indicate how we might answer this question in general, by considering the particular case of $P_{iterate}$.

Although as it stands $P_{iterate}$ does not contain any code for computing the parameters *unit* and *op*, it makes sense to talk about its correctness: $P_{iterate}$ is *correct* with respect to the specifications (2.1) and (2.2) because it always computes *iterate* correctly in terms of correct computations of *unit* and *op* in any interpretation of $\mathcal{ITER}$. In other words, $P_{iterate}$ can be composed with correct code for *unit* and *op* in that interpretation.

For example, if D is the set of natural numbers, $\circ$ is $+$, e is 0, then $\times(a,n) = 0 + a + \cdots + a = na$. Now, if $Q^{+}_{unit(u)}$ computes $u = 0$ and $Q^{+}_{op(x,y,z)}$ computes $z = x+y$, then $P_{iterate} \cup Q^{+}_{unit(u)} \cup Q^{+}_{op(x,y,z)}$ will compute na, i.e., $P_{iterate}$ is correct in this first interpretation.

As another example, if D is the set of integers, $\circ$ is $-$, e is 0, then $\times(a,n) = 0 - a - \cdots - a = -na$. If $Q^{-}_{unit(u)}$ computes $u = 0$ and $Q^{-}_{op(x,y,z)}$ computes $z = x - y$, then $P_{iterate} \cup Q^{-}_{unit(u)} \cup Q^{-}_{op(x,y,z)}$ will compute $-na$ for an integer a, i.e., $P_{iterate}$ is also correct in this second interpretation.

In general, for any interpretation i of $\mathcal{ITER}$, and programs $Q^{\mathsf{i}}_{unit(u)}$ and $Q^{\mathsf{i}}_{op(x,y,z)}$, the correctness of $P_{iterate} \cup Q^{\mathsf{i}}_{unit(u)} \cup Q^{\mathsf{i}}_{op(x,y,z)}$, and hence the correctness of $P_{iterate}$, is guaranteed provided that $Q^{\mathsf{i}}_{unit(u)}$ and $Q^{\mathsf{i}}_{op(x,y,z)}$ compute *unit* and *op* according to the specifications (2.2).

The reusability of $P_{iterate}$ with different Q's to yield a correct program for different interpretations of the framework is an important semantic property from the point of view of modularity, reusability and correctness in formal program development. We call this property *steadfastness* in an open framework with a class of models.

As an example of a non-steadfast program in $\mathcal{ITER}$, consider the following program $P^*_{iterate}$:

$$
\begin{array}{l}
iterate(a,0,v) \leftarrow unit(v)\\
iterate(a,n,v) \leftarrow m+m=n, iterate(a,m,w), op(w,w,v)\\
iterate(a,n,v) \leftarrow m+s(m)=n, iterate(a,m,w), op(w,w,z), op(z,a,v)
\end{array}
$$

$P^*_{iterate}$ is more efficient than $P_{iterate}$: the number of recursive calls is linear in $P_{iterate}$, whereas it is logarithmic in $P^*_{iterate}$. $P^*_{iterate}$ is correct with respect to (2.1) and (2.2) if D is the set of natural numbers with $+$ as $\circ$ and 0 as e. In contrast, it is incorrect if D is the set of integers with $-$ as $\circ$ and 0 as e. For instance, for $iterate(a,s(s(s(s(0)))),v)$, $P^*_{iterate}$ computes 0 instead of the correct answer $-4a$. Thus $P^*_{iterate}$ is *not* steadfast in $\mathcal{ITER}$.

However, if we require that e and $\circ$ satisfy the additional axioms $\forall x \,.\, e \circ x = x$ and $\forall x,y,z \,.\, x \circ (y \circ z) = (x \circ y) \circ z$, then we can prove that the following properties hold:[2]

$$
\begin{cases}
\times(a,n) = \times(a, n \div 2) \circ \times(a, n \div 2) \circ a & \text{if } n \text{ is odd}\\
\times(a,n) = \times(a, n \div 2) \circ \times(a, n \div 2) & \text{if } n \text{ is even}
\end{cases}
$$

and in the subclass $\mathcal{ITER}^*$ of interpretations of $\mathcal{ITER}$ that satisfy the additional axioms, $P^*_{iterate}$ computes correctly, and thus it is steadfast in $\mathcal{ITER}^*$.

For instance, if D is the set of m-dimensional square matrices, with the m-dimensional identity matrix as e, then since matrix multiplication $\times$ is associative, $P^*_{iterate}$ is correct, where *op* computes matrix products.

3. FRAMEWORKS: THE CONTEXT FOR STEADFASTNESS

In this section we define specification frameworks formally, to set the scene for a formal treatment of steadfastness in the sequel.

A framework is characterised by a *signature* Σ and a class $\mathcal{I}$ of *intended interpretations*. For example, $\mathcal{ITER}$ has a signature containing D, $\circ$, e, $\times$, and the usual signature of natural numbers, and it has the intended interpretations informally described in the previous section. Signatures and interpretations are defined formally as follows:

Definition 3.1. (Signatures)
A (first-order many-sorted) *signature* Σ consists of:

(i) A set S of sort symbols.

[2] Here $m \div n$ means $\lfloor \frac{m}{n} \rfloor$, the floor function.

(ii) A set F of function declarations. Each function declaration has the form $f : a \to s$, where f is the declared function symbol, $a = (s_{i_1}, \ldots, s_{i_n})$ is its arity, and s is its sort.[3] Constants will be functions with empty arity, i.e., $c : \to s$.

(iii) A set R of relation declarations. Each relation declaration has the form $r : a$, where r is the declared relation symbol, and $a = (s_{k_1}, \ldots, s_{k_m})$ is its arity.

The *first-order language* $L_{\Sigma,X}$ generated by a signature Σ and a set X of sorted variables is defined as usual. It contains terms and formulas. Overloaded relation symbols are allowed. Each term has one sort, and we write $\tau : s$ whenever τ is a term of s. Often, the set of sorted variables is implicit, i.e., we shall simply write L_Σ. Formulas of L_Σ are called Σ*-formulas*, and *closed* Σ*-formulas* are called Σ*-sentences*.

Σ-interpretations are defined in the usual way:

Definition 3.2. (Interpretations)
A Σ-*interpretation* i interprets the symbols of a signature Σ in the following way:

(i) Every sort symbol s is interpreted as a domain D_s^{i}.

(ii) Every function declaration $f : a \to s$ is interpreted as a function

$$f^{\mathsf{i}} : D_a^{\mathsf{i}} \to D_s^{\mathsf{i}}$$

where, if $a = (s_1, \ldots, s_n)$, then D_a^{i} is $D_{s_1}^{\mathsf{i}} \times \cdots \times D_{s_n}^{\mathsf{i}}$.

(iii) Every relation declaration $r : a$ is interpreted as a relation

$$r^{\mathsf{i}} \subseteq D_a^{\mathsf{i}}.$$

As usual, an *assignment* $\mathbf{a}$ over an interpretation i is a map $\mathbf{a}$ that associates with every sorted variable $x : s$ an element $\mathbf{a}(x) \in D_s^{\mathsf{i}}$. In an interpretation i and assignment $\mathbf{a}$, every term $\tau : s$ denotes an element of D_s^{i} that we call the *value* of τ in $(\mathsf{i}, \mathbf{a})$, and every formula evaluates to *true* or *false*.

We write $val_{\mathsf{i}}(\tau, \mathbf{a})$ to denote the value of the term τ in an interpretation i and assignment $\mathbf{a}$.

We write $\mathsf{i} \models_{\mathbf{a}} H$ to indicate that H is *true* in i and $\mathbf{a}$; if H is closed, then we simply write $\mathsf{i} \models H$, since the truth of a closed formula does not depend on $\mathbf{a}$. If H is closed and *true* in (every interpretation of) a class $\mathcal{I}$ of interpretations, then we write $\mathcal{I} \models H$.

We will use *first-order logic with identity*, that is, identity axioms and overloaded identity $= : (s, s)$, for every sort s, are understood, and $= : (s, s)$ is interpreted as the identity over D_s^{i}.

As usual, a Σ-interpretation i is a *model* of a set Ax of axioms, written $\mathsf{i} \models Ax$, if $\mathsf{i} \models A$, for every $A \in Ax$.

$Ax \vdash H$ and $Ax \models H$ will have the usual meaning: respectively that H is *provable from* Ax, and that H is a *logical consequence of* Ax. By the Completeness

[3] Here, $s_{i_1}, \ldots, s_{i_n}, s$ indicate sort symbols.

Theorem of first-order logic with identity, $Ax \vdash H$ if and only if $Ax \models H$. As usual, a *theorem* of an axiomatisation Ax is a sentence H such that $Ax \vdash H$.

Often a Σ-interpretation is seen as a Σ-structure:

Definition 3.3. (Structures)
A Σ-*structure* based on an interpretation i is a triple $M^{\mathsf{i}} = \langle D^{\mathsf{i}}, F^{\mathsf{i}}, R^{\mathsf{i}} \rangle$, where D^{i} is the (S-indexed) family of domains interpreting S, F^{i} is the (F-indexed) family of functions interpreting F, and R^{i} is the (R-indexed) family of relations interpreting R.

A Σ-structure thus corresponds more directly to the idea of a data type, where the domains D^{i} are the types of data, and F^{i} and R^{i} are the basic operations and predicates. For convenience, we will use Σ-interpretations, or simply interpretations, and Σ-structures, or simply structures, interchangeably.

Now we can give the formal definition of a framework.

Definition 3.4. (Frameworks)
A *framework* is composed of a signature Σ and a class $\mathcal{I}$ of Σ-interpretations. It will be denoted by $\mathcal{F} = \langle \Sigma, \mathcal{I} \rangle$.

Thus a framework is formalised as a class $\mathcal{I}$ of Σ-structures, that correspond to the data types and relations that are relevant for the problem domain at hand.

For certain purposes, it may be more convenient to represent a framework by an axiomatisation Ax, such that $\mathcal{I}$ is the class of intended models of Ax. We will introduce axiomatised frameworks in Section 6, and show how framework axioms and theorems can be used to reason about specifications and program correctness.

In the next two sections, however, we will concentrate on the model-theoretic semantics of steadfastness, and for this purpose, Definition 3.4 of frameworks will suffice.

4. STEADFASTNESS IN TERMS OF MODELS

As we saw in Section 2, steadfastness of an open program P is meaningful in the context of a framework $\mathcal{F} = \langle \Sigma, \mathcal{I} \rangle$. In model-theoretic terms, steadfastness in $\mathcal{I}$ essentially means that the class of intended models of P coincides with $\mathcal{I}$. In this section, we give such a model-theoretic characterisation of steadfastness of an open program, by adapting the completion, as well as the minimum model semantics, of standard (closed) logic programs.

We shall also establish that steadfast programs enjoy properties such as compositionality, reusability and inheritance, that are of crucial importance in modular program development.

4.1. The Open Completion and its Models

In our approach, programs are defined in the context of a framework $\mathcal{F}$. That is, their relation, constant and function symbols are symbols of the signature of $\mathcal{F}$.

Definition 4.1. (Σ-Programs)
A Σ-*clause* is a Σ-formula of the form $\forall(A \leftarrow B_1 \wedge \cdots \wedge B_n)$, $0 \leq n$, where $A, B_1, \ldots, B_n$ are atoms.
A Σ-*program* is a set of Σ-clauses.

The relation symbols used in a Σ-program P are usually called the *predicates* of P. We shall distinguish between defined and open predicates:

Definition 4.2. (Defined and Open Predicates)
Let P be a Σ-program. A predicate in P is *defined* (by P) if and only if it occurs in the head of at least one clause of P. Otherwise it is *open* (in P).

Often the open predicates of P are also called *parameters*. Their meaning is left open by P, along with the meaning of the sort, constant and function symbols in P. In contrast, the meaning of the defined predicates is determined by P in terms of that of the open symbols. To express this dependence more precisely, we introduce the type of a Σ-program as follows:

Definition 4.3. (Type)
Let P be a Σ-program, where Σ is the signature of the framework. We say that P has *type* $\delta \Leftarrow \Pi$ in Σ, written $P : \delta \Leftarrow \Pi$, if δ are the defined predicates of P, and Π is the subsignature of Σ that does not contain δ.

Thus $P : \delta \Leftarrow \Pi$ indicates that P defines the meaning of δ in terms of interpretations of Π. The symbols of Π will be called the open symbols of P in the signature Σ.

Now we can define the *open completion* of an open program $P : \delta \Leftarrow \Pi$ in terms of formulas that we call P's *open completion axioms*. We shall use the following notation:

Notation.
$\mathsf{eq}(r(\alpha_1, \ldots, \alpha_n), r(\beta_1, \ldots, \beta_n))$ stands for $\alpha_1 = \beta_1 \wedge \ldots \wedge \alpha_n = \beta_n$.
For an open program $P : \delta \Leftarrow \Pi$,

(i) $\mathit{Cdef}(P, r)$ stands for the *completed definition* of the predicate $r \in \delta$, i.e., the formula:

$$\forall a \,.\, r(a) \leftrightarrow (\exists y_1 \,.\, \mathsf{eq}(r(a), A_1) \wedge \mathcal{B}_1) \vee \cdots \vee (\exists y_n \,.\, \mathsf{eq}(r(a), A_n) \wedge \mathcal{B}_n)$$

where $\forall y_j \,.\, A_j \leftarrow \mathcal{B}_j$, $1 \leq j \leq n$, are the clauses of P with defined predicate r (for more details, see e.g. [25]);

(ii) $\mathit{Cdef}^+(P, r)$ stands for the *if-definition* of r in P, i.e., the $\leftarrow$-part of $\mathit{Cdef}(P, r)$;

(iii) $\mathit{Cdef}^-(P, r)$ stands for the *only-if-definition* of r in P, i.e., the $\rightarrow$-part of $\mathit{Cdef}(P, r)$.

Definition 4.4. (Open Completion)
Let $P : \delta \Leftarrow \Pi$ be an open program.

(i) The *positive open completion* of P, $Ocomp^+(P)$, is the set of if-definitions of the predicates δ.

(ii) The *negative open completion* of P, $Ocomp^-(P)$, is the set of only-if-definitions of the predicates δ.

(iii) The *open completion* of P is $Ocomp(P) = Ocomp^+(P) \cup Ocomp^-(P)$.

$Ocomp^+(P)$ is, of course, logically equivalent to P, and we shall use them interchangeably. $Ocomp$ is different from the usual completion, $Comp$, as defined in e.g. [25]. $Comp$ contains $\forall x \,.\, \neg p(x)$, for every open predicate p in P. In contrast, $Ocomp$ does not contain axioms for the open predicates, since they are considered to be parameters. In addition, $Comp$ contains Clark's equality theory CET [1, 6, 25], which is left out of $Ocomp$. We do not assume any fixed equality theory, since the meaning of the constant and function symbols is pre-defined by the (open) framework $\mathcal{F}$.

Now we define the intended models of $Ocomp^+(P)$,[4] $Ocomp^-(P)$ and $Ocomp(P)$. The intended models depend on interpretations of Π, and we shall call them j-models, where j stands for a Π-interpretation. We shall define minimum and maximum j-models.

To define j-models, we need to make use of expansions of interpretations to larger signatures and reducts to smaller ones.

Definition 4.5. (Reducts and Expansions)
Let $\Pi \subset \Sigma$ be a signature, and let i be a Σ-interpretation.

(i) The Π-*reduct* of i, written $\mathsf{i}\,|\,\Pi$, is the Π-interpretation obtained by restricting i to the symbols of Π.

(ii) Conversely, a Σ-*expansion* of a Π-interpretation j is any Σ-interpretation i such that $\mathsf{i}\,|\,\Pi = \mathsf{j}$.

Intuitively, a reduct to a smaller signature forgets the interpretation of the eliminated symbols, and an expansion to a larger signature introduces an interpretation of the *new* symbols, while preserving the interpretation of the old ones.

A useful, well-known property of the reduct $\mathsf{i}\,|\,\Pi$ of a Σ-interpretation i, is that it behaves like i for the formulas of the sub-language $L_\Pi \subseteq L_\Sigma$. More precisely:

Proposition 4.1. Let Π be a subsignature of a signature Σ, i be a Σ-interpretation, and H be a Π-formula. Then $i \models_{\mathbf{a}} H$ if and only if $i\,|\,\Pi \models_{\mathbf{a}|\Pi} H$, where $\mathbf{a}\,|\,\Pi$ is the assignment $\mathbf{a}$ restricted to the variables with sorts from Π.

We will be interested in expansions of Π into a signature Σ that introduce a set δ of new relation symbols (which are the defined predicates of an open program $P : \delta \Leftarrow \Pi$), and we will use the following notions of sub-interpretation, intersection and union with respect to δ:

[4] We will use $Ocomp^+(P)$ instead of P, for uniformity.

Definition 4.6. (Sub-δ-interpretation, δ-union, δ-intersection)
Let Π be a signature and Σ be the signature obtained by introducing in Π a set δ of new relation symbols. Let j be a Π-interpretation, and i_1 and i_2 be two Σ-expansions of j. Then:

(i) i_1 is a *sub-δ-interpretation* of i_2, written $\mathsf{i}_1 \subseteq_\delta \mathsf{i}_2$, if and only if $h^{\mathsf{i}_1} \subseteq h^{\mathsf{i}_2}$, for every $h \in \delta$;

(ii) the *δ-intersection* $\mathsf{i}_1 \cap_\delta \mathsf{i}_2$ is the expansion i of j such that $h^{\mathsf{i}} = h^{\mathsf{i}_1} \cap h^{\mathsf{i}_2}$, for every $h \in \delta$;

(iii) the *δ-union* $\mathsf{i}_1 \cup_\delta \mathsf{i}_2$ is the expansion i of j such that $h^{\mathsf{i}} = h^{\mathsf{i}_1} \cup h^{\mathsf{i}_2}$, for every $h \in \delta$.

For a given Π-interpretation j, $\subseteq_\delta$ is a *complete partial ordering* in the set of Σ-expansions of j, and $\cap_\delta$ and $\cup_\delta$ are the *greatest lower bound* and the *least upper bound* with respect to $\subseteq_\delta$. It is easy to see that the following properties hold for an *atom* A:

$$\begin{array}{l} \mathsf{i}_1 \cap \mathsf{i}_2 \models_{\mathbf{a}} A \quad \text{iff } \mathsf{i}_1 \models_{\mathbf{a}} A \quad \text{and } \mathsf{i}_2 \models_{\mathbf{a}} A \\ \mathsf{i}_1 \cup \mathsf{i}_2 \models_{\mathbf{a}} \neg A \text{ iff } \mathsf{i}_1 \models_{\mathbf{a}} \neg A \text{ and } \mathsf{i}_2 \models_{\mathbf{a}} \neg A \end{array} \tag{4.1}$$

Now we can define j-models of an open Σ-program $P : \delta \Leftarrow \Pi$. We will consider the meaning of Π to be pre-defined, and call j a pre-interpretation:

Definition 4.7. (Pre-interpretations)
Let $P : \delta \Leftarrow \Pi$ be a Σ-program. A Π-interpretation will be called a *pre-interpretation* of P.

This definition of pre-interpretation is an extension of that in [25]. The two definitions become equivalent when P does not contain open predicates, and P's signature coincides with Σ.

For a given pre-interpretation j for a Σ-program $P : \delta \Leftarrow \Pi$, the j-models of $Ocomp^+(P)$, $Ocomp^-(P)$ and $Ocomp(P)$ are defined as follows:

Definition 4.8. (j-models)
Let $P : \delta \Leftarrow \Pi$ be a Σ-program, and j be a pre-interpretation. A j-*model* of $Ocomp^+(P)$ is any Σ-interpretation m such that:

(i) m is a Σ-expansion of j, i.e., $\mathsf{m}|\Pi = \mathsf{j}$;

(ii) m is a model of $Ocomp^+(P)$, i.e., $\mathsf{m} \models Ocomp^+(P)$.

The j-models of $Ocomp^-(P)$ and $Ocomp(P)$ are defined similarly.

If we have a Σ-program $P : \delta \Leftarrow \Pi$ without open predicates, and if Σ coincides with the signature of P, then Π contains only the identity, and the sort, constant and function symbols of P. In this case, the notion of a j-model corresponds to that of a model based on the pre-interpretation j, as in [25].

We can prove the following closure properties:[5]

[5]This works for definite programs and for constraint logic programs [14]. Negation and full first-order languages are used in frameworks.

Proposition 4.2. Let $P : \delta \Leftarrow \Pi$ be a Σ-program, and j be a pre-interpretation. The sets of j-models of $Ocomp^+(P)$ and $Ocomp^-(P)$ are respectively closed under δ-intersection $\cap_\delta$ and δ-union $\cup_\delta$.

For a given pre-interpretation j for a Σ-program $P : \delta \Leftarrow \Pi$, the intended j-model of $Ocomp^+(P)$ is its minimum j-model, written j^{P^+}. It is defined as follows:

Definition 4.9. (Minimum j-models)
Let $P : \delta \Leftarrow \Pi$ be a Σ-program, and j be a pre-interpretation of P. The *minimum j-model* of $Ocomp^+(P)$ is the model j^{P^+} such that $\mathsf{j}^{P^+} \subseteq_\delta \mathsf{m}$, for every j-model m of $Ocomp^+(P)$.

The existence of j^{P^+} follows from the closure under δ-intersection $\cap_\delta$ of the j-models of $Ocomp^+(P)$: j^{P^+} is the δ-intersection of all the j-models. As a consequence of (4.1), we can prove the following:

Proposition 4.3. Let $P : \delta \Leftarrow \Pi$ be a Σ-program. For every predicate $r \in \delta$ and every assignment $\mathbf{a}$, *$\mathsf{j}^{P^+} \models_{\mathbf{a}} r(x)$ if and only if $\mathsf{m} \models_{\mathbf{a}} r(x)$, for every j-model m of $Ocomp^+(P)$.*

Thus $Ocomp^+(P)$ is related to the truth of the defined predicates, and j^{P^+} represents their truth in all the j-models of $Ocomp^+(P)$.

In contrast, $Ocomp^-(P)$ is related to the falsity of the defined predicates.

Definition 4.10. (Maximum j-models)
Let $P : \delta \Leftarrow \Pi$ be a Σ-program, and j be a pre-interpretation of P. The *maximum j-model* of $Ocomp^-(P)$ is the model j^{P^-} such that $\mathsf{m} \subseteq_\delta \mathsf{j}^{P^-}$, for every j-model m of $Ocomp^-(P)$.

The existence of j^{P^-} follows from the closure under δ-union $\cup_\delta$ of the j-models of $Ocomp^-(P)$: j^{P^-} is the δ-union of all the j-models. As a consequence of (4.1), we can prove:

Proposition 4.4. Let $P : \delta \Leftarrow \Pi$ be a Σ-program. For every predicate $r \in \delta$ and every assignment $\mathbf{a}$, *$\mathsf{j}^{P^-} \models_{\mathbf{a}} \neg r(x)$ if and only if $\mathsf{m} \models_{\mathbf{a}} \neg r(x)$, for every j-model m of $Ocomp^-(P)$.*

Finally, the following proposition links j^{P^+}, j^{P^-} and $Ocomp(P)$.

Proposition 4.5. Let $P : \delta \Leftarrow \Pi$ be a Σ-program. Then $\mathsf{j}^{P^+} \models Ocomp^+(P)$, $\mathsf{j}^{P^-} \models Ocomp^-(P)$ and, for every model m of $Ocomp(P)$, $\mathsf{j}^{P^+} \subseteq_\delta \mathsf{m} \subseteq_\delta \mathsf{j}^{P^-}$.

We omit the proofs for these propositions because they are mostly an adaptation of known results (see e.g. [1]) to j-models. In Section 5, we will introduce open *SLD*-derivations and we will show that j^{P^+} coincides with the success set, and j^{P^-} is contained in the finite failure set, of P with respect to j. This will complete the picture. Now we use j-models and their properties to characterise steadfastness.

4.2. Steadfast Logic Programs

Consider an open Σ-program $P : \delta \Leftarrow \Pi$, in the context of a framework $\langle \Sigma, \mathcal{I} \rangle$. Any interpretation $\mathsf{i} \in \mathcal{I}$ contains a pre-interpretation $\mathsf{i} \,|\, \Pi$ of the open predicates of P, i.e., $\mathsf{i} \,|\, \Pi$ acts as parameter passing. Thus the minimum $(\mathsf{i} \,|\, \Pi)$-model of P represents the interpretation of δ *defined by* P with parameter passing $\mathsf{i} \,|\, \Pi$. If this interpretation coincides with i, then we can say that P is correct with respect to i. If this happens for *every* $\mathsf{i} \in \mathcal{I}$, then we can say that P is correct with respect to $\mathcal{I}$. Steadfastness is just this kind of *model-theoretic correctness* in a class of interpretations.

Definition 4.11. (Steadfast Logic Programs)
Let $\mathcal{I}$ be a class of Σ-interpretations and $P : \delta \Leftarrow \Pi$ be an open Σ-program. Then:

(i) P is *steadfast* in a Σ-interpretation i if $(\mathsf{i} \,|\, \Pi)^{P^+} = \mathsf{i}$, i.e., if the minimum $(\mathsf{i} \,|\, \Pi)$-model of $Ocomp^+(P)$ coincides with i.

(ii) P is *steadfast* in $\mathcal{I}$ iff it is steadfast in every interpretation $\mathsf{i} \in \mathcal{I}$.

We can prove a useful, necessary and sufficient condition for steadfastness in purely model-theoretic terms, for Σ-programs that decide the defined predicates.

To decide a predicate, we use j^{P^+} for evaluating truth and j^{P^-} for evaluating falsity.[6] Thus, deciding $r(x)$ means that, for every assignment of x, either $r(x)$ is evaluated by j^{P^+}, or $\neg r(x)$ is evaluated by j^{P^-}. Thus we introduce the following definition:

Definition 4.12. (Decision Programs)
Let $P : \delta \Leftarrow \Pi$ be a Σ-program.

(i) P *decides* $r \in \delta$ in a pre-interpretation j whenever for each assignment $\mathbf{a}$, either $\mathsf{j}^{P^+} \models_{\mathbf{a}} r(x)$ or $\mathsf{j}^{P^-} \models_{\mathbf{a}} \neg r(x)$.

(ii) P *decides* r in a class $\mathcal{C}$ of pre-interpretations if and only if it decides r for every $\mathsf{j} \in \mathcal{C}$.

We can easily prove the following theorem:

Theorem 4.1. $P : \delta \Leftarrow \Pi$ decides δ (i.e., it decides every $r \in \delta$) in a class $\mathcal{J}$ of pre-interpretations if and only if, for every $j \in \mathcal{J}$, $j^{P^+} = j^{P^-}$.

We get the following steadfastness condition:

Theorem 4.2. Let $\mathcal{I}$ be a class of Σ-interpretations and $P : \delta \Leftarrow \Pi$ be a Σ-program that decides δ in $\mathcal{I} \,|\, \Pi$. Then P is steadfast in $\mathcal{I}$ if and only if $\mathcal{I} \models Ocomp(P)$.

PROOF. $(\rightarrow)$ Consider a generic $\mathsf{i} \in \mathcal{I}$. By Theorem 4.1, $\mathsf{i} = (\mathsf{i} \,|\, \Pi)^{P+} = (\mathsf{i} \,|\, \Pi)^{P-}$. Then i is a model of $Ocomp^+(P)$ and $Ocomp^-(P)$. Since $\mathsf{i} \in \mathcal{I}$ is generic, we get $\mathcal{I} \models Ocomp(P)$.

[6]This assumption implicitly presupposes a logic of evaluation where the excluded middle does not hold. Three-valued logic could work, though it may not be the best choice, but we shall not discuss this issue in this paper.

($\leftarrow$) Consider a generic $\mathsf{i} \in \mathcal{I}$. $\mathsf{i} \models Ocomp^+(P)$ and $\mathsf{i} \models Ocomp^-(P)$. Then $(\mathsf{i}\,|\,\Pi)^{P+} \subseteq_\delta \mathsf{i} \subseteq_\delta (\mathsf{i}\,|\,\Pi)^{P-}$, and our assert follows from $(\mathsf{i}\,|\,\Pi)^{P+} = (\mathsf{i}\,|\,\Pi)^{P-}$. $\square$

Theorem 4.2 shows the role of the open completion. The picture will be completed in Section 5.4, where decision programs will be linked to appropriate termination properties.

4.3. Composition of Steadfast Programs

We can also study the the compositional properties of steadfast programs purely model-theoretically.

Definition 4.13. (Program Composition)
Let $P : \delta_1 \Leftarrow \Pi_1$ and $Q : \delta_2 \Leftarrow \Pi_2$ be two Σ-programs.

(i) The *composition* of P and Q is simply the union $P \cup Q$.

(ii) If no predicate of δ_1 belongs to Q, then we say the composition is *hierarchical*, and we denote it by $P[Q]$.

Lemma 4.1. (Hierarchical Program Composition) *Let $P : \delta_1 \Leftarrow \Pi_1$ and $Q : \delta_2 \Leftarrow \Pi_2$ be two Σ-programs. If P and Q are steadfast in a Σ-interpretation i and the composition of P and Q is hierarchical, then $P[Q]$ is steadfast in i.*

Proof. The type of $P \cup Q$ is $\delta_1 \cup \delta_2 \Leftarrow \Pi_3$, where $\Pi_3 = \Pi_2 \setminus \delta_1 = \Pi_1 \setminus \delta_2$. By the steadfastness of P and Q, i is a model of P and Q, hence of $P \cup Q$. Then it contains the minimum $(\mathsf{i}\,|\,\Pi_3)$-model of $P \cup Q$:

$$(\mathsf{i}\,|\,\Pi_3)^{(P\cup Q)+} \subseteq_{\delta_1 \cup \delta_2} \mathsf{i} \tag{4.2}$$

Let Σ_P and Σ_Q be the signatures containing the predicates of P and of Q respectively, as well as equality and the sort, constant and function symbols of Σ. Let Π_P be the open symbols of P in Σ_P, and Π_Q be those of Q in Σ_Q. The proof goes on as follows.

$(\mathsf{i}\,|\,\Pi_3)^{(P\cup Q)+}$ is a model of $P \cup Q$, hence of Q. By Proposition 4.1, its restriction to Σ_Q is a model of Q. Since Π_3 coincides with Π_Q over Σ_Q (indeed $\delta_1 \notin \Sigma_Q$ because $P[Q]$ is definite), the restriction to Σ_Q is a $(\mathsf{i}\,|\,\Pi_Q)$-model of Q (with signature Σ_Q) and contains the minimum $(\mathsf{i}\,|\,\Pi_Q)$-model. The latter coincides with $\mathsf{i}\,|\,\Sigma_Q$, because Q is steadfast. Then:

$$\mathsf{i}\,|\,\Sigma_Q \subseteq_{\delta_2} (\mathsf{i}\,|\,\Pi_3)^{(P\cup Q)+}\,|\,\Sigma_Q \tag{4.3}$$

Since $\Pi_3 \cup \delta_2 = \Pi_1$ and Π_1 does not contain δ_1, by (4.2) and (4.3) we obtain:

$$\mathsf{i}\,|\,\Pi_1 = (\mathsf{i}\,|\,\Pi_3)^{(P\cup Q)+}\,|\,\Pi_1 \tag{4.4}$$

Therefore $(\mathsf{i}\,|\,\Pi_3)^{(P\cup Q)+}$ is an $(\mathsf{i}\,|\,\Pi_1)$-model of $P \cup Q$, hence of P, and we can reason about P and $P \cup Q$ as we have done for Q and $P \cup Q$. We obtain:

$$\mathsf{i}\,|\,\Sigma_P \subseteq_{\delta_1} (\mathsf{i}\,|\,\Pi_3)^{(P\cup Q)+}\,|\,\Sigma_P \tag{4.5}$$

The final proof comes from (4.2), (4.3) and (4.5). $\square$

The relevance of steadfastness for modularity and reusability is expressed by the following results. We omit the proofs, since they are immediate.

Theorem 4.3. (Hierarchical Compositionality) *Let $P : \delta_1 \Leftarrow \Pi_1$ and $Q : \delta_2 \Leftarrow \Pi_2$ be two Σ-programs that are steadfast in a class $\mathcal{I}$ of Σ-interpretations. If the composition of P and Q is hierarchical, then $P[Q]$ is steadfast in $\mathcal{I}$.*

Theorem 4.4. (Inheritance) *Let $\mathcal{I}$ be a class of Σ-interpretations, and $P : \delta_1 \Leftarrow \Pi_1$ be a Σ-program that is steadfast in $\mathcal{I}$. Then P is steadfast in any subclass $\mathcal{I}^* \subseteq \mathcal{I}$.*

A subclass $\mathcal{I}^* \subseteq \mathcal{I}$ is typically obtained by adding new axioms. For example, if we axiomatise rectangles, the subclass of squares is obtained by adding an axiom that states that all the edges are equal in length. Inheritance means that programs remain correct in subclasses. Thus it enables reuse.

Corollary 4.1. (Reuse) *Let $P : \delta_1 \Leftarrow \Pi_1$ be a Σ-program that is steadfast in a class $\mathcal{I}$ of Σ-interpretations. Consider a program $Q : \delta_2 \Leftarrow \Pi_2$ that is steadfast in a subclass $\mathcal{I}^* \subseteq \mathcal{I}$. If the composition of P and Q is hierarchical, then $P[Q]$ is steadfast in $\mathcal{I}^*$.*

Example 4.1. Consider a program $P : \delta \Leftarrow \mathit{iterate}, \ldots$ that uses *iterate* as a parameter and is steadfast in $\mathcal{ITER}$ of Example 2.1. Consider the program $P_2 : \mathit{iterate} \Leftarrow \mathit{unit}, \mathit{op}, \ldots$, that is steadfast in the subclass $\mathcal{ITER}^*$ of $\mathcal{ITER}$ that satisfies the additional associativity axiom. Since P is inherited by $\mathcal{ITER}^*$, the hierarchical composition $P[P_2] : \delta, \mathit{iterate} \Leftarrow \mathit{unit}, \mathit{op}, \ldots$ is steadfast in $\mathcal{ITER}^*$, i.e., P can be correctly reused in subclasses.

In contrast, non-hierarchical composition may not preserve steadfastness.

Example 4.2. Consider the open framework $\mathcal{F}(D, p, q)$ with a signature containing one sort D and two relations p and q, and with a class $\mathcal{I}$ of interpretations that satisfy the axiom $\forall x \,.\, p(x) \leftrightarrow q(x)$. Let $P : p \Leftarrow q, D$ be a program containing the one clause $\forall x(p(x) \leftarrow q(x))$, and $Q : q \Leftarrow p, D$ be a program containing the one clause $\forall x(q(x) \leftarrow p(x))$. We can easily see that both P and Q are steadfast in $\mathcal{I}$. However, the composition $P \cup Q : p, q \Leftarrow D$ is not steadfast, since, for every $\mathsf{i} \in \mathcal{I}$, p and q are interpreted as the empty relations in the minimum $(\mathsf{i}\,|\,D)$-model.

If we restrict our attention to decision programs, however, then non-hierarchical composition works, as shown by the following result:

Theorem 4.5. (Decision Program Composition) *Let $P : \delta_1 \Leftarrow \Pi_1$ and $Q : \delta_2 \Leftarrow \Pi_2$ be two Σ-programs that are steadfast in a Σ-interpretation i and decide δ_1 and δ_2 (respectively) in i. Let $P \cup Q : \delta_1 \cup \delta_2 \Leftarrow \Pi_3$ be their union. If $P \cup Q$ decides $\delta_1 \cup \delta_2$ in i, then $P \cup Q$ is steadfast in i.*

PROOF. By Theorem 4.2, i is a model of $\mathit{Ocomp}(P)$ and of $\mathit{Ocomp}(Q)$. Since $\mathit{Ocomp}(P \cup Q) = \mathit{Ocomp}(P) \cup \mathit{Ocomp}(Q)$, i is a model of $\mathit{Ocomp}(P \cup Q)$. Since $P \cup Q$ decides $\delta_1 \cup \delta_2$, by Theorem 4.2 it is steadfast in i. $\square$

Example 4.3. In Example 4.2, P decides p and Q decides q, but $P \cup Q : p, q \Leftarrow D$ does not decide p, q. Indeed, for every $\mathsf{i} \in \mathcal{I}$, p and q are interpreted as the empty relations in the minimum $(\mathsf{i}\,|\,D)$-model of $\mathit{Ocomp}^+(P \cup Q)$, while they are interpreted as the total relations in the maximum model of $Ocomp^{-}(P \cup Q)$. Thus we cannot conclude that $P \cup Q$ is steadfast.

Consider now the closed framework with the signature containing one sort N, one constant 0, one unary function s and two unary relations *even* and *odd*, and with a unique interpretation i such that N^{i} is the set of natural numbers, and 0, s (successor), *even* and *odd* are interpreted in the usual way. Consider the programs P_{even} and P_{odd} (as usual, universal quantification of program clauses is understood):

$$
\begin{array}{lll}
P_{even}: & even(0) & \leftarrow \\
 & even(s(x)) & \leftarrow odd(x) \\
P_{odd}: & odd(s(x)) & \leftarrow even(x)
\end{array}
$$

Now all the hypotheses of Theorem 4.5 hold, and we can conclude that $P_{even} \cup Q_{odd}$ is steadfast in i.

It is obvious that the property of being a decision program is *inherited* by subclasses. Therefore, by Theorem 4.5, we obtain:

Corollary 4.2. In the class of decision programs, (non-hierarchical) program composition preserves steadfastness, and steadfastness is inherited by subclasses.

This corollary is interesting, since non-hierarchical composition $P \cup Q$ allows mutual recursion. However, to establish the steadfastness of a mutually recursive composition $P \cup Q$ we need to verify *a posteriori* that $P \cup Q$ is a decision program. This makes non-hierarchical composition unsuitable for modular programming. However, it is useful for program derivation, since it allows us to derive separately clauses of single predicates and then compose them by verifying that the resulting program decides its predicates. As we will see, this can be achieved by verifying existential termination.

In contrast, (hierarchical) composition $P[Q]$ works for modular programming, because it preserves steadfastness unconditionally.

5. STEADFASTNESS IN TERMS OF COMPUTATIONS

In this section, we shall consider the models of an open program in terms of computations.

To this end, we introduce suitable variants of SLD derivations and trees, that we call $\mathrm{SLD_E}$-derivations and $\mathrm{SLD_E}$-trees, and we link them to provability from $Ocomp^{+}(P)$ and $Ocomp^{-}(P)$ respectively.

$\mathrm{SLD_E}$-derivations and trees allow us to define j-success and j-failure sets of an open program $P : \delta \Leftarrow \Pi$ parametrically, i.e., success and finite failure sets in terms of pre-interpretations j, and to study their relationship with j-models. In this way, we can relate the model-theoretic results of Section 4 to their computational counterpart. We introduce $\mathcal{J}$-complete programs, a subclass of decision programs that satisfy suitable termination properties. This will provide a basis for formal proof methods of steadfastness by proving open completion and termination.

5.1. Computations of Open Programs

First we introduce $\mathrm{SLD_E}$-derivations and trees. We shall use goals that contain equations as well as atoms, and adopt the following notation:

Notation.
We write A, B, ..., for atoms, $\mathcal{A}$, $\mathcal{B}$, ..., for (possibly empty) sequences of atoms, and E, E_1, ..., for (possibly empty) sequences of equations.
The conjunction of a sequence of formulas is written $\wedge(\ldots)$. We shall omit $\wedge$ and the parentheses whenever possible without confusion. For example, a clause with head C and body $\wedge(\mathcal{B})$ will be written as $\forall x \,.\, C \leftarrow \mathcal{B}$.
We write G, G_1, ... for goals. To indicate a goal with atoms $\mathcal{A}$ and equations E, we write $\mathcal{A}, \mathsf{E}$.

Now we can define $\mathrm{SLD_E}$-derivations.

Definition 5.1. ($\mathrm{SLD_E}$-derivations)

(i) Let $\mathcal{A}_1, A, \mathcal{A}_2, \mathsf{E}$ be a goal containing an atom A. Then

$$\mathcal{A}_1, A, \mathcal{A}_2, \mathsf{E} \quad \overset{\forall x \,.\, C \leftarrow \mathcal{B}}{\Longrightarrow} \quad \mathcal{A}_1, \rho\mathcal{B}, \mathcal{A}_2, \mathsf{E}, \mathsf{eq}(A, \rho C)$$

is a $\mathrm{SLD_E}$*-step* with *selected atom* A and *applied clause* $\forall x \,.\, C \leftarrow \mathcal{B}$, if A and C have the same relation symbol and ρ is a renaming of the variables x of the clause by variables that do not occur in $\mathcal{A}_1, A, \mathcal{A}_2, \mathsf{E}$.

(ii) An *SLD_E-derivation* from a program P is a finite sequence of $\mathrm{SLD_E}$-steps, such that each step applies a clause of P and introduces *new* renaming variables, i.e., variables not introduced in previous steps.

For short, an $\mathrm{SLD_E}$-derivation $G_1 \overset{\mathtt{K}_1}{\Longrightarrow} \ldots \overset{\mathtt{K}_{n-1}}{\Longrightarrow} G_n$ from a program P is written $G_1 \overset{P}{\Longrightarrow} G_n$. The logical meaning of $\mathrm{SLD_E}$-derivations follows from our next result.

Lemma 5.1. For every SLD_E-derivation $G_1 \overset{P}{\Longrightarrow} G_n$, $Ocomp^+(P), G_n \vdash \wedge G_1$

PROOF. It is easy to see that, if $G \overset{\mathtt{K}}{\Longrightarrow} G'$, then $\mathtt{K}, G' \vdash \wedge G$. $\square$

Now we can introduce $\mathrm{SLD_E}$-proofs.

Definition 5.2. ($\mathrm{SLD_E}$-proofs)
An $\mathrm{SLD_E}$-derivation $G_1 \overset{P}{\Longrightarrow} G_n$ is an *SLD_E-proof* if the final goal G_n contains only equations.

$\mathrm{SLD_E}$-proofs correspond to SLD-refutations, where the answer substitution σ is replaced by the equations E in the final goal. To extract σ from E, first-order logic with identity is not sufficient: we need the freeness axioms [36] which are included in Clark's equality theory *CET*. We do not assume *CET* here, however, and other equality theories could be considered, i.e., our treatment of steadfastness can be extended to constraint logic programs for instance.

Example 5.1. Consider a program for deciding the *even* relation:

$$\begin{array}{ll} \mathtt{K}_1: & even(0) \leftarrow \\ \mathtt{K}_2: & even(s(s(x))) \leftarrow even(x) \end{array}$$

A possible $\mathrm{SLD_E}$-derivation is:

$$\begin{aligned} even(s(x_0)) &\xRightarrow{\mathtt{K}_2} even(x_1), s(x_0) = s(s(x_1)) \\ &\xRightarrow{\mathtt{K}_1} s(x_0) = s(s(x_1)), x_1 = 0 \end{aligned} \tag{5.1}$$

By Lemma 5.1, we have:

$$\mathtt{K}_1,\ \mathtt{K}_2,\ s(x_0) = s(s(x_1)),\ x_1 = 0 \vdash even(s(x_0))$$

In the intended model of natural numbers, 0 and s satisfy CET and we can easily see that $CET, s(x_0) = s(s(x_1)), x_1 = 0 \vdash x_0 = s(0)$, i.e., the final equations contain the answer substitution.

Now we can define $\mathrm{SLD_E}$-trees. They are defined like standard SLD-trees, with the following differences: they are finite, and paths are (partial) $\mathrm{SLD_E}$-derivations or (complete) $\mathrm{SLD_E}$-proofs.

Definition 5.3. ($\mathrm{SLD_E}$-trees)
An *SLD_E-tree* generated by a goal G from a program P is a finite tree, with goals as nodes, such that the root is G and, for every node G', either G' is a leaf, or G' has k children $G'_1, \ldots, G'_k$, where $r(\tau)$ is the selected atom, $\mathtt{K}_1, \ldots, \mathtt{K}_k$ are all the clauses with head predicate r, and $G' \xRightarrow{\mathtt{K}_i} G'_i$, for $1 \le i \le k$.

Let $\triangle$ be an SLD_E-tree from P with root $G(x)$ and leaves $G_1, \ldots, G_n$. Every leaf $G_i, 1 \le i \le n$, contains all the variables x of the root, in the equations introduced by the first $\mathrm{SLD_E}$-step, and the renaming variables w_i introduced in the path ending with G_i. We will write $G_i(x, w_i)$ to indicate the variables of G_i. We can prove the following lemma, which explains the logical meaning of $\mathrm{SLD_E}$-trees.

Lemma 5.2. Let $P : \delta \Leftarrow \Pi$ be an open program, $G(x)$ be a goal with variables x, and $\triangle$ be an SLD_E-tree from P with root $G(x)$ and leaves $G_1(x, w_1), \ldots, G_n(x, w_n)$. Then

$$Ocomp^-(P), G(x) \vdash \bigvee_{i=1}^{n} (\exists w_i \,.\, G_i(x, w_i)).$$

PROOF. Let $\mathcal{A}_1, r(\tau), \mathcal{A}_2, \mathsf{E}$ be a node of the tree with selected atom $r(\tau)$. We can directly see that

$$Cdef^-(P, r), r(\tau) \vdash \bigvee_{i=1}^{k} \exists v_i \,.\, \mathsf{eq}(r(\tau), \rho_i A_i) \wedge \rho_i \mathcal{B}_i$$

where $\exists y_i \,.\, \mathsf{eq}(r(a), A_i) \wedge \mathcal{B}_i$, $1 \le i \le k$, are the disjuncts of the right hand side of $Cdef^-(P, r)$, and ρ_i renames y_i by v_i. Since v_i $(1 \le i \le k)$ do not occur in

$\mathcal{A}_1, \mathcal{A}_2, \mathsf{E}$, we can prove:

$$Cdef^-(P,r), \exists w \,.\, \mathcal{A}_1, r(\tau), \mathcal{A}_2, \mathsf{E} \vdash \bigvee_{i=1}^{k} \exists w, v_i \,.\, \mathcal{A}_1, \rho_i\mathcal{B}_i, \mathcal{A}_2, \mathsf{E}, \mathsf{eq}(r(\tau), \rho_i A_i)$$

where w are the variables of $\mathcal{A}_1, r(\tau), \mathcal{A}_2, \mathsf{E}$ that are different from x.

Now the proof comes by an easy induction on the size of the tree. □

5.2. Success Sets of Open Programs

Roughly speaking, for every fixed pre-interpretation j, the j-success set of P is what P computes successfully in j. Technically, the j-success set of P, written $\mathsf{j}^{P_{ss}}$, is a Σ-expansion of j, and it is defined operationally, by $\mathrm{SLD_E}$-derivability.

Moreover, $\mathsf{j}^{P_{ss}}$ can be shown to coincide with the minimum j-model j^{P^+} of P, and thereby we achieve a (desired) correspondence between operational and model-theoretic semantics. Here we will concentrate on the definition of $\mathsf{j}^{P_{ss}}$ and its properties. Other details on computations are deferred to Section 5.4.

To define $\mathsf{j}^{P_{ss}}$, we need the notion of open $\mathrm{SLD_E}$-proofs, or open proofs for short.

Definition 5.4. (Open Proofs)
Let $P : \delta \Leftarrow \Pi$ be an open program. An $\mathrm{SLD_E}$-derivation $G_1 \overset{P}{\Longrightarrow} G_n$ is an *open SLD_E-proof* from P if G_n contains only equations and (possibly) open atoms from Π.

The final goal G_n of an open proof contains only Π-formulas. Therefore, G_n can be interpreted by Π-interpretations, i.e., pre-interpretations for $P : \delta \Leftarrow \Pi$, and we can define successful atoms in terms of Π-interpretations.

Definition 5.5. (Successful Proofs)
Let $P : \delta \Leftarrow \Pi$ be an open Σ-program, j be a Π-interpretation, and $\mathbf{a}$ be an assignment.

(i) An open proof $G_1(x) \overset{P}{\Longrightarrow} G_n(x,y)$ is a *successful proof* in j for $\mathbf{a}$ whenever $\mathsf{j} \models_{\mathbf{a}} \exists y \,.\, G_n(x,y)$.

(ii) An atom $A(x)$ is a *successful atom* in j for $\mathbf{a}$ if there is a successful open proof $A(x) \overset{P}{\Longrightarrow} G_n(x,y)$ in j for $\mathbf{a}$.

Example 5.2. Consider the following open program with a defined predicate *ord* and an open predicate $\leq$:

$$\begin{array}{ll} \mathtt{K}_1 : & ord(nil) \leftarrow \\ \mathtt{K}_2 : & ord(x.nil) \leftarrow \\ \mathtt{K}_3 : & ord(x.y.L) \leftarrow x \leq y \wedge ord(y.L) \end{array}$$

Consider the open goal $ord(a.b.nil)$. There is an open proof:

$$\begin{array}{l} ord(a.b.nil) \overset{\mathtt{K}_3}{\Longrightarrow} x_0 \leq y_0,\ ord(y_0.L_0),\ a.b.nil = x_0.y_0.L_0 \\ \overset{\mathtt{K}_2}{\Longrightarrow} x_0 \leq y_0,\ a.b.nil = x_0.y_0.L_0,\ x_1.nil = y_0.L_0 \end{array} \tag{5.2}$$

The final goal contains the open predicate $x_0 \leq y_0$. By Definition 5.5, $ord(a.b.nil)$ is successful for those pre-interpretations j of $\leq$ and assignments $\mathbf{a}$ such that

$$\mathsf{j} \models_{\mathbf{a}} \exists x_0, y_0, L_0, x_1 \ .\ x_0 \leq y_0 \wedge a.b.nil = x_0.y_0.L_0 \wedge x_1.nil = y_0.L_0$$

If we restrict ourselves to the intended interpretations of lists, then the list constructors nil and $\cdot$ satisfy CET. Since

$$CET \vdash (\exists x_0, y_0, L_0, x_1 \ .\ x_0 \leq y_0 \wedge a.b.nil = x_0.y_0.L_0 \wedge x_1.nil = y_0.L_0) \\ \leftrightarrow a \leq b$$

$ord(a.b.nil)$ is successful for those pre-interpretations j of $\leq$ and assignments $\mathbf{a}$ such that $\mathsf{j} \models_{\mathbf{a}} a \leq b$.

Now, we can define the j-success sets of an open program.

Definition 5.6. (j-Success Sets)
Let $P : \delta \Leftarrow \Pi$ be an open Σ-program, and j be a Π-interpretation. The *success set* of P determined by j, written $\mathsf{j}^{P_{ss}}$, is the Σ-expansion of j such that, for each $r \in \delta$, $\mathsf{j}^{P_{ss}} \models_{\mathbf{a}} r(x)$ if and only if $r(x)$ is successful in j for $\mathbf{a}$.

The j-success set is related to the models of $Ocomp^{+}(P)$ by the following results.

Lemma 5.3. Let $P : \delta \Leftarrow \Pi$ be an open Σ-program and j be a Π-interpretation. Let m be a j-model of $Ocomp^{+}(P)$. Then, for every assignment $\mathbf{a}$, if $r(x)$ is successful in j for $\mathbf{a}$, then $\mathsf{m} \models_{\mathbf{a}} r(x)$, for $r \in \delta$.

Proof. Since $r(x)$ is successful, there exists a successful open proof $r(x) \overset{P}{\Longrightarrow} G_n(x, y)$. By Lemma 5.1, $Ocomp^{+}(P), G_n(x, y) \vdash r(x)$. Therefore, $Ocomp^{+}(P)$, $\exists y \,.\, G_n(x, y) \vdash r(x)$. Since $\mathsf{m} \models Ocomp^{+}(P)$ (by hypothesis) and $\mathsf{m} \models_{\mathbf{a}} \exists y \,.\, G_n(x, y)$ (because the open proof is successful, $\mathsf{j} = \mathsf{m} \mid \Pi$ and, by Proposition 4.1, $\mathsf{m} \models_{\mathbf{a}} \exists y_n \,.\, G_n(x, y_n)$ iff $\mathsf{m} | \Pi \models_{\mathbf{a}} \exists y_n \,.\, G_n(x, y_n)$), we get $\mathsf{m} \models_{\mathbf{a}} r(x)$. □

Theorem 5.1. (Soundness of j-success) *Let $P : \delta \Leftarrow \Pi$ be an open Σ-program, and j be a Π-interpretation. Then $\mathsf{j}^{P_{ss}} \subseteq_{\delta} \mathsf{j}^{P^{+}}$.*

Proof. The theorem holds if and only if, for every assignment $\mathbf{a}$ and every j-model m of $Ocomp^{+}(P)$,

$$\mathsf{j}^{P_{ss}} \models_{\mathbf{a}} r(x) \Rightarrow \mathsf{m} \models_{\mathbf{a}} r(x).$$

This follows from Lemma 5.3 and the fact that, if $\mathsf{j}^{P_{ss}} \models_{\mathbf{a}} r(x)$, then $r(x)$ is successful in j for $\mathbf{a}$. □

This theorem is a soundness result with respect to truth: j-success entails truth in $\mathsf{j}^{P^{+}}$ and, hence, in every j-model m of P. The following theorem is a completeness result: the j-success set and the minimum j-model coincide.

Theorem 5.2. (Completeness of j-success) *Let $P : \delta \Leftarrow \Pi$ be an open Σ-program, and j be a pre-interpretation for P. Then $\mathsf{j}^{P_{ss}} = \mathsf{j}^{P^{+}}$.*

Proof. By Theorem 5.1, $\mathsf{j}^{P_{ss}} \subseteq_{\delta} \mathsf{j}^{P^{+}}$. Therefore, if $\mathsf{j}^{P_{ss}}$ is a model of $Ocomp^{+}(P)$, then it is the minimum one $\mathsf{j}^{P^{+}}$. To prove that $\mathsf{j}^{P_{ss}}$ is a model of $Ocomp^{+}(P)$, we use P instead of $Ocomp^{+}(P)$, since they are logically equivalent.

Let $\forall x \,.\, C \leftarrow \mathcal{B}$ be a generic clause of P and $\mathbf{a}$ be a generic assignment. We have to prove $\mathsf{j}^{P_{ss}} \models_{\mathbf{a}} C \leftarrow \mathcal{B}$.

Assume $\mathsf{j}^{P_{ss}} \models_{\mathbf{a}} \mathcal{B}$. Let ρ be a renaming of x by new variables y. Let $\mathbf{b}$ be an assignment such that $\mathbf{a}(x) = \mathbf{b}(x) = \mathbf{b}(y)$. Then $\mathsf{j}^{P_{ss}} \models_{\mathbf{b}} \rho\mathcal{B}, x = y$ and the atoms of $\rho\mathcal{B}$ have successful open proofs, which can be used to build an open proof

$$\rho\mathcal{B}, x = y \ \stackrel{P}{\Longrightarrow}\ G(x, y, w)$$

that is successful in j for $\mathbf{b}$. Therefore $\mathsf{j} \models_{\mathbf{b}} \exists w \,.\, G(x, y, w)$. Now we can build the open proof [7]

$$C \ \stackrel{\forall x \,.\, C \leftarrow \mathcal{B}}{\Longrightarrow}\ \rho\mathcal{B}, x = y \ \stackrel{P}{\Longrightarrow}\ G(x, y, w).$$

Since $\mathbf{a}(x) = \mathbf{b}(x)$, by $\mathsf{j} \models_{\mathbf{b}} \exists w \,.\, G(x, y, w)$ we obtain $\mathsf{j} \models_{\mathbf{a}} \exists y, w \,.\, G_m(x, y, w)$. Therefore $\mathsf{j}^{P_{ss}} \models_{\mathbf{a}} C$. $\square$

5.3. Failure Sets of Open Programs

Now we introduce $\mathrm{SLD_E}$-trees failed in a pre-interpretation j. Using such trees we will define operationally the finite failure set of a program P determined by j.

Definition 5.7. (Failed $\mathrm{SLD_E}$-trees)
Let $P : \delta \Leftarrow \Pi$ be an open Σ-program, j be a Π-interpretation, and $\mathbf{a}$ be an assignment.

(i) An $\mathrm{SLD_E}$-tree with root $G(x)$ and leaves $G_1(x, y_1), \ldots, G_n(x, y_n)$ is a *failed SLD_E-tree* in j for $\mathbf{a}$ if and only if $\mathsf{j} \models_{\mathbf{a}} \neg\exists y_i \,.\, G_i(x, y_i)$, for $1 \leq i \leq n$.

(ii) An atom $A(x)$ is a *failed atom* in j for $\mathbf{a}$ if there is an $\mathrm{SLD_E}$-tree $\triangle$ with root $A(x)$, such that $\triangle$ is failed in j for $\mathbf{a}$.

Definition 5.8. (j-failure Sets)
Let $P : \delta \Leftarrow \Pi$ be a Σ-program. The *finite failure set* of P determined by a Π-interpretation j, written $\mathsf{j}^{P_{ff}}$, is the Σ-expansion of j such that, for each $r \in \delta$,

$$\mathsf{j}^{P_{ff}} \models_{\mathbf{a}} \neg r(x) \ \text{ if and only if } \ r(x) \text{ is failed in } \mathsf{j} \text{ for } \mathbf{a}.$$

Lemma 5.4. Let $P : \delta \Leftarrow \Pi$ be an open Σ-program, $G(x)$ be a goal, and $\triangle$ be an $\mathrm{SLD_E}$-tree with root $G(x)$ and leaves $G_1(x, y_1), \ldots, G_n(x, y_n)$. Let j be a Π-interpretation and m be a j-model of $Ocomp^-(P)$. If $\triangle$ is failed in j for an assignment $\mathbf{a}$, then $\mathsf{m} \models_{\mathbf{a}} \neg G(x)$.

PROOF. By Lemma 5.2:

$$Ocomp^-(P), G(x) \vdash \bigvee_{i=1}^{n} (\exists y_i \,.\, G_i(x, y_i)).$$

Since $\mathsf{m} \models Ocomp^-(P)$ and $\mathsf{m} \models_{\mathbf{a}} \neg\exists y_i \,.\, G_i(x, y_i)$ for $1 \leq i \leq n$ (indeed, $\triangle$ is failed in j, $\mathsf{j} = \mathsf{m} \,|\, \Pi$ and, by Proposition 4.1, $\mathsf{m} \models_{\mathbf{a}} \neg\exists y_i \,.\, G_i(x, y_i)$ iff $\mathsf{m} \,|\, \Pi \models_{\mathbf{a}} \neg\exists y_i \,.\, G_i(x, y_i)$), we get $\mathsf{m} \models_{\mathbf{a}} G(x) \rightarrow \mathit{false}$, i.e., $\mathsf{m} \models_{\mathbf{a}} \neg G(x)$. $\square$

[7] Note that $\mathsf{eq}(C, \rho C)$ is $x = y$.

Theorem 5.3. (Soundness of j-failure) *Let $P : \delta \Leftarrow \Pi$ be an open Σ-program. For every Π-interpretation j, $j^{P^-} \subseteq_\delta j^{P_{ff}}$.*

PROOF. The theorem holds if, for every assignment $\mathbf{a}$ and every j-model m of $Ocomp^-(P)$,

$$\mathsf{j}^{P_{ff}} \models_{\mathbf{a}} \neg r(x) \Rightarrow \mathsf{m} \models_{\mathbf{a}} \neg r(x)$$

This follows from Lemma 5.4 and the fact that, if $\mathsf{j}^{P_{ff}} \models_{\mathbf{a}} \neg r(x)$, then $r(x)$ is failed in j for $\mathbf{a}$. □

This is a soundness result with respect to falsehood, i.e., j-failure entails falsehood in j^{P^-}, and hence in every j-model of $Ocomp^-(P)$. A completeness result does not hold unconditionally for j-failure. Indeed there are programs and pre-interpretations such that $\mathsf{j}^{P^-} \subset_\delta \mathsf{j}^{P_{ff}}$ (see [1] for examples of Herbrand models). However, there are conditional completeness results, such as the following (we omit the easy proof):

Theorem 5.4. (Completeness of j-failure) *Let $P : \delta \Leftarrow \Pi$ be an open Σ-program. Then:*

(i) For every pre-interpretation j, $j^{P^-} = j^{P_{ff}}$ if and only if $j^{P_{ff}}$ is a model of $Ocomp^-(P)$.

(ii) If $j^{P_{ss}} = j^{P_{ff}}$, then $j^{P^+} = j^{P^-} = j^{P_{ff}}$.

Our next definition combines the completeness of j-success and j-failure.

Definition 5.9. ($\mathcal{J}$-completeness)
Let $P : \delta \Leftarrow \Pi$ be a Σ-program. Let $\mathcal{J}$ be a class of Π-interpretations. P is $\mathcal{J}$-complete if and only if $\mathsf{j}^{P_{ff}} = \mathsf{j}^{P_{ss}}$, for every $\mathsf{j} \in \mathcal{J}$.

In $\mathcal{J}$-complete programs, $\mathsf{j}^{P_{ss}} = \mathsf{j}^{P^+} = \mathsf{j}^{P^-} = \mathsf{j}^{P_{ff}}$ for every $\mathsf{j} \in \mathcal{J}$. We have now arrived at a result for establishing steadfastness using the notion of $\mathcal{J}$-completeness.

Theorem 5.5. (Steadfastness Condition) *Let $\mathcal{I}$ be a class of Σ-interpretations, and let $P : \delta \Leftarrow \Pi$ be an open Σ-program such that P is $\mathcal{I} \,|\, \Pi$-complete. Then P is steadfast in $\mathcal{I}$ if and only if $\mathcal{I} \models Ocomp(P)$.*

The theorem follows directly from theorems 4.1, 4.2, and 5.4.

So, now we can establish steadfastness by proving the open completion and $\mathcal{J}$-completeness.

5.4. Steadfastness and Open Termination

The preceding results relate truth and j^{P^+} to the j-success set of a program, and falsity and j^{P^-} to the j-failure set. Thus we have arrived at a connection between models and computations. As a result, we can also relate Theorem 5.5 to termination of open programs. We shall do so in this section, and for this purpose, we shall consider $\mathrm{SLD_E}$-derivations as computations of an idealised 'open interpreter', which computes the pre-interpretations 'virtually'. We will omit the proofs of theorems in this section, since they are easy consequences of those of the previous sections.

A computation for a program $P : \delta \Leftarrow \Pi$ *starts* with a goal $G_0(x, y)$, a Π-interpretation j and an assignment $\mathbf{a}$ of x, where x is a subset of the variables x, y

of G_0. In the limiting cases, x or y may be empty.

For a given j, G_0 and $\mathbf{a}$, the interpreter generates $\mathrm{SLD_E}$-derivations as follows. Let $\mathcal{O}_i$ and E_i be the open atoms and equations of the goal $G_i(x,y,w_i)$, obtained after i $\mathrm{SLD_E}$-steps, $0 \leq i$. The interpreter decides $\mathsf{j} \models_{\mathbf{a}} \exists y, w_i \,.\, \mathcal{O}_i, \mathsf{E}_i$. We have three cases:

(i) $\exists y, w_i \,.\, \mathcal{O}_i, \mathsf{E}_i$ is false in j. Then G_i is *failed* and the current $\mathrm{SLD_E}$-derivation is halted.

(ii) $\exists y, w_i \,.\, \mathcal{O}_i, \mathsf{E}_i$ is true in j, and G_i coincides with $\mathcal{O}_i, \mathsf{E}_i$, i.e., it contains only open atoms and equations. The current derivation is halted with *computed answer* $\mathcal{O}_i, \mathsf{E}_i$.

(iii) $\exists y, w_i \,.\, \mathcal{O}_i, \mathsf{E}_i$ is true in j, and $G_i = \mathcal{A}_i, \mathcal{O}_i, \mathsf{E}_i$. The interpreter *selects* an atom in $\mathcal{A}_i$ and a clause, performs the corresponding $\mathrm{SLD_E}$-step and generates a new goal G_{i+1}.

In (iii), we assume a *fair selection rule.* Search in the $\mathrm{SLD_E}$-tree is performed in the usual way, with backtracking, using either a depth-first or a breadth-first strategy.

The answer of a successful derivation $G_0(x,y) \stackrel{P}{\Longrightarrow} G_i(x,y,w_i)$ is a goal $G_i(x,y,w_i)$ that contains only open atoms and equations. Since $\mathsf{j} \models_{\mathbf{a}} \exists y, w_i \,.\, G_i(x,y,w_i)$, there is an assignment $\mathbf{b}$ of y such that $\mathsf{j} \models_{\mathbf{a}\cup\mathbf{b}} \exists w_i \,.\, G_i(x,y,w_i)$. Therefore the $\mathrm{SLD_E}$-derivation is successful in j for $\mathbf{a}\cup\mathbf{b}$, and we obtain:

$$\mathsf{j}^{P_{ss}} \models_{\mathbf{a}\cup\mathbf{b}} G_0(x,y).$$

We will call $\mathbf{a}$ the *input assignment* and $\mathbf{b}$ *the output assignment*. The output assignment is (virtually) extracted from the final goal by the interpreter, which is assumed to (virtually) compute j. In practice, we have the following cases:

(i) If only equations are involved, and the constants and functions of the program satisfy *CET*, then by *CET* we can solve the equations, and compute the output assignment. Moreover, in this case, $\mathrm{SLD_E}$-derivations can be easily translated into standard *SLD*-derivations.

(ii) If G_i contains only formulas of a constraint language, then we get constraint logic programs. The answer will be computed by the constraint solver.

(iii) G_i contains open predicates that are not solved by *CET* or by a constraint solver. The idea is that they should be computed by other not yet known programs $Q_1, \ldots, Q_n$, to be composed with $P : \delta \Leftarrow \Pi$. Therefore the open interpreter 'virtually computes' pre-interpretations, in place of $Q_1, \ldots, Q_n$, i.e., it is assumed to contain an 'oracle', in the sense of relative computability. Here, effective computability of pre-interpretations is not needed. Whenever, in a pre-interpretation, a (steadfast) program for the open predicates exists, it can replace the oracle, since it composes correctly with our (steadfast) P.

Now we can relate termination to $\mathcal{J}$-completeness, where $\mathcal{J}$ is the class of pre-interpretations. A necessary and sufficient condition for $\mathcal{J}$-completeness is existential termination in $\mathcal{J}$, defined below. Here we assume a *breadth-first search strategy* and a *fair selection rule*. Termination of 'real' programs , as well as other kinds of termination, will be briefly addressed in Section 7.

Definition 5.10. (Existential Termination)
Let $P : \delta \Leftarrow \Pi$ be an open Σ-program, and r be a predicate of δ. Then:

(i) *P existentially terminates* in a Π-interpretation j with respect to r whenever, for every assignment $\mathbf{a}$ of x, either $r(x)$ is successful, or $r(x)$ is finitely failed in j for $\mathbf{a}$.

(ii) $P : \delta \Leftarrow \Pi$ *existentially terminates* in a class $\mathcal{J}$ of pre-interpretations if it existentially terminates in every $\mathsf{j} \in \mathcal{J}$, with respect to every $r \in \delta$.

Existential termination has the following meaning. For every pre-interpretation $\mathsf{j} \in \mathcal{J}$ and every input assignment $\mathbf{a}$ of x, we get a positive or a negative answer in finite time, i.e., the interpreter halts either with a successful $\mathrm{SLD_E}$-derivation or with a finitely failed $\mathrm{SLD_E}$-tree. In the first case, $\mathsf{j}^{P_{ss}} \models_{\mathbf{a}} r(x)$, and in the second $\mathsf{j}^{P_{ff}} \models_{\mathbf{a}} \neg r(x)$. As a consequence, $\mathsf{j}^{P_{ff}} = \mathsf{j}^{P_{ss}}$.
On the other hand, if $\mathsf{j}^{P_{ff}} = \mathsf{j}^{P_{ss}}$ then we have existential termination. Then we have the following:

Theorem 5.6. $P : \delta \Leftarrow \Pi$ existentially terminates with respect to every predicate of δ, in a class $\mathcal{J}$ of pre-interpretations, if and only if P is $\mathcal{J}$-complete.

Moreover, since $\mathsf{j}^{P_{ff}} = \mathsf{j}^{P_{ss}}$ entails $\mathsf{j}^{P^-} = \mathsf{j}^{P^+}$, we have:

Theorem 5.7. If $P : \delta \Leftarrow \Pi$ existentially terminates with respect to (the predicates of) δ, in a class $\mathcal{J}$ of pre-interpretations, then P decides δ in $\mathcal{J}$.

Finally, as an immediate consequence of Theorem 5.6 and Theorem 5.5, we get the following steadfastness condition:

Theorem 5.8. Let $\mathcal{I}$ be a class of Σ-interpretations, and let $P : \delta \Leftarrow \Pi$ be an open Σ-program which existentially terminates in $\mathcal{I} \mid \Pi$. Then P is steadfast in $\mathcal{I}$ if and only if $\mathcal{I} \models Ocomp(P)$.

These theorems provide the basis for methods for proving steadfastness in a framework. Such methods consist of two steps: proving the open completion and proving termination. We discuss their application to correctness in the next section.

6. STEADFASTNESS AND PROGRAM CORRECTNESS

Our treatment of steadfastness thus far is complete in its own right, providing both an abstract, model-theoretic characterisation and an operational semantics. However, so far we have not explicitly explained our motivation for characterising steadfastness, or its relation to the usual notion of program correctness. To round off this paper, in this section we will demonstrate the applicability of steadfastness to program correctness, and thereby clarify our motivation to use steadfastness as a formal basis for modular development of correct programs. For brevity and clarity, we will be somewhat informal, and omit some proofs.

6.1. Parametric Correctness

In this section we show how parametric correctness, i.e., correctness of open programs, can be formalised in terms of steadfastness. The standard definition of correctness (of closed programs) is with respect to a *specification*, as follows (see e.g. [13]):

> A (closed) program P for computing a relation $r(x)$ is *correct* with respect to a specification S if, for every tuple t of ground terms, the following correctness condition holds:

$$S \models r(t) \;\text{ iff }\; P \vdash r(t) \tag{6.1}$$

We shall extend this definition to open programs.

Now a specification of an open program $P : \delta \Leftarrow \pi$ is a pair (S_δ, S_π), where S_δ and S_π are respectively the specification of the defined predicates δ, and that of the open ones π. Any reasonable definition of correctness of open programs should satisfy the following *reuse requirement*:

> If P is correct with respect to (S_δ, S_π), then, for every program Q that is correct with respect to S_π, the hierarchical composition $P[Q]$ should be correct with respect to $S_\delta \cup S_\pi$.

Furthermore, to be useful, an open program should compute different relations δ, when composed with different programs Q. Therefore we should consider parametric specifications of the form $(S_\delta(p), S_\pi(p))$. For example, the parametric specification of the program $P_{iterate}$ in Example 2.1 (in Section 2) is of this form:

$$(\{S_{iterate}(D, \times)\}, \{S_{op}(D, \circ), S_{unit}(D, e)\}) \tag{6.2}$$

where *iterate*, *op* and *unit* are defined in terms of the interpretation of the parameters D, $\times$, $\circ$, and e. Thus we define the correctness of open programs with respect to parametric specifications as follows:[8]

> An open program $P : \delta \Leftarrow \pi$ is *correct* with respect to the specification $\{(S_\delta(p), S_\pi(p))\}$ if and only if, for every instantiation $p := \mathbf{k}$ of the parameters, P is correct with respect to $(S_\delta(\mathbf{k}), S_\pi(\mathbf{k}))$.

This ensures that the aforementioned reuse requirement holds for every instantiation.

Now parametric correctness can be formalised precisely in an abstract, model-theoretic way, in terms of steadfastness. As a first step, for closed programs (and non-parametric specifications), we reformulate (6.1) in model-theoretic terms, as follows:

$$\mathsf{i}_S \models r(t) \;\text{ iff }\; m_P \models r(t) \tag{6.3}$$

where the domain of both i_S and m_P is the Herbrand Base of P, m_P is the minimum Herbrand model of P, and i_S interprets r according to S. Since m_P and i_S have the same domain, (6.3) is equivalent to:

$$\mathsf{i}_S = m_P \tag{6.4}$$

i.e., P is steadfast in i_S.

As a second step, we formalise a parametric specification $\{(S_\delta(p), S_\pi(p))\}$ in a framework $\mathcal{F} = \langle \Sigma, \mathcal{J} \rangle$. The parameters p are symbols of Σ, and their interpretations in $\mathcal{J}$ represent the possible instantiations of p. For example, in the parametric

[8] This definition is very similar to that in [9].

specification (6.2), we can consider the instantiation $D :=$ natural numbers, $\circ := +$, $e := 0$, $\times := *$.

Finally, each instance $(S_\delta(\mathbf{k}), S_\pi(\mathbf{k}))$ of a parametric specification $\{(S_\delta(p), S_\pi(p))\}$ defines an interpretation i_k of δ and π. If P is *steadfast* in the class of all i_k's, then it is correct with respect to the parametric specification. Indeed, by Lemma 4.1, it correctly composes with every steadfast program Q for computing π, in every interpretation i_k, i.e., $P[Q]$ is steadfast in i_k, as required by (6.4).

To formalise the relation between steadfastness and parametric correctness, we first define specifications as follows:

Definition 6.1. (Specifications)
Let $\mathcal{F} = \langle \Sigma, \mathcal{J} \rangle$ be a framework and η be a set of relation symbols not in Σ.
A *specification* S_η of η is a set of $(\Sigma + \eta)$-axioms.[9] S_η is *sound* in $\mathcal{F}$ if, for every $\mathsf{j} \in \mathcal{J}$, there is at least one $(\Sigma + \eta)$-expansion j_η of j such that $\mathsf{j}_\eta \models S_\eta$.
If every $\mathsf{j} \in \mathcal{J}$ has exactly one such expansion j_η, then S_η is *strict* in $\mathcal{F}$; otherwise it is *non-strict*.

An example of a strict specification is an explicit definition:

Definition 6.2. (Explicit Definitions)
In a framework $\mathcal{F} = \langle \Sigma, \mathcal{J} \rangle$, an *explicit definition* of a new relation $r(x)$ is a formula S_r of the form

$$\forall x \,.\, r(x) \leftrightarrow R(x)$$

where $R(x)$ is a Σ-formula.

Explicit definitions are sound in every framework $\mathcal{F}$. The specifications of *iterate*, *op* and *unit*, considered in Example 2.1, are examples of explicit definitions.

An example of a non-strict specification is a super/sub specification:

Definition 6.3. (Super/Sub Specifications)
In a framework $\mathcal{F} = \langle \Sigma, \mathcal{J} \rangle$, a *super/sub* specification of a new relation $r(x)$ is a formula $SupSub_r$ of the form

$$\forall x \,.\, (R_{sub}(x) \rightarrow r(x)) \wedge (r(x) \rightarrow R_{super}(x))$$

where $R_{sub}(x)$ and $R_{super}(x)$ are Σ-formulas such that $\mathcal{J} \models \forall x \,.\, R_{sub}(x) \rightarrow R_{super}(x)$.

The condition $\mathcal{J} \models \forall x \,.\, R_{sub}(x) \rightarrow R_{super}(x)$ guarantees the soundness of the specification in the framework. For every interpretation $\mathsf{j} \in \mathcal{J}$, a $(\Sigma + r)$-expansion j_r of j is a model of $SupSub_r$ if and only if r^{j_r} is contained in the relation defined by R_{super} and contains the one defined by R_{sub}. Therefore, $SupSub_r$ is non-strict.

Other examples of non-strict specifications include *conditional specifications* and *selector specifications*. These are defined and discussed in [21], where it is also shown that conditional specifications are a special case of super/sub specifications.

To define parametric correctness in terms of steadfastness, we need to use S_η-expansions defined thus:

[9] $\Sigma + \eta$ is the signature obtained by including the declarations of η in Σ.

Definition 6.4. (S_η-expansions)
Let $\mathcal{F} = \langle \Sigma, \mathcal{J} \rangle$ be a framework, and S_η be a specification of η in $\mathcal{F}$.
An *S_η-expansions* of $\mathcal{J}$ is a class $\mathcal{J}_\eta$ of $(\Sigma+\eta)$-interpretations such that $\mathcal{J}_\eta \models S_\eta$ and, for every $\mathrm{j} \in \mathcal{J}$, $\mathcal{J}_\eta$ contains one expansion j_η of j.

Of course, a strict S_η has one S_η-expansion, while a non-strict S_η has many.

Now, we can define parametric correctness in terms of steadfastness. In the sequel, we will consider the correctness of (open) $\Sigma+\delta+\pi$-programs $P : \delta \Leftarrow \Pi$, with defined predicates δ and open predicates π, with respect to specifications (S_δ, S_π). Definitions and results apply also to closed programs, as they are a limiting case $(\pi = \emptyset)$. For conciseness, the signature and type of programs will be left implicit, since they can be reconstructed from the specifications.

Definition 6.5. (Parametric Correctness)
Let $\mathcal{F} = \langle \Sigma, \mathcal{J} \rangle$ be a framework, and let P be an open program with specification (S_δ, S_π).
P is *correct* in $\mathcal{F}$ with respect to (S_δ, S_π) if and only if, for every S_π-expansion $\mathcal{J}_\pi$ of $\mathcal{J}$ there is an S_δ-expansion $\mathcal{I}$ of $\mathcal{J}_\pi$ such that P is steadfast in $\mathcal{I}$.

Note that, in the limiting case where S_π and S_δ are strict, this definition simplifies to the requirement that P is steadfast in the (unique) $(S_\pi \cup S_\delta)$-expansion of $\mathcal{J}$.

Finally, by this definition, correct programs inherit the compositional properties of steadfast programs:

Theorem 6.1. (Composition of Correct Programs) *Let $\mathcal{F} = \langle \Sigma, \mathcal{J} \rangle$ be a framework, let S_{δ_1}, S_{π_1}, S_{δ_2}, S_{π_2} be specifications, let P be a correct program with respect to $(S_{\delta_1}, S_{\pi_1} \cup S_{\delta_2})$, and let Q be correct with respect to $(S_{\delta_2}, S_{\pi_2})$. Then $P[Q]$ is correct with respect to $(S_{\delta_1} \cup S_{\delta_2}, S_{\pi_1} \cup S_{\pi_2})$.*

PROOF. Consider a generic $(S_{\pi_1} \cup S_{\pi_2})$-expansion $\mathcal{J}_{\pi_1 \cup \pi_2}$ of $\mathcal{J}$. We have to prove that $P[Q]$ is steadfast in a $(S_{\delta_1} \cup S_{\delta_2})$-expansion $\mathcal{I}$ of $\mathcal{J}_{\pi_1 \cup \pi_2}$.

Since $\mathcal{J}_{\pi_1 \cup \pi_2}$ is an expansion of a S_{π_2}-expansion by predicates π_1 that do not occur in Q, by the correctness of Q, there is a S_{δ_2}-expansion $\mathcal{I}'$ of $\mathcal{J}_{\pi_1 \cup \pi_2}$, such that Q is steadfast in $\mathcal{I}'$. Since $\mathcal{I}'$ is an expansion of a $(S_{\pi_1} \cup S_{\delta_2})$-expansion of $\mathcal{J}$ by predicates π_2 that do not occur in P, then by the correctness of P there is a S_{δ_1}-expansion $\mathcal{I}$ of $\mathcal{I}'$ such that P is steadfast in $\mathcal{I}$. Since δ_1 do not occur in Q, Q is also steadfast in $\mathcal{I}$ and, by Theorem 4.3, $P[Q]$ is steadfast in $\mathcal{I}$. $\Box$

Correctness is also inherited by sub-frameworks:

Theorem 6.2. (Inheritance of Correct Programs) *Let $\mathcal{F} = \langle \Sigma, \mathcal{J} \rangle$ be a framework and let $\mathcal{J}^* \subset \mathcal{J}$ be a subclass of $\mathcal{J}$. If a program P is correct with respect to (S_δ, S_π) in $\mathcal{F}$, then it is correct with respect to (S_δ, S_π) in the sub-framework $\mathcal{F}^* = \langle \Sigma, \mathcal{J}^* \rangle$.*

As usual, inheritance allows reuse:

Corollary 6.1. (Reuse of Correct Programs) *If an open program P is correct with respect to $(S_{\delta_1}, S_{\pi_1} \cup S_{\delta_2})$ in a framework $\mathcal{F} = \langle \Sigma, \mathcal{J} \rangle$, and a program Q is correct with respect to $(S_{\delta_2}, S_{\pi_2})$ in a subframework $\mathcal{F}^* = \langle \Sigma, \mathcal{J}^* \rangle$, then $P[Q]$ is correct with respect to $(S_{\delta_1} \cup S_{\delta_2}, S_{\pi_1} \cup S_{\pi_2})$ in $\mathcal{F}^*$.*

In the limiting case where π_1 and π_2 are empty and $\mathcal{J}^*$ contains one interpretation $\mathsf{j} \in \mathcal{J}$, Q has no open predicates and is correct with respect to S_{δ_2} in j, and $P[Q]$ is correct with respect to $S_{\delta_1} \cup S_{\delta_2}$ in j. Since this holds for every $\mathsf{j} \in \mathcal{J}$, our definition of parametric correctness meets the reuse requirement, as expected.

6.2. Proving Correctness

Having defined parametric correctness (with respect to parametric specifications) in terms of steadfastness, it remains to show how we can use this definition to prove correctness. To do so, we shall need to represent and make use of knowledge codified by first-order axioms and theorems. That is, we shall use axiomatised frameworks, defined as follows:

Definition 6.6. (Axiomatised Frameworks)
An *axiomatised framework* $\mathcal{F} = \langle \Sigma, Ax \cup Th, \mathcal{J} \rangle$ contains a *signature* Σ, a set $Ax \cup Th$ of *axioms and theorems*, and a class $\mathcal{J}$ of *intended models* of Ax. If $\mathcal{J}$ contains a unique (up to isomorphism) model, then $\mathcal{F}$ is *closed*, otherwise it is *open*.

In general, the class $\mathcal{J}$ of the intended models is properly contained in the class of all first-order models of Ax and is defined by means of suitable higher-order assumptions such as the closed world assumption [6]. However, the results in this section are largely independent from any specific intended model semantics. Therefore we simply assume that there is a class $\mathcal{J}$ of intended models of Ax.

Th is a set of theorems that are provable (in first-order logic) from Ax. It does not characterise $\mathcal{J}$, but it is very important because it represents what is actually *known*. As yet unknown theorems are not useful at all, even if they are 'implicitly present' in the axioms Ax.

Now we exhibit two examples of axiomatised frameworks. The first one shows a typical closed framework, namely Peano's Arithmetic. The second one shows a parametric framework, i.e., an open framework where some symbols of the signature act as parameters and, for every interpretation of the parameters, there is one intended interpretation of the other symbols. In these examples, the intended models are singled out by means of suitable domain assumptions.

Example 6.1. (Closed Axiomatised Frameworks)
Peano Arithmetic $\mathcal{NAT}$ can be defined by the following closed axiomatised framework:[10]

Framework $\mathcal{NAT}$;
SORTS: Nat;
FUNCTIONS: $0 \ : \to Nat$;
$s \ : Nat \to Nat$;
$+, * : (Nat, Nat) \to Nat$;

[10] We will omit the outermost universal quantifiers.

$$
\begin{array}{ll}
\text{AXIOMS:} & \neg 0 = s(x) \wedge (s(a) = s(b) \rightarrow a = b); \\
& x + 0 = x; \\
& x + s(y) = s(x + y); \\
& x * 0 = 0; \\
& x * s(y) = x + x * y; \\
& H(0) \wedge (\forall i \,.\, H(i) \rightarrow H(s(i)) \rightarrow \forall x \,.\, H(x). \\
\text{THEOREMS:} & x + (y + z) = (x + y) + z; \\
& x * (y * z) = (x * y) * z; \\
& \vdots \\
& x + y = y + x; \\
& \ldots
\end{array}
$$

We assume the following *domain assumption*:

> *Nat* is the Herbrand Base generated by 0 and s, i.e., it is the set of numerals $0, s(0), s(s(0)), \ldots$ We can consider numerals as abstract representations of the natural numbers $0, 1, 2, \ldots$

Under this assumption, $\mathcal{NAT}$ is a closed framework. Indeed it has only one model that satisfies the domain assumption.

Example 6.2. (Parametric Axiomatised Frameworks)
The following parametric axiomatised framework defines the (kernel of the) theory of lists with parametric element type *Elem* and a parametric binary relation $\lhd$ on *Elem*:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{Framework } \mathcal{LIST}(\mathit{Elem}, \lhd);} \\
\text{IMPORT:} & \mathcal{NAT}; \\
\text{SORTS:} & \mathit{Elem}, \mathit{List}; \\
\text{FUNCTIONS:} & \mathit{nil} \;:\, \rightarrow \mathit{List}; \\
& \;\;.\;\; : (\mathit{Elem}, \mathit{List}) \rightarrow \mathit{List}; \\
& \mathit{nocc} : (\mathit{Elem}, \mathit{List}) \rightarrow \mathit{Nat}; \\
\text{RELATIONS:} & \mathit{elemi} : (\mathit{List}, \mathit{Nat}, \mathit{Elem}); \\
& \;\lhd\;\; : (\mathit{Elem}, \mathit{Elem}); \\
\text{AXIOMS:} & \neg \mathit{nil} = a.B \wedge (a_1.B_1 = a_2.B_2 \rightarrow a_1 = a_2 \wedge B_1 = B_2)); \\
& H(\mathit{nil}) \wedge (\forall a, J \,.\, H(J) \rightarrow H(a.J)) \rightarrow \forall L \,.\, H(L)\,; \\
& \mathit{nocc}(x, \mathit{nil}) = 0; \\
& a = b \rightarrow \mathit{nocc}(a, b.L) = \mathit{nocc}(a, L) + 1; \\
& \neg a = b \rightarrow \mathit{nocc}(a, b.L) = \mathit{nocc}(a, L); \\
& \mathit{elemi}(L, 0, a) \leftrightarrow \exists B \,.\, L = a.B; \\
& \mathit{elemi}(L, s(i), a) \leftrightarrow \exists b, B \,.\, L = b.B \wedge \mathit{elemi}(B, i, a); \\
\text{THEOREMS:} & \neg \mathit{elemi}(\mathit{nil}, i, x) \\
& \neg \mathit{elemi}(a.\mathit{nil}, s(i), x) \\
& \ldots
\end{array}
$$

The imported symbols of $\mathcal{NAT}$ are not explicitly mentioned, and their intended model is that of $\mathcal{NAT}$.

The intended interpretations of *List* are given by following *parametric domain assumption*:

For every interpretation $Elem^{\mathsf{i}}$ of *Elem*, *List* is the Herbrand Base generated by the list constructors *nil* and $\cdot$, and by $Elem^{\mathsf{i}}$, considered as a set of constants of sort *Elem*.

Every interpretation of *Elem* and $\lhd$ has one expansion that satisfies the parametric domain assumption, and is a model of $\mathcal{LIST}$. In this expansion, *nil* is the empty list; $\cdot$ is the usual list constructor; $nocc(a, L)$ is the number of occurrences of a in L; and $elemi(L, i, e)$ means that the element e occurs in L at position i.

Now we approach the problem of proving correctness. We first give theorems that show the role of the open completion in correctness proofs. Then we discuss the existence of proof-methods based on them. We assume that all specifications are sound. Soundness guarantees that the specifications are consistent with the axioms.

Theorem 6.3. Let $\mathcal{F} = \langle \Sigma, Ax \cup Th, \mathcal{J} \rangle$ *be a framework and* P *be a program with specification* (S_δ, S_π). *If* $Ax \cup S_\delta \cup S_\pi \vdash Ocomp(P)$ *and* P *decides* δ *in every* S_π*-expansion of* $\mathcal{J}$*, then:*

(i) P *is correct in* $\mathcal{F}$ *with respect to* (S_δ, S_π)*;*

(ii) for every S_π*-expansion* $\mathcal{J}_\pi$ *of* $\mathcal{J}$*,* S_δ *is strict in* $\langle \Sigma + \pi, \mathcal{J}_\pi \rangle$.

PROOF. (i) Consider a generic S_π-expansion $\mathcal{J}_\pi$ of $\mathcal{J}$. Let $\mathcal{I}$ be an arbitrary S_δ-expansion of $\mathcal{J}_\pi$. Then $\mathcal{I} \models Ax \cup S_\delta \cup S_\pi$. Since $Ax \cup S_\delta \cup S_\pi \vdash Ocomp(P)$, then $\mathcal{I} \models Ocomp(P)$. Moreover, P decides δ in $\mathcal{J}_\pi$. By Theorem 4.2, P is steadfast in $\mathcal{I}$. Then it is correct with respect to (S_δ, S_π).

(ii) Since $\mathcal{I}$ is arbitrary, P is steadfast in every S_δ-expansion of $\mathcal{J}_\pi$. However, it is easy to see that, if P is steadfast in two expansions $\mathcal{I}_1$, $\mathcal{I}_2$ of $\mathcal{J}_\pi$, then $\mathcal{I}_1 = \mathcal{I}_2$. Therefore S_δ is strict in $\langle \Sigma + \pi, \mathcal{J}_\pi \rangle$. $\square$

(i) is useful for proving correctness. (ii) allows us to use programs to prove that a specification is strict. Moreover, it shows that this theorem cannot be applied to proving correctness with respect to non-strict specifications.

For a non-strict specification, the form of a correctness proof depends on the form of the specification. For example, for correctness with respect to super/sub specifications, we have the following result:

Theorem 6.4. Let $\mathcal{F} = \langle \Sigma, Ax \cup Th, \mathcal{J} \rangle$ *be a framework,* $SupSub_r$ *be the specification:*

$$\forall x \,.\, (R_{sub}(x) \rightarrow r(x)) \wedge (r(x) \rightarrow R_{super}(x)),$$

and P *be a program with specification* $(SupSub_r, S_\pi)$. *If*

(a) $Ax \cup S_\pi \cup \{\forall x \,.\, r(x) \leftrightarrow R_{super}(x)\} \vdash Ocomp^+(P)$
(b) $Ax \cup S_\pi \cup \{\forall x \,.\, r(x) \leftrightarrow R_{sub}(x)\} \vdash Ocomp^-(P)$

and P *decides* r *in every* S_π*-expansion of* $\mathcal{J}$*, then* P *is correct in* $\mathcal{F}$ *with respect to* $(SupSub_r, S_\pi)$.

PROOF. Let $\mathcal{J}_\pi$ be a generic S_π-expansion of $\mathcal{J}$. Let j_π be an interpretation of $\mathcal{J}_\pi$. Let i_{sup} be the $\forall x \,.\, r(x) \leftrightarrow R_{super}(x)$-expansion of j_π and i_{sub} be the

$\forall x \,.\, r(x) \leftrightarrow R_{sub}(x)$-expansion of j_π. By (a), $\mathsf{j}_\pi^{P^+} \subseteq_r \mathsf{i}_{sup}$. By (b), $\mathsf{i}_{sub} \subseteq_r \mathsf{j}_\pi^{P^-}$. Since P decides r in j_π, $\mathsf{j}_\pi^{P^+} = \mathsf{j}_\pi^{P^-}$. Now it is easy to conclude the proof. □

These theorems (and similar ones for other forms of specifications) provide the basis for an effective proof-method of program correctness (with respect to specifications).

As far as $\vdash$ is concerned, an interactive theorem prover for classical logic can be used. In our approach, a very important part would be a mechanism of information retrieval in well organised libraries of theorems, already proved in frameworks. This would lead to more realistic methods for proving correctness and for deriving correct programs.

Proving completion alone is not sufficient. Indeed the above theorems require us to prove that a program decides a relation in a class of interpretations. For this purpose, the results of Section 5 become important. They reduce the problem of proving that P decides δ into one of existential termination in $\mathcal{J}_\pi$ (see Theorem 5.7).

Thus termination proofs play an important role in correctness proofs. However, they are not the main concern of this paper, and we will only give some hints in the following examples of correctness proofs.

Example 6.3. (Correctness in Closed Frameworks)
Consider the closed framework $\mathcal{NAT}$, and the strict specification S_{prod}:

$$S_{prod}: \quad prod(x, y, z) \leftrightarrow z = x * y$$

We can prove:

$$\mathcal{NAT} \cup S_{prod} \vdash prod(x, 0, 0) \tag{6.5}$$

and derive the program $P_0 = \{prod(x, 0, 0) \leftarrow\}$. Clearly P_0 terminates. However, it is only partially correct. In our approach this is revealed by the fact that $Ocomp(P_0, prod)$ cannot be proved from $\mathcal{NAT} \cup S_{prod}$ (it is inconsistent). We can go on and prove:

$$prod(x, s(y), z) \leftrightarrow \exists w \,.\, prod(x, y, w) \wedge z = w + x \tag{6.6}$$

If we introduce the specification S_{sum}:

$$S_{sum}: \quad sum(x, y, z) \leftrightarrow z = x + y$$

we get the the following open program P_1:

$$\begin{array}{lll} p_0: & prod(x, 0, 0) & \leftarrow \\ p_1: & prod(x, s(y), z) & \leftarrow prod(x, y, w), sum(w, x, z) \end{array}$$

Now we can prove:

$$\mathcal{NAT} \cup S_{sum} \cup S_{prod} \vdash Ocomp^-(P_1, prod)$$

Since the open completion $Ocomp^+$ has been proved in the derivation of the clauses, we have obtained the proof of the open completion of P_1. Therefore it remains to prove that P_1 existentially terminates.

To study termination, we consider the *recursive initial paths* of the SLD$_{\mathrm{E}}$-tree with root $prod(x_0, y_0, z_0)$. In this case there is only the recursive path:

$$\begin{array}{l} prod(x_0, y_0, z_0) \xRightarrow{p_1} prod(x_1, y_1, w_1), sum(w_1, x_1, z_1) \\ \qquad\qquad\qquad\qquad\quad x_0 = x_1, y_0 = s(y_1), z_0 = z_1 \end{array}$$

The equation in the last goal is such that:

$$\mathcal{NAT} \cup \{x_0 = x_1, y_0 = s(y_1), z_0 = z_1\} \vdash y_1 < y_0$$

Since $<$ is well-founded in the intended model of $\mathcal{NAT}$, every SLD$_{\mathrm{E}}$-derivation performed by the interpreter of Section 5 is finite, when $prod(x_0, y_0, z_0)$ is the starting goal, the input assignment grounds at least y_0 and the pre-interpretation is the S_{sum}-expansion of the intended model.

This entails existential termination in the (unique) S_{sum}-expansion. Thus, by Theorem 6.3, we have proved that P_1 is correct with respect to $(\{S_{prod}\}, \{S_{sum}\})$.

By Theorem 6.1, P_1 correctly composes with every program that is correct with respect to S_{sum}.

Example 6.4. (Correctness in Open Frameworks)
Consider the open framework $\mathcal{LIST}(Elem, \lhd)$, and the parametric specification $(\{S_{chain}(Elem, \lhd, elemi)\}, \{S_{linked}(Elem, \lhd)\})$:

$$\begin{array}{ll} S_{chain}(Elem, \lhd, elemi) & : chain(L) \leftrightarrow \\ & \quad (\forall i \,.\, elemi(L, i, x) \wedge elemi(L, s(i), y) \rightarrow x \lhd y) \\ S_{linked}(Elem, \lhd) & : linked(x, y) \leftrightarrow x \lhd y \end{array}$$

We can prove in $\mathcal{LIST}(Elem, \lhd)$ the open completion of the following program P_{chain}:

$$\begin{array}{lll} c_1 : & chain(nil) & \leftarrow \\ c_2 : & chain(x.nil) & \leftarrow \\ c_3 : & chain(x.L) & \leftarrow L = y.A, linked(x, y), chain(L) \end{array}$$

We omit the proof, and only remark that the first two theorems listed in the framework are useful for proving the first two clauses.

Concerning termination, we have the recursive path:

$$chain(L_0) \stackrel{c_3}{\Longrightarrow} L_1 = y_1.A_1, linked(x_1, y_1), chain(L_1), L_0 = x_1.L_1$$

The equation in the last goal is such that we can prove in the framework that $len(L_1) < len(L_0)$. Since $<$ is well-founded on natural numbers, we get existential termination in the S_{linked}-expansions of the intended models of $\mathcal{LIST}(Elem, \lhd)$.

As we have seen in the case of *prod*, our correctness analysis can be done in a modular way, i.e., we can study *prod* and *sum* separately. This works for logic programs. For real Prolog programs, however, this does not necessarily mean that termination is preserved. Indeed, our notion of correctness is model-theoretic, while in Prolog there is a discrepancy between operational and model-theoretic semantics. Moreover, we may require termination with respect to non-ground goals. These problems are important (see e.g. [30]), but they are beyond the scope of this paper. We will only briefly comment on them in Section 7.

Finally, we remark that the theorems of a framework are useful in the proof of the completion. For example, distributivity is useful to prove (6.6), and the first two theorems given in the framework for lists are useful to prove the first two clauses of P_{chain}. Moreover, frameworks are also useful in the analysis of termination. In both the examples, we have proved termination using only the properties of the framework. This approach to termination works as well for constraint logic programs, of course by using suitable frameworks.

7. RELATED WORK

We now briefly summarise the key features of steadfastness, and compare them with related work.

7.1. Open Logic Programs

An open steadfast program is an open Σ-program. The semantics of an open program P is given by the j-models j^{P^+} and j^{P^-}, and steadfastness in a framework $\mathcal{F} = \langle \Sigma, \mathcal{I} \rangle$ intuitively means parametric correctness.

Open logic programs have been considered by Brogi, Lamma and Mello. In [3], they introduce Admissible Herbrand Models, to provide a model-theoretic semantics for program composition.

Admissible models can be seen as a particular case of j-models, where only suitable Herbrand pre-interpretations j are considered. By contrast, in our approach, we place no restriction on j, since we need to compare j-models with the interpretations of an arbitrary framework. That is, whereas [3] is concerned with a model-theoretic semantics for program composition, our primary goal is 'open correctness' in a class of interpretations. Modularity is a by-product of steadfastness, since the latter is preserved by composition.

Moreover, allowing arbitrary pre-interpretations yields a greater generality. For example, we can switch to constraint logic programs without difficulties, and we have a greater freedom in managing signatures, signature morphisms, and signature composition, since reachability is not required for j.

7.2. Modularity

A steadfast program makes an ideal module, since it is parametrically correct and can be composed with different (correct) parameters into different correct programs.

In logic programming, an in-depth survey of modularity can be found in [5]. There are broadly two main approaches: one based on an algebra for logic program composition, and one based on extensions of Horn logic.

The first approach is exemplified by Brogi, Mancarella, Pedreschi and Turini [4]. Here composition operators and their semantics are defined for composing programs. Compositional semantics is based on the immediate consequence operator, although in other papers (e.g. [3]) a compositional model-theoretic semantics is used.

The second approach is exemplified by Miller [27], who extends Horn clauses by allowing implication goals whose bodies are sets of clauses defined in an external module. Monteiro and Porto [29] use a similar idea to introduce *contextual logic programming*. Modularity can also be achieved by introducing types [31]. In general, enriching first-order Horn logic yields scoping and abstraction mechanisms that support modules, data abstraction, and metaprogramming. This has been nicely explained in [28], where different extensions of first-order Horn logic are considered, and the power of higher-order logic programming is briefly discussed.

However, in the above approaches, there is no notion of formal correctness like steadfastness, which ensures that the composed program is correct with respect to the intended interpretations of the framework. Rather, these approaches study module abstraction and composition at a syntactic and/or semantic level, but do not compare programs with (separate) specifications: the component modules are 'correct by definition', and the composed program is also 'correct by definition', i.e., the composition rules are assumed to be 'correct by definition'. Whereas steadfast program composition guarantees *a priori correctness* with respect to a specification S in a framework, the correctness of any program composed using the above approaches with respect to S must be established *a posteriori*.

7.3. Parametric Specifications

Specification frameworks allow us to introduce parametric specifications in a simple and natural way, together with accompanying benefits like modularity.

However, such specifications (and their benefits) can also be (and indeed have been) captured by other formalisms, such as algebraic specifications (e.g. [10, 11, 38]), and institutions [12]. The latter have even been applied to logic programs [33]. In functional programming, Sanella and Tarlecki [35, 34] have proposed formal development of (parametric) ML programs from algebraic specifications. Their work is very close to ours in spirit, and allows parametric specifications and programs.

In our approach, however, we always distinguish between frameworks, specifications and programs. A framework $\mathcal{F}$ defines a general problem domain, and specifications in $\mathcal{F}$ represent computational tasks in this domain. At the framework level, our approach [15] is very similar to the algebraic one, except that we use full first-order languages, since we require their expressiveness at the specification level. At the specification and program levels, we introduce a notion of model-theoretic correctness, based on steadfastness, which is the distinctive feature of our approach. By this three-level approach, we can put in the specification only what is needed for the program, while maintaining our relevant knowledge of the problem domain in the framework. This allows us to introduce in a clear way both strict and non-strict specifications, and to define the corresponding notions of correctness. A discussion of this can be found in [22].

7.4. Correctness and Termination

We define correctness, in a declarative, model-theoretic manner, with respect to both strict and non-strict parametric specifications. This correctness is defined in terms of steadfastness and thus 'inherits' the latter's compositional properties. We have also discussed methods for proving correctness, involving the open completion and existential termination.

Correctness of logic programs has been widely studied (see Chapter 8 of [2] for instance). What distinguishes our approach is the presence of a class of interpretations, wherein specifications assume their proper meaning, and wherein correctness must be established. From this point of view, Deville's [9] 'correctness with respect to separate specifications' is more similar than most to our approach in spirit. Even here, however, there is an important difference in that our characterisation is a more abstract, model-theoretic characterisation, based on steadfastness. Our general results can be used to introduce proof methods, based on the open completion and existential termination.

There are other kinds of (strict and non-strict) specifications, besides those in Section 6. In general, different kinds of specifications may involve different kinds of termination proofs (see e.g. [21]). Using model-theoretic correctness, however, may cause problems with a real logic programmming language such as Prolog, owing to the discrepancy between the model-theoretic semantics and the operational semantics of the latter. For 'real' programs, we have to take into account depth-search strategy with a given computation rule. Thus, for practical program development we might distinguish two phases in the derivation of a correct program. In the first one, we derive a semantically correct program, using the results of this paper. Then in the second, we adapt the program for execution by a specific interpreter. This is not discussed in this paper, and will most probably involve transformation techniques.

To deal with termination, well-known techniques [8, 2] can be adapted for open programs. To handle modularity, and the effects of the computation rule and possible delay mechanisms, results like those given for example in [30] should be useful.

8. CONCLUSION

In this paper, we have introduced the semantics of steadfast logic programs, and we have studied their compositional properties, in an abstract, model-theoretic way. We have also introduced an operational semantics for open programs, based on a 'virtual interpreter', in the manner of constraint logic programs [14]. Our treatment of steadfastness, given for definite logic programs, immediately extends to constraint logic programs. Moreover, it can be easily extended to terminating normal programs. In fact, we have extended a kernel of the theory of (closed) logic programs [37, 7, 25, 1] to open logic programs. Thus, we have established soundness, and completeness results for open logic programs.

We have demonstrated the application of steadfastness to correctness with respect to parametric specifications in Section 6, and in the previous section we have briefly discussed related pieces of work and their differences with our approach.

The idea of steadfastness is primarily motivated by our work in deductive synthesis [24, 17, 18] of logic programs. However, we believe that it is an important concept for formal software development in general. Indeed, steadfastness is a combination of modularity, reusability, and (formal) correctness. As such it provides a basis for hierarchical formal program development that can be used to construct not only correct parametric modules but, more importantly, also libraries of such modules that can be composed correctly. Steadfast programs can be reused *correctly* with different parameters in different instances of an open framework.

Our next step is to return to our original motivation, and design a synthesis method that is based on steadfastness. For such a method, the object-oriented technology would be an efficient vehicle for implementing reusability of specifications, theorems and steadfast programs. Reusability is of crucial importance, since the effort of deriving a correct program (together with its formal proof) is much higher than that of simply writing it (and hoping it is correct). In particular, knowledge reuse (by framework-libraries of steadfast programs and theorems) would make program synthesis a more realistically achievable task. For example, a lot of useful theorems have been proved on numbers, lists, trees, arrays, and so on. Their reuse is allowed by frameworks. When dealing with a specific problem domain, we set up the appropriate specification framework by importing the frameworks for the necessary data types, and inheriting their theorems. These theorems are useful for writing down the right specifications and to reason about program correctness and synthesis.

Equally we believe that steadfastness is a contribution to object-oriented programming. The reusability afforded by steadfastness differs from that found in object-oriented programming (e.g. [26]) mainly in two ways.

Firstly, steadfastness characterises reusability at a high level of abstraction, i.e., at the level of the framework. In contrast, in object-oriented programming, code reuse occurs at program level, either by placing common code in a superclass that is shared by subclasses, or by reusing the same code for a method in subclasses that all inherit the same superclass. In other words, steadfastness provides a kind of declarative semantics for reusability, whereas in object-oriented programming, such a semantics is lacking. Nevertheless, object-oriented programming would be an efficient vehicle for implementing declarative reusability.

Secondly, steadfastness guarantees *correct* reuse. Here, correctness is defined formally with respect to the given framework. In contrast, in object-oriented programming, correctness is usually only defined informally. Thus, steadfastness could provide the basis for developing libraries which are (formally) correct with respect to frameworks. This would be a useful extension to current object-oriented technology, especially in applications where correctness and safety are critical.

In summary, steadfastness captures a kind of correctness that is preserved through inheritance hierarchies. As such, it provides the essential criterion for formal correctness for the development of next generation *component-based* as well as *distributed* software. Another direction of our future work will be to provide more evidence to substantiate this claim, by considering the following aspects.

Since steadfastness defines *a priori* correct reuse, it should provide guidance for choosing the right components to construct a specified composite. Steadfast programs make ideal units in a library of components, and can also be used as distributed programs, with the advantage that they are *a priori* correct.

Constructing steadfast programs should also contribute directly to component-based software development. This is because steadfastness is defined in the context of a framework formalising the problem domain. Our formalisation of a framework corresponds to what is called an *Object-oriented Design* (*OOD framework* [23]) in component-based software development. Thus developing steadfast programs entails constructing (libraries of) OOD frameworks. It is now widely recognised that such frameworks are a more promising answer to software reuse than objects, and industry is increasingly using (OOD) frameworks rather than objects as the unit of reuse.

Acknowledgements

We wish to thank the referees for their comments and constructive suggestions that have enabled us to improve this paper. Sten-Åke Tärnlund thanks Professors Luis Monteiro, Luis Pereira, and Antonio Porto.

REFERENCES

1. K.R. Apt. Logic Programming. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science* Vol. B, pages 493-574, Elsevier, 1990.
2. K.R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.
3. A. Brogi, E. Lamma and P. Mello. Compositional model–theoretic semantics for logic programs. *New Generation Computing* 11:1-21, 1992.
4. A. Brogi, P. Mancarella, D. Pedreschi and F. Turini. Modular logic programming. *ACM TOPLAS* 16(4):1361-1398, 1994.
5. M. Bugliesi, E. Lamma and P. Mello. Modularity in logic programming. *J. Logic Programming* 19,20:443-502, 1994. Special issue: Ten years of logic programming.
6. K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293-322. Plenum Press, 1978.
7. K.L. Clark. Predicate Logic as a Computational Formalism. Research Report 79/59, Dept of Computing, Imperial College, 1979.
8. D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *J. Logic Programming* 19,20:199-260, 1994.
9. Y. Deville. *Logic Programming: Systematic Program Development.* Addison-Wesley, 1990.
10. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*. Springer-Verlag, 1985.
11. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2*. Springer-Verlag, 1989.
12. J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM* 39(1):95-146, 1992.
13. C.J. Hogger. *Introduction to Logic Programming.* Academic Press, 1984.
14. J. Jaffar and M. Maher. Constraint logic programming: A survey. *J. Logic Programming* 19,20:503-581, 1994. Special issue: Ten years of logic programming.
15. C. Kreitz, K.-K. Lau, and M. Ornaghi. Formal reasoning about modules, reuse and their correctness. In D.M. Gabbay and H.J. Ohlbach, editors, *Proc. Int. Conf. on Formal and Applied Practical Reasoning*, *LNAI* 1085, pages 384-399, Springer-Verlag, 1996.
16. K.-K. Lau, C.D.M. Moss and M. Ornaghi. Formal development of correct classes in computational logic. In D.J. Duke and A.S. Evans, editors, *Proc. BCS-FACS Northern Formal Methods Workshop*, Springer-Verlag, *electronic* Workshops in Computing Series, 1997. Abstract on page 3. (http://ewic.springer.co.uk)
17. K.-K. Lau, M. Ornaghi and S.-Å. Tärnlund. The halting problem for deductive synthesis of logic programs. In P. van Hentenryck, editor, *Proc. 11th Int. Conf. on Logic Programming*, pages 665-683, MIT Press, 1994.
18. K.-K. Lau and M. Ornaghi. On specification frameworks and deductive synthesis of logic programs. In L. Fribourg and F. Turini, editors, *Proc. LOPSTR 94 and META 94*, *LNCS* 883, pages 104-121, Springer-Verlag, 1994.
19. K.-K. Lau and M. Ornaghi. A formal approach to deductive synthesis of constraint logic programs. In J.W. Lloyd, editor, *Proc. 1995 Int. Logic Programming Symp.*, pages 543-557, MIT Press, 1995.

20. K.-K. Lau and M. Ornaghi. Towards an object-oriented methodology for deductive synthesis of logic programs. In M. Proietti, editor, *Proc. LOPSTR 95*, *LNCS* 1048, pages 152-169, Springer-Verlag, 1996.

21. K.-K. Lau and M. Ornaghi. Forms of Logic Specifications: A Preliminary Study. In J. Gallagher, editor, *Proc. LOPSTR 96*, *LNCS* 1207, pages 295-312, Springer-Verlag 1997.

22. K.-K. Lau and M. Ornaghi. The relationship between logic programs and specifications – the subset example revisited. *J. Logic Programming* 30(3):239-257, 1997.

23. K.-K. Lau, M. Ornaghi, and A. Wills, Frameworks in *Catalysis*: Pictorial Notation and Formal Semantics, *Proceedings of the First IEEE International Conference on Formal Engineering Methods*, pages 213-220, IEEE Computer Society, 1997.

24. K.-K. Lau and S.D. Prestwich. Synthesis of a Family of Recursive Sorting Procedures. In V. Saraswat and K. Ueda, editors, *Proc. 1991 Int. Logic Programming Symposium*, pages 641-658. MIT Press, 1991.

25. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

26. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

27. D. Miller. A logical analysis of modules in logic programming. *J. Logic Programming* 6:79-108, 1989.

28. D. Miller. Abstractions in logic programs. In P. Odifreddi, editor, *Logic in Computer Science*, pages 329-356, Academic Press, 1990.

29. L. Monteiro and A. Porto. Contextual Logic Programming. In G. Levi and M. Martelli, editors, *Proc 6th Int. Conf. on Logic Programming*, pages 284-299, MIT Press, 1989.

30. L. Naish. Coroutining and the construction of terminating logic programs. *Australian Computer Science Comm.* 15(1):181-190, 1993.

31. F. Pfenning. *Types in Logic Programming*. MIT Press, 1992.

32. R. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

33. F. Orejas, E. Pino and H. Ehrig. Institutions for Logic Programming. *Theoretical Computer Science* 173:485-511, 1997.

34. D. Sannella, S. Sokolowski and A. Tarlecki. Towards formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica* 29:689-736, 1992.

35. D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: model-theoretic foundations. *Lecture Notes in Computer Science* 623, pages 656-671, Springer-Verlag, 1992.

36. J.C. Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19-88. Morgan Kaufmann, 1988.

37. M.H. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *J. ACM* 23(4):733-742, October 1976.

38. M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675-788. Elsevier, 1990.