Relating Logic Programming and Propositions-as-Types: A Logical Compilation

SUMMARY

James Lipton
Dept. of Mathematics
University of Pennsylvania

June 1, 1993

Abstract

We analyze logic programs (Horn Clause programs and extensions) informally as specifications or types in the sense of the Curry-Howard isomorphism, rather than as programs. Using a realizability interpretation we develop a translation of these clauses to equational specifications. These give rise to various ways of associating non-deterministic terms or computations with this type that satisfy the logic program as a specification. By adapting a 1956 result of Nerode, (generalizing Herbrand-Gödel computability to term models) we are able to solve the equational specification directly over the term model, and produce a (multivalued) Turing-machine index for the solution. We also define a realizability semantics in a partial applicative structure over the Herbrand Universe, and consider a non-deterministic or disjunctive \( \lambda \)-calculus. All interpretations are independent of any choice of logic programming interpreter. Rather, they transfer control and implementation features of proof search to control and implementation features of the recursion theorem. Completeness of the former is mapped to correctness of the latter.

Our translation provides a framework for integrating logic programming directly into a typed or untyped functional programming environment in a way that preserves the Curry-Howard content of typing. Our interpretations also provide new completeness theorems for various classes of logic programs. The results apply to Horn Logic Programs, and the Miller-Scedrov-Nadathur Uniform logic programming languages.

1 Introduction: Logic Programming as a specification Language

The results in this paper come from exploiting a largely unused uniformity that is present in most logic programming paradigms: the extraction of a witness to a query is uniform in the parameters and predicates in the program. By explicitly rewriting a logic program as a realizability goal, we search not for a specific witness, but a function that returns this witness for every choice of parameters. This reformulation is done first in the framework of an abstract form of the recursion theorem, adapted from [30], and then in terms of realizability over Feferman’s partial applicative structures, where we can formalize logic programming languages, and where we are guaranteed the existence of the required partial recursive function via the appropriate fix-point theorem as in [1].

What we obtain is a compilation of logic programs into an equational specification of functional code via a generalized logic programming paradigm. The logic program is treated as a type, the computation process
as one of finding a uniform inhabitant for the type. The entire development is implementation independent: we obtain not a particular evaluation strategy but a specification which gives different functional solutions according to different evaluation sequences. We also define a multivalued realizability which captures the nondeterminism of the declarative program directly. In this outline we develop one of these paradigms in detail, with an example, and give a quick sketch of the remaining ones.

1.1 Towards a Realizability Semantics for Logic Programs

In this section we develop an informal notion of realizability for logic programs. We begin with a logic program \( \mathcal{P} \) decorated with abstract realizers \( \alpha_i \), which are taken as atomic evidence that the clauses are true (because the programmer says so). We first consider the first-order case: Horn or First Order Hereditarily Harrop logic programs, whose syntax is defined as follows. Let

\[
q := \top | a | q_1 \wedge q_2 | q_1 \vee q_2 | \exists x q \\
h := a \rightarrow a | h_1 \wedge h_2 | \forall x h \\
g := \top | a | g_1 \wedge g_2 | g_1 \vee g_2 | \exists x g | \forall x g | f \rightarrow g \\
f := a \rightarrow a | f_1 \wedge f_2 | \forall x f
\]

where \( a \) stands for an atomic formula. Then a Horn clause program is a finite set of closed \( h \)-formulas, and a query (or goal) for such a program is a \( q \)-formula (or a finite set of them). A First Order Harrop program is a finite set of closed \( f \)-formulas, and a query (or goal) for such a program is a \( g \)-formula (or a finite set thereof).

In the first part of this paper we will be considering an arbitrary Horn or FOHH program

\[
\alpha_1 : h_1(s_1) \leftarrow T_1(t_1) \\
\cdots \\
\alpha_m : h_m(s_m) \leftarrow T_m(t_m)
\]

where, \( s_i \) and \( t_i \) are \( -\)tuples of terms and where each clause is "decorated" with (abstract) realizers \( \alpha_1, \cdots, \alpha_m \). These realizers supply atomic evidence that the corresponding clauses are being considered true.

Realizability over the Herbrand Universe Our first approach to extracting functional content from the program \( \mathcal{P} \) defined above in (1) together with a query \( Q[\bar{u}, \bar{v}] \) where the \( \bar{u} \) are parameters and the \( \bar{v} \) are variables, is to inhabit the Horn Clause specification directly with a realizor over the Herbrand Universe. This means finding a term \( e \) from a suitable partial combinatory structure \( E(\mathcal{H}) \) defined over the Herbrand universe of the program satisfying

\[
e : \forall \bar{u} \exists \bar{v} [ \mathcal{P} \rightarrow Q(\bar{u}, \bar{v})].
\]

As will be shown below, this entails finding a function \( \bar{e} \) which, on a suitable domain \( D \), satisfies the specification

\[
(\forall \bar{u}) \ ( \mathcal{P} \rightarrow Q[\bar{u}, \bar{e}(\bar{u})]).
\]

We show how to produce a series of recursion equations specifying \( \bar{e} \) from (2). Variants of Kleene's [17] and Nerode’s [30] recursion theorem guarantee the existence of a solution in the set of partial recursive functions. We start with an example that shows how we obtain a specification for a multivalued function which captures the full non-determinism of the original declarative program seen independently of any choice of an interpreter.

1.1.1 An non-deterministic Example

We consider the realizability interpretation and the induced translations for the following Horn clause program, \( \mathcal{P} \), equipped with "dummy realizers":

\[
\alpha_1 : h_1(s_1) \leftarrow T_1(t_1) \\
\cdots \\
\alpha_m : h_m(s_m) \leftarrow T_m(t_m)
\]
\[
\alpha : \text{add}(0, x, x). \quad (3) \\
\beta : \text{add}(s(x), Y, s(Z)) : = \text{add}(X, Y, Z). \quad (4)
\]

Which means (see realizability definitions (2,5), below)
\[
(\forall x) \alpha x : \text{add}(0, x, x) \quad (5)
\]
\[
(\forall x)(y)(z)(f) f : \text{add}(x, y, z) \to (\beta xyz) f : \text{add}(s(x), y, s(z)) \quad (6)
\]

We trace through the compilation of this program to a specification of the non-deterministic function
\[
f : \mathbb{N} \to \mathbb{N} \times \mathbb{N} \text{ satisfying}
\]
\[
f(x)_0 + f(x)_1 = x
\]
First, we translate this program to the following realizability goal, assigning what we will call below a \((001)\) template to the predicate \(\text{add}\), that is to say, the character of an input to the third variable, and of an output to the first two:
\[
\exists e : \forall u \exists v \exists w \ \mathcal{P} \to \text{add}(v, w, u)
\]
Unravelling this according to the definitions of realizability given below (with the notation \(g\) and \(e\) for left and right projections of \(e\)),
\[
(\forall u)[\text{gu} : \mathcal{P} \to \text{add}(\hat{e}_0 u, \hat{e}_1 u, u)]
\]

i.e.
\[
(\forall u)(\forall \gamma)(\gamma : \mathcal{P} \to \text{gu} \hat{\gamma} : \text{add}(\hat{e}_0 u, \hat{e}_1 u, u))
\]

Taking \(\gamma\) to be \(\langle \alpha, \beta \rangle\) from (5) and (6) above, we have
\[
\text{gu} \langle \alpha, \beta \rangle : \text{add}(\hat{e}_0 u, \hat{e}_1 u, u) \quad (7)
\]
Now, unifying (7) on the third variable (that is to say, unifying on the template \((001)\)) with the first clause of the original program (5), we have:
\[
\alpha u : \text{add}(0, u, u) \quad (8)
\]
\[
\text{gu} \langle \alpha, \beta \rangle : \text{add}(\hat{e}_0 u, \hat{e}_1 u, u) \quad (9)
\]

which has a solution \(\hat{e}\) if we choose:
\[
\hat{e}_0 u = 0 \quad \text{and} \quad \hat{e}_1 u = u \quad (10)
\]
as well as \(\text{gu} \langle \alpha, \beta \rangle = \alpha u\). Now, assuming an \(e\) exists satisfying (7) we have, by applying the second clause (5) of the original program:
\[
\forall u : \beta[\hat{e}_0 u][\hat{e}_1 u][u][\text{gu} \langle \alpha, \beta \rangle] : \text{add}(s(\hat{e}_0 u), \hat{e}_1 u, s(u)) \quad (11)
\]

and applying (7) to \(s(u)\)
\[
\forall u \in s(u) \langle \alpha, \beta \rangle : \text{add}(\hat{e}_0 s(u), \hat{e}_1 s(u), s(u)) \quad (12)
\]

where the choice of argument in (11) and (12) is dictated by unifying (7) and (6) on the last variable. Now (12) has a solution if \(\hat{e}\) satisfies the recursive equations
\[
\hat{e}_0 (s(u)) = s(\hat{e}_0 u) \quad \text{and} \quad \hat{e}_1 (s(u)) = \hat{e}_1 u. \quad (13)
\]
as well as the corresponding condition $\zeta(su)\langle \alpha, \beta \rangle = \beta[\zeta_0u][\zeta_1u][\zeta_2u(\alpha, \beta)]$ for $\zeta$. Leaving aside for the moment the requirements for the evidence $\zeta$, we have the following conditions for $\zeta$:

$$\begin{align*}
\zeta_0u &= 0 \\
\zeta_1u &= u \\
\zeta_0(s(u)) &= s(\zeta_0u) \\
\zeta_1(s(u)) &= \zeta_1u
\end{align*}$$

We must be careful about how we deal with the multivalued character of these equations. Taken at face value they logically imply the collapse of the underlying Herbrand Universe, since, for every $u$, we have $0 = \zeta_0u = u$.

For this reason, we interpret different conditions for $\zeta_0$ and $\zeta_1$ disjunctively. Before delving into our formalized treatment below, we give an informal treatment. Using $\setminus$ for (non-deterministic) disjunction, we can write this specification somewhat in the style of ML as:

$$\begin{align*}
\text{fun } \zeta 0 &= \langle 0, 0 \rangle \\
\zeta \text{ su} &= \langle 0, su \rangle \setminus \langle s(\zeta_0u), \zeta_1u \rangle
\end{align*}$$

There is a natural execution model for this specification corresponding to every complete proof search strategy for the original program, for example a breadth-first traversal of the induced computation tree, until a leaf is found in normal form.

We also obtain a solution to this which captures the non-determinism of the specification itself by lifting the solution $\zeta$ to a set-valued function $\zeta$ (i.e. to the non-deterministic monad $[\_\_\_]$) in the obvious way. Let $\text{set}$ be the canonical lifting of functions $f : D \to D$ to maps $\text{set}(f) : \wp(D) \to \wp(D)$ between the power sets, given by:

$$\text{set}(f)(X) = \{f(x) : x \in X\}.$$ 

Then $\zeta : D \to \wp(D)$ is given by

$$\begin{align*}
\zeta 0 &= \{\langle 0, 0 \rangle\} \\
\zeta \text{ su} &= \text{set}(\lambda x.\langle 0, su \rangle \setminus \langle s(\zeta_0u), \zeta_1u \rangle)(\zeta u) \\
&= \text{set}(\lambda x.\langle 0, su \rangle)(\zeta u) \cup \text{set}(\lambda x.\langle s(\zeta_0u), \zeta_1u \rangle)(\zeta u)
\end{align*}$$

where $p_0$ and $p_1$ are left and right components of the pairs in $\zeta u$.

These approaches suggest several computational and realizability formalisations, each with its own fixpoint or recursion theorem. We sketch one below adapted from Nerode’s 1956 dissertation, which is in some sense independent of implementation (i.e. developed in terms of indices). Several concrete developments of the lambda calculus along these lines are sketched in the appendix.

### 1.2 Nerode-Kleene computability over term-algebras

The following development, adapted from Nerode’s dissertation, provides an immediately applicable framework for generating and solving equations along the lines of the example studied above.

**Definition 1.1** A recursion calculus (a finite-signature one-sorted equational calculus) is a triple $e = (V, F, C)$ where $V$ is a set of variables, and $F$ and $C$ are finite, nonempty sets of functions and constant symbols, respectively. We will also call $(F, C)$ the e-signature.

The word e-algebra $W_e$ is the Herbrand Universe of ground terms for the recursion calculus $e$.

A finite set $A$ of equations in the recursion calculus $e = (V, F, C)$ is said to overlay the word e-algebra $W$ if $e = (V, F, C)$ and $F \subseteq F'$. We say such a set of equations distinguishes $W$ if for any $w_1, w_2 \in W$,

$$A \vdash w_1 \equiv w_2 \quad \Rightarrow \quad W \vdash w_1 \equiv w_2$$

We say $A$ is complete for $W$ if for any function letter $f \in F'$ of arity $k$ and any $w_1, \ldots, w_k \in W$ there is a $w_{k+1} \in W$ such that

$$A \vdash f(w_1, \ldots, w_k) = w_{k+1}$$
We say $A$ is a partial definition over $W$ if $A$ overlays and distinguishes $W$. $A$ is a definition if in addition it is complete.

**Definition 1.2** If $A$ is a (partial) definition over the word algebra $W$ and $f$ is a $k$-ary function symbol in $A$, we define the (partial) function

$$[[f]]_A : W^k \rightarrow W$$

by $[[f]]_A(w_1, \ldots, w_k) = w_{k+1} \iff A \vdash f(w_1, \ldots, w_k) = w_{k+1}$.

It is easy to see that if $A$ is a partial definition then $[[f]]_A$ is well-defined, and is total if $A$ is complete over $W$.

**Definition 1.3** The (partial) function $g : W^k \rightarrow W$ is definable if there is a (partial) definition $A$ with a function symbol $f$ of arity $k$ such that $[[f]]_A = g$.

We now extend the notion of recursive function to a term algebra in a completely straightforward manner:

**Definition 1.4** Let $e = (F, C)$ be a finite signature, where $F$ and $C$ are ordered sets. Then the similarity type of $e$ is a finite-support sequence of natural numbers $\tau$ with $\tau(n) = m$ iff there are $m$ function symbols of-arity $n$. We define a Gödel numbering of the term algebra $W_e$ (in fact the Gödel numbering induced by any encoding of sequences in $N$ and by the ordering of $F$ and $C$) to be a function $\# : W \rightarrow N$ given by the standard encoding $\# f(t_1, \ldots, t_n) = \text{Seq}(\alpha(f), \# t_1, \ldots, \# t_n)$. A numbering of $W_e$ is a bijection $I : N \rightarrow W_e$ such that such that there are recursive bijections $\alpha$ and $\beta$ with $\beta \circ \# = I^{-1}$ and $\alpha \circ I^{-1} = \#$.

**Definition 1.5** Let $I$ be a numbering of the word algebra $W$. The (partial) function $g : W^k \rightarrow W$ is recursive over $W$ if there is a recursive (partial) function $f : N^k \rightarrow N$ such that the following diagram commutes:

\[
\begin{array}{ccc}
N^k & \xrightarrow{f} & N \\
\downarrow{I^k} & & \downarrow{I} \\
W^k & \xrightarrow{g} & W
\end{array}
\]

We then have the following theorem, proven in [30]:

**Theorem 1.6** The (partial) function $g : W^k \rightarrow W$ is definable iff it is recursive over $W$.

The proof also shows that (an index for) the associated partial function $f : N \rightarrow N$ can be uniformly effectively computed from (codes for) the equations in a way similar to Kleene's original development of Herbrand-Gödel computability for partial recursive functions.

### 1.3 A nondeterministic term-model extension

Let $e$ be a recursion calculus, and $W_e$ its term model. Then the term model $W_p$ of the calculus $e_p = (F \cup \{\text{par}\}, C \cup \{\text{fail}\})$ obtained by adding one distinguished binary function symbol $\text{par}$ and the constant $\text{fail}$ is called the pair-extension of $W_e$. We call the set of equations $A$ overlaying $W_e$ a pair-extension theory if the signature of $A$ includes the function symbol $\text{par}$ and the constant symbol $\text{fail}$, and if $A$ includes the following equations.

1. (associativity of $\text{par}$)
   \[ \text{par}(\text{par}(x, y), z) = \text{par}(x, \text{par}(y, z)) \]
2. (failure) \[ \text{pair}(\text{fail}, x) = \text{pair}(x, \text{fail}) = \text{pair}(x, x) \]

3. (lifting) for each \( f \in \hat{F} \) other than \( \text{par} \) of \(-\)-arity \( n > 0 \)
\[ \begin{align*}
\text{pair}(x_1, \ldots, x_{j-1}, \text{pair}(y, z), x_{j+1}, \ldots, x_n) & = \\
\text{pair}(f(x_1, \ldots, x_{j-1}, y, x_{j+1}, \ldots, x_n), f(x_1, \ldots, x_{j-1}, z, x_{j+1}, \ldots, x_n))
\end{align*} \]

**Remark:** A slight reformulation of the above will give us a syntactic variant of the non-determinism triple or monad construction (e.g. [27]). Let \( \mathcal{C} \) be the category of (partial) \( \epsilon \)-algebras and algebra homomorphisms. Observe that if \( B \) is an \( \epsilon \)-algebra then so is the algebra \( B_0 \) whose carrier \( |B_0| \) is given by the set of terms
\[ \{ \text{pair}(a_1, \ldots, a_m) : a_i \in |B| \cup \{ \text{fail} \} \} \]
where \( \text{pair}(a_1, \ldots, a_m) \) is shorthand for \( \text{pair}(a_1, \text{pair}(a_2, \ldots, \text{pair}(a_{m-1}, a_m) \ldots)) \). The algebra operations are induced by lifting (as given in the equations above), with constants \( a \) interpreted as \( \text{pair}(a, a) \). Let \( \text{T} : \mathcal{C} \to \mathcal{C} \) be a functor defined by \( \text{T}(B) = B_0 \), with \( \text{T} \)'s action on morphisms induced by the corresponding notion of lifting. Then let \( \mu(B) : \text{T}^2(B) \to \text{T}(B) \) be given by
\[ \text{pair}(\text{pair}(a_1, \ldots, a_m), b) \mapsto \text{pair}(a_1, \ldots, a_m, b) \]
and \( \eta(B) : B \to \text{T}(B) \) by \( a \mapsto \text{pair}(a, a) \) or \( \text{pair}(a, \text{fail}) \). Then \( (T, \eta, \mu) \) is a monad.

The aim of this term-model construction is to reformulate equations of the sort produced in the example above (e.g. 14) which failed to distinguish the program signature, i.e. led to a collapse of the underlying Herbrand universe, as equations over a pair-extension which constitute a legitimate partial definition over that extension. All that needs to be shown is that all the necessary definitions used in theorem 1.6 lift to the new structure.

**Lemma 1.7** Let \( I \) be a numbering of an \( \epsilon \)-term algebra \( W_\epsilon \). Then there is a numbering \( I_p \) of its associated pair-extension and a pair of recursive injections \( \alpha \) and \( \beta \) from \( \mathbb{N} \) to \( \mathbb{N} \) such that for all \( a \) in \( W_\epsilon \)
\[ I^{-1}_p(a) = \alpha I^{-1}_p(a) \quad \text{and} \quad I^{-1}_p(a) = \beta I^{-1}(a) \]
Since the pair-extension of \( W_\epsilon \) is just another term-algebra, the main theorem 1.6 states that a function is definable over the pair-extension of \( W_\epsilon \) iff it is partial recursive.

### 1.3.1 Multivalued Turing Machines

In the next section we define a procedure which when applied to a logic program and uniform query \( \mathcal{P} \to Q \) produces a set of equations \( A \) which give partial definition over the pair-extension of the Herbrand universe of \( \mathcal{P} \). By the main theorem of this section, every \( A \)-defined function \( f \) has an associated Turing-machine index \( e_f \) in the sense of definition (1.5). We now define the associated multivalued Turing machine \( R_f \) in the obvious way:
\[ R_f(n) = \text{pair.set}(e_f(n)) \]
where
\[ \text{pair.set}(n) = \{ I^{-1}(a) : a \in I_p(n) \} \]
and where the relation \( \in \) is given by
\[ x \in \text{pair}(y, z) \iff x = y \lor x \in z. \]
Our development is not particularly different than defining \( \text{r.e.} \) set valued recursive functions by associating, with any partial recursive \( f : \mathbb{N} \to \mathbb{N} \), the function \( F : \mathbb{N} \to \mathbb{N} \) given by \( F(x) = S_e \) where \( S_e \) is the domain of the \( e \)-th Turing machine. The only significant difference is that the numbering of sets is more natural since it is induced by (Gödel numbering of) terms used to defined sets of values.
generic variables

We must also deal with fresh constants or variables introduced by the *generic* step for SLDR-computation (see below) or by equations induced by *unbalanced* programs: those with occurrences of a free variable in the head of a clause that does not also occur in the tail. Thus, we may need to generate equations of the form
eq a = y \quad (19)

where \( a \) is a constant, or a term in which \( y \) is not free. In order to solve these over pair-extensions we must include special generic-variable terms \( \text{gen}(c_i) \) \( (i > 0) \) where the \( c_i \) are fresh constants. Then equations like (19) are rewritten

\[ ea = \text{gen}(c_i) \]

where \( c_i \) is the first such constant not already used. When we lift the partial recursive functions computed via theorem (1.6) to nondeterministic functions, we interpret \( \text{gen}(c_i) \) as the set of all terms over the original Herbrand model. These generic variables appear as uppercase logic variables in the lambda calculi introduced in the appendix.

1.3.2 Term-models with choice operators

A useful variant of the term-model construction just given, is the introduction of ternary function symbols

\[ \text{cho}(\sigma, a, b) \]

in lieu of \( \text{pair}(a, b) \), with \( \sigma \) an "unspecified" choice index \( \in \{0, 1\} \). \( \text{cho} \) satisfies the additional equations

\[ \text{cho}(0, a, b) = a \quad \text{cho}(1, a, b) = b \]

This indexing device can also be added to the disjunctive lambda calculus (see appendix) to preserve unrestricted \( \beta \)-reduction and the Church-Rosser property. For convenience, we can extend this in the obvious way to terms

\[ \text{cho}(\sigma, a_1, \ldots, a_n) \]

where \( \sigma \) is in \( \{0, 1, \ldots, n-1\} \) and e.g., \( \text{cho}(1, a, \text{cho}(1, b, c)) = \text{cho}(2, a, b, c) \). We will call an assignment of specific values to all \( \sigma \) in a choice term-model an *instantiation* of the model, and by abuse of language, we will also call them instantiations of the associated pair-terms in the pair-extension. With such instantiations, the multivalued functions computed above now become single valued functions over the Herbrand Universe. These are also called instances of the corresponding pair- or multi-valued functions.

2 SLDR-computation of Equations

We now describe the SLDR-computation algorithm for generating equations over the pair-extension of the Herbrand Universe (or a generic extension).

**Definition 2.1 (\( F \)-unification)** Let \( \Sigma \) be a signature and \( F \) a set of function symbols disjoint from those in \( \Sigma \). Let \( s \) and \( t \) be terms over the signature \( \Sigma \cup F \). A term \( s \) over this signature is called a \( \Sigma \)-term (or just a pure term) if no symbol in \( F \) occurs in \( s \), otherwise it is called an \( F \)-term. An \( F \)-term \( s \) is said to be rigid, or in head form if \( s \) is \( e(t_1, \ldots, t_n) \) for some \( e \in F \).

The set \( C(s, t) \) of \( F \)-unification constraints for \( \langle s, t \rangle \) is a finite set of equations defined according to the structure of \( s \) and \( t \) as follows:

**Case 1.** \( s \) and \( t \) pure: then \( C(s, t) \) is the equational representation of the most general unifier of \( s \) and \( t \): namely a finite set of *basic* equations, i.e., a set of equations of the form \( \{ \cdots \colon x_i = \tau_i \cdots \} \) where the \( x_i \) are variables not occurring in any of the terms \( \tau_j \), representing the substitution \( x_i \mapsto \tau_i \).
Case 2. \( s \) or \( t \) \( F \)-terms: If either \( s \) or \( t \) is rigid, \( C(s, t) \equiv \{ s = t \} \). Otherwise we consider the usual cases found in first-order unification:

- \( s = f(s_1, \ldots, s_n) \) and \( t = a \) or \( t = g(t_1, \ldots, t_m) \) where \( a, f, g \) are in \( \Sigma \) and \( f \) is distinct from \( g \): then \( C(s, t) = \{ s = a, t = a \} \).
- \( s = f(s_1, \ldots, s_n) \) and \( t = f(t_1, \ldots, t_n) \) where \( f \in \Sigma \) : then \( C(s, t) = \bigcup C(s_i, t_i) \) (with the proviso that if \( f \) fails occurs in \( C(s, t) \) then this is equivalent to \( C(s, t) = \{ \text{fail} \} \)).

any basic equations in \( C(s, t) \) contributed as a result of case (1) will be called pure unification constraints.

Scheduling: We do not go into the details of scheduling here. Since the compilation procedure below always terminates, bad scheduling cannot cause divergence. Let us remark, however, that the development is in the style of constraint logic programming, so that failure of unification may not be noticed unless the generated equations are periodically analyzed for consistency with respect to the (decidable) first order theory of equality over the Herbrand Universe. (Identities involving introduced function symbols never introduce inconsistencies). For example, an attempt to unify \( Q(f(x)) \) and \( Q(y) \) in the presence of the constraint \( \{ y = g(a) \} \) may result in success followed by the expansion of the constraint set to \( \{ y = g(a), y = f(x) \} \) which must eventually generate failure. In prolog, this happens at unification time, and there is no reason why this could not be done with SLDR-resolution. Then we must modify the definitions above appropriately to include \( F-C \)-unification of \( s \) and \( t \), i.e. unification in the presence of constraints \( C \). We just apply \( C \) to \( s \) and \( t \) prior to carrying out \( F \)-unification.

We now describe the SLDR-computation process. First a few conventions regarding introduction of function symbols and scheduling.

Introduction of function symbols: We make the following assumptions on the predicates, to facilitate their functional interpretation: all unary predicates \( P(a) \) are treated as predicates in two variables \( P(a, \text{true}) \) so that the associated function \( e_P(a) \) satisfies \( e_P(a) = \text{true} \) when \( P(a) \). 0-ary predicates are just treated as Boolean constants. \( n \)-ary predicates \( Q(t_1, \ldots, t_n) \) are assigned an input-output template: a binary string of length \( n \) defining some slots as inputs, and the remaining ones as outputs, i.e., as components of the associated function value. Because of the inherently relational nature of the induced multivalued equations, there is no need to consider more than one template per predicate and (with the exception of the top-level query) no reason to pick one over another. The equations generated by the SLDR-procedure below will work for any choice. For efficiency in solving these equations and ease of representation, it may prove useful to introduce more than one function for a given predicate corresponding to different templates, (e.g. to associate subtraction with \( \text{add}(m_1, \text{out}, m_2) \) and addition with \( \text{add}(m_1, m_2, \text{out}) \)), but this can be done at the time of solving the equations, and it is not required for the theoretical development or the results below.

In what follows, we will describe the local state of a program \( \alpha : \mathcal{P} \) with distinguished clause \( C \), current goals \( \tau_1 : Q_1, \ldots, \tau_n : Q_n \), equations \( C \) as a node on the SLDR tree with the notation

\[
(C, \alpha : \mathcal{P}, C \vdash Q_1, \ldots, Q_n)
\]

and we show how to develop the subsequent nodes of the tree by induction on the structure of the current query \( \tau_1 \).

Definition 2.2 Let \( \mathcal{P} \) be a first-order program with assigned realizer \( \hat{\alpha} \), and suppose one of the clauses of \( \mathcal{P} \) is

\[
\beta : T(t) \rightarrow Q[r,s]
\]

where \([r,s]\) is any breakdown and rearrangement of the predicate \( Q(t_1, \ldots, t_n) \) (e.g. \( Q(a,b,c) \) with \( x = b \) and \( y = (a,c) \)). Further suppose that the current query is \( Q[x,y] \) i.e. the current state is

\[
(C, \alpha : \mathcal{P}, \beta : T(t) \rightarrow Q[v,s] \vdash Q[x,y])
\]
and that the predicate letter $Q$ has not already occurred as a goal. Then, we introduce a function symbol $e$ (or $e_Q$), rewrite the current state of the program as

$$
\langle C, \alpha : P, \beta : T(t) \rightarrow Q(v, s) \downarrow \text{var} : Q[u, \text{var}] \rangle
$$

Then the following is the SLDR-reduction step backchain:

$$
\langle C, \alpha : P, \beta : T(t) \rightarrow Q[v, s] \downarrow e\alpha' : Q[u, e\alpha'] \rangle
$$

where $\mu$ is the set of pure unification constraints (as defined above in 2.1) generated by $\mathcal{F}$-unification of $Q[v, s]$ and $Q[u, e\alpha]$.

For the sake of thoroughness we have shown how to process the evidence (e.g., $e$ above) along with the witnesses. As remarked above, for self-realizing classes of formulas there is no point to carrying along this extra information, so we omit these terms in the remaining proof steps\footnote{Their sole purpose here, other than to give another motivation for considering hereditarily self-realizing classes of formulas in logic programming languages, is to make possible the treatment of more general classes via the generation of realizability constraints. If the program is not self-realizing, it is only true in the induced realizability model. But this will be taken up elsewhere!}

We assume that the program is written in such a way that there is never more than one clause with a given predicate occurring in the head. Any logic program can be so rewritten by the use of disjunction in the tail of a clause, and the possible addition of equational goals. For example

$$(Q(s) \leftarrow S) \land (Q(t) \leftarrow T)$$

can be rewritten to

$$Q(x) \leftarrow ([x = s, S] \lor [x = t, T]).$$

for a fresh (vector) variable $x$.

**Definition 2.3** Let $P$ be as above and suppose a goal is of the form $S(s) \lor T(t)$. The following is the SLDR-reduction step co-satisfy

$$\langle C, P \vdash S \lor T \rangle$$

$\checkmark$ $\checkmark$

$$\langle C, P \vdash S \rangle \quad \langle C, P \vdash T \rangle$$

The steps corresponding to all the remaining ones in e.g. FOHH logic programming, e.g. **augment** and **generic** in [M2] remain unchanged. **augment** is

$$\langle C, P \vdash A \rightarrow B \rangle$$

$$\downarrow$$

$$\langle C, P, A \vdash B \rangle$$

and **generic**

$$\langle C, P \vdash \forall x A \rangle$$

$$\downarrow$$

$$\langle C, P \vdash A(c) \rangle$$

where $c$ is a fresh constant. The most important step in the SLDR derivation of equations is the recursion step, which occurs every time a goal, e.g. $Q(t_1, \ldots, t_{n+m})$ occurs for a second time on the same path of an SLDR-tree. Then a witnessing function $e_Q$ has already been introduced for it, with the condition

$$\forall Q(x_1, \ldots, x_n, (e_Q(x_1, \ldots, x_n))_1, \ldots, (e_Q(x_1, \ldots, x_n))_m)$$

so the goal is immediately satisfied, with the corresponding $\mathcal{F}$-unification equations added to $C$:...
The search terminates when every goal on the right of the turnstile has a set of recursion equations specifying the corresponding realizer in \( C \). Termination depends only on the predicate letters occurring in a (first-order) program. Call a predicate letter \( A \) active if it has not yet recursed in the sense of the recursion step above. Then, a give predicate \( B \) can only occur on the right of a turnstile if:

- it has never occurred before,
- it recurs on some path for the first time
- it is recurisng again on some path because it is a subgoal associated with an active goal \( A \) (e.g., \( B, C_1, \ldots, C_n \vdash A \)), for if \( A \) were recurrent we would satisfy it using recursion rather than backtrack.

Thus the number of recurrences of an inactive letter is bounded by \( 1 + \) the total number of predicate letters in the program. We omit the straightforward (but tedious) details in this summary.

In particular, termination is guaranteed irrespective of termination of SLD resolution, even if the original program never halts for any input, in which case a solution will be a code or term denoting the totally divergent function, e.g. a least-fix-point solution to the equation \( e = e \).

We also leave for a fuller development of this paper a description of the compilation of the multivalued equations for all introduced function variables. Essentially, the equations on different paths through a node of an SLDRT-tree are processed separately and then merged via disjunctive expressions

\[ \hat{e} x = [t_1 \mid \cdots \mid t_n] \]

or using the formal terms in a pair-extension of the Herbrand Universe. The particular pair-terms that arise are clearly determined by the order in which the equations on different branches are processed.

2.1 Semantics and correctness

Formalizing Realizability and computation over a Herbrand Universe

We will let \( \mathcal{L} = \mathcal{L}(\mathcal{P}) \) denote the language of the logic program. Then define \( \mathcal{H}_P \) to be the Herbrand Universe of the program (the set of ground terms over \( \mathcal{L} \)). We now build a partial applicative theory around \( \mathcal{H}_P \) or its associated pair-extension along lines similar to e.g. (\([1],[36]\)). We will take Beeson’s formulation of Feferman’s theory APP (which Beeson calls PCA+) with a primitive unary postfix predicate \( \downarrow \) for “terms that denote”, together with the standard partial combinators \( s \) and \( k \), pairing and unpairing, a unary integer sort \( N \) and a 4-ary definition by integer cases \( \text{id} \) operator \( d \). The key results we will need here about APP are that it satisfies combinatorial completeness (admits lambda abstraction), and Kleene’s (single and double) recursion theorems, admits “strong” definition by cases, and satisfies the numerical and term existence property. See (\([1],[36]\)) for details.

Definition 2.4 Let \( \mathcal{P} \) be a (Horn or FOHII) logic program decorated with abstract realizers \( \alpha = \langle \alpha_1, \ldots, \alpha_\ell \rangle \). Then we define the associated applicative theory \( \mathcal{E}(\mathcal{H}_P) \) extending APP to include the following:

- constants \( c, f \) for every constant \( c \) and \( n \)-ary function symbol \( f \) in \( \mathcal{L} \). We call these “symbols imported from \( \mathcal{L} \)”, and also denote them \( (c)^* \) and \( (f)^* \). We also include the abstract realizers \( \alpha_i \) directly as constants.

\[ \text{def} \quad x = y \implies \text{true} \quad \text{def} \quad \text{false} \]
• constants $t$ for every ground term $t$ from $\mathcal{L}$. This means that e.g. for constants $a, b, c$ and function symbols $f, g$ imported from $\mathcal{L}$ the terms $f(a, g(b, c))$

\[
(((fa)g)b)c \overset{\text{def}}{=} \text{App(App(App(f, a), g), b), c)}
\]

are syntactically distinct objects in $\mathcal{E}(\mathcal{H}_P)$, (although they are identified via new axioms below).

• A unary predicate $H$ denoting the universe of the interpretation.

• for every term $t$ and function symbol $f$ imported into APP from $\mathcal{H}_P$, and for each abstract realizer $\alpha_i$, the axioms

\[
t \downarrow, \quad H(t), \quad f \downarrow \quad \alpha_i \downarrow \quad (1 \leq i \leq n)
\]

and for each $n$-ary function symbol $f$ and terms $t_1, \ldots, t_n$ imported from $\mathcal{L}$,

\[
(ft_1) \downarrow, \quad (ft_1)t_2 \downarrow \quad \ldots \quad ft_1 \cdots t_n = (f(t_1, \ldots, t_n))^*
\]

• The constant $\text{fail}$ together with the axiom $\text{fail} \downarrow$.

• All schemas in APP are to be extended to the new terms (e.g. $t \downarrow \land \forall x A(x) \rightarrow A(t)$ for each term $t$).

We extend the $*$-embedding of terms to formulas in the obvious way:

$p(t_1, \ldots, t_n)^* = p((t_1)^*, \ldots, (t_n)^*)$ for $* \in \{\to, \land, \lor\}$, etc. We define $\varepsilon$ to be $\mathcal{E}(\mathcal{H}_P)$ together with the embedded program itself (i.e. in the form $((\varepsilon)^*: \varepsilon \in \mathcal{P})$) as a set of nonlogical axioms.

We now develop realizability style semantics over the applicative theories just defined in a manner analogous to e.g. [9, 1, 36], but restricting ourselves to interpreted formulas from the language of the original program. Although we have carried out this interpretation in order to have one theory to discuss both program and evidence, one should perhaps think of this also as a realizability in a distinct object language, namely that of the program, and a realizing metalanguage, namely $\mathcal{E}(\mathcal{H}_P)$ or $\mathcal{E}(P)$.

**Definition 2.5 (Syntactic Realizability over $\mathcal{E}(\mathcal{H}_P)$)** For a term $t$ of $\mathcal{E}(\mathcal{H}_P)$ and a formula $\theta$ over the language of the original program $\mathcal{L}$, we define the binary relation $t : \theta$ (is realized by $\theta$) by cases and by induction on the structure of $\theta$ as follows:

\[
\begin{align*}
\alpha_i : \theta & \quad \text{if } \theta \text{ is the } i\text{-th clause of the program} \\
t : \theta \land \varphi & \overset{\text{def}}{=} p_0 t : \theta \land p_1 : \varphi \\
t : \theta \lor \varphi & \overset{\text{def}}{=} N(p_0 t) \land (p_0 t \iff p_1 t : \varphi) \\
t : \theta \rightarrow \varphi & \overset{\text{def}}{=} \forall \xi (\xi : \theta \rightarrow \xi : \varphi) \\
t : (\exists x \in D) \theta(x) & \overset{\text{def}}{=} D(p_0 t) \land p_1 t : \theta(x/p_0 t) \\
t : (\forall x \in D) \theta(x) & \overset{\text{def}}{=} \forall u D(u) \rightarrow tu \downarrow \land tu : \theta(x/u)
\end{align*}
\]

We now can state our main representation theorem for first-order logic programs. We use the following notational convention. If $t$ is a term in $\mathcal{E}(\mathcal{H}_P)$, we write $t = \langle t_1, \ldots, t_n \rangle$ to denote iterated pairing with association to the right. Thus $p_0 t = t_1$, $p_0 (p_1 t) = t_2$, etc.

**Theorem 2.6** Let $\mathcal{P}$ be a first-order program, with generic query $\theta[u, v]$. Then

1. Any instance $e$ of a SLDR-computed term $e$ in $\mathcal{E}(\mathcal{H}_P)$ provably realizes the program in the sense that for some $D$ definable in $\mathcal{E}(\mathcal{H}_P)$:

\[
\mathcal{E}(\mathcal{H}_P) \vdash e : (\forall l \in D)(\exists r \in D)(\mathcal{P} \rightarrow \theta \langle \theta[l], r \rangle)
\]
2. In particular, there is a term \( t = \langle t_1, \ldots, t_n \rangle \) in \( \mathcal{E}(\mathcal{H}_P) \) which is (the \( * \)-image of) an instance of an SLDR computed term and which maximally satisfies the program as a specification in the sense that for any input \( u \in \mathcal{H} \) for which
\[ \mathcal{P} \vdash \exists \theta \widehat{u} \theta \left[ \widehat{u}; \widehat{v} \right] \]
we have
\[ \mathcal{E}(\mathcal{H}_P) \vdash (\mathcal{P} \rightarrow \theta \left[ \widehat{u}; \widehat{v} \right]) \]
Such a witnessing term \( t \) is faithful to the program in that
- whenever \( (t_1 u_1 \cdots u_m) \downarrow \) the values \( (t_1 u_1 \cdots u_m) \) are (modulo the \( * \)-translation between language and metalanguage) correct answers for the original program \( \mathcal{H} \).
- whenever values \( u_1, \ldots, u_3 \) exist for a set of inputs \( u_1, \ldots, u_m \), we have \( (t_1 u_1 \cdots u_m) \downarrow \).

The theorem follows from the fact that all logic programming languages considered here are self-realizing classes of formulas (see e.g. [1, 36]). Thus whenever
\[ (\mathcal{P} \rightarrow \theta \left[ u_1, \ldots, u_m; (t_1 u_1 \cdots t_1 u_m), \ldots, (t_n u_1 \cdots t_n u_m) \right]) \]
is realized provably in \( \mathcal{E}(\mathcal{H}_P) \), it is provable, and conversely. It is shown in e.g. [1] that all the key properties of \( \text{APP} \) are preserved in extensions by self-realizing theories, including the soundness of most realizability notions (in particular, ours), the existence property, and the recursion theorem.

It is clear that the structure of SLDR-computation is geared towards preserving the computational content of the logic programs without selecting an arbitrary evaluation strategy. To make this precise we need to compare proof search strategies (selection and branching rules) with evaluation strategies of the resulting terms. The notational machinery for this is cumbersome and is omitted here, although the results and ideas are straightforward.

**Corollary 2.7** Suppose terms \( e \) and \( t \) from the theorem above are instances of Nerode-Kleene solutions to a finite set of equational constraints \( \mathcal{C} \) over the disjunctive term calculus obtained by SLDR-computation. Then these terms preserve the computational content of the logic program in the sense that every fair evaluation instantiation strategy for the resulting terms in a choice-extension of the Herbrand model corresponds to a complete proof search strategy for the logic program.

3 Other realizability interpretations

3.1 Adding a Domain or Type variable: Constraint Logic programming

We provide a brief sketch of the way a modified realizability enables us to capture constraint logic programming and at the same time ensure a total realizability witness, by constraining the domain of the computed realizer. The idea is to formally add an existentially quantified domain variable \( D \) when formulating the realizability goal for \( \mathcal{P} \):
\[ e : \exists D(\forall u \in D)(\exists v \in D)(\mathcal{P} \rightarrow Q(u, v)). \]
This formulation does not require a partial applicative structure, and can be developed in total type theoretic frameworks such as, e.g., Martin-Löf type theory [5, 36, 1, 24]. It can also be developed using Kreisel-Troelstra realizability over HAS or IZF (see e.g. [21]). In this case constraints are added to the list of goals to be solved, and determine the instantiation of the variable \( D \).

3.2 A Kripke model for first-order logic programs

We can capture the notion of partial realizability of logic programs formalized directly over the Herbrand model using Nerode-Kleene computability by considering pairs \((e, D)\) where \( D \) is a subset of the Herbrand
Universe on which e converges as follows.
We define a Kripke model $K$ with the following carrier set:

$$K = \bigcup K_{x,y} \text{ where } K_{x,y} = \{(e, D, \bar{x}, \bar{y}) : \forall \bar{u} \in D \text{ eif } \}$$

where $\bar{x}, \bar{y}$ range over all tuples of variables $x_1, \ldots, x_k$ and $x_{k+1}, \ldots, x_{k+m}$ ($n, m > 0$) and with order given by $(e, D, \bar{x}, \bar{y}) \leq (e', D', \bar{x}', \bar{y})$ if $e$ extends $e'$ (i.e., $\forall x \in D \downarrow \exists x \in D' \downarrow x = x'$) and $D$ is a subset of $D'$. Nodes $(e, D, \bar{x}, \bar{y})$ may only force atomic formulas with free variables matching the tuples $\bar{x}, \bar{y}$. Atomic forcing is given by

$$(e, D, \bar{x}, \bar{y}) \models \phi$ 

$$\iff \forall \bar{u} \in D \models \phi[ar{u}, eif]$$

where eif is a tuple of length equal to $\bar{y}$. We introduce the following notion of cover: The set $S \subseteq K$ is a cover of $(e, D, \bar{x}, \bar{y})$ if for every $(e', D', \bar{x}', \bar{y})$ in $S$, $(e', D')$ is above $S$ and $D \subseteq \bigcup_{D' \in S}$. Then we define forcing of disjunctions via covers (see e.g. [36]) and obtain a model in which for all first-order goals $A$

$$(e, D, \bar{x}, \bar{y}) \models \phi$$

$$\iff \forall \bar{u} \in D \models \phi[ar{u}, eif]$$

Using this semantics we obtain another completeness theorem for the SLDRT translation (provide(1.27 1.16) 1.1 d P is assumed to be Horn or Hereditarily Horn); every node $(e, D, \bar{x}, \bar{y})$ in $K$ forcing a query $Q[\bar{y}]$ is extended by some instance of an SLDRT-computed term $e'$, and every SLDRT-computed term occurs in a node of $K$.

### 3.3 Realizability by multivalued functions

As we illustrated by example above, we can also choose constructive set theory (IZF) as our meta-theory, with an applicative structure embedded in it, as in e.g. McCarty’s ([21]). We are then able to construct the realizability interpretation of IZF as a basic model, (also known as the realizability universe, or $\Rightarrow$ with some variation—the effective topos, see e.g. [15]). Our realizers live in this model, but will not be the terms of the formalized applicative structure. They are sets and multivalued functions definable in IZF, with the following possible realizability definitions induced by term realizability “$\Rightarrow$” over the logic program, which we call strong and weak realizability ($\Rightarrow$ and $\Rightarrow$). Strong realizability by sets of realizers is just

$$e \Rightarrow \theta \overset{\text{def}}{=} (\forall x \in e)(x : \theta)$$

Weak realizability is as follows. Conjunction and implication and existence are as before. The interesting clause is disjunction.

$$e \Rightarrow \theta \vee \varphi \overset{\text{def}}{=} (\forall x \in e)((x : \theta) \vee (x : \varphi))$$

Towards a non-deterministic realizability of logic programs

If we use the more syntactic formulations of multivalued functions described above and in the appendix, then order of the terms, and the possible presence of a fail token suggest more complex definitions, to capture negation as failure

$$[t_1 \mid t_2] : \theta \vee \varphi \overset{\text{def}}{=} t_1 : \theta \perp t_2 : \varphi \perp (t_1 = \text{fail} \rightarrow t_2 : \varphi) \wedge (t_2 = \text{fail} \rightarrow t_1 : \varphi$$

where

$$t : (\theta) \perp t : \theta \vee t = \text{fail}$$

and where for any predicate $Q(x_1, \cdots, x_n)$, $Q[x_i/fail]$ is true. We are able to obtain an analogue of theorem 2.6 for multivalued realizability, discussed in the [19].
4 Conclusion

Using realizability interpretations of logic programs we can translate them into disjunctive terms that (weakly) inhabit these programs as types, or specifications in the Curry-Howard sense. These translations maintain the declarative meaning of the program while leaving control features untouched. These are transferred to the evaluation-control problems: the way in which the recursion theorem and disjunctive branching conditions are evaluated in the target model. This points to the possibility of studying control features of logic programming in the context of functional programming with nonlocal control operators. It also gives a natural way of associating domain-theoretic interpretations to logic programs. Our translations also transform logic programs into first and higher order constraints over various kinds of applicative structures. This also permits the formalization within logic programming itself of constraint solving in a new way. Many new questions need to be addressed here: which constraints generated by logic programs in the style discussed here have solutions over different typed calculi. Can we identify a (typed) logic programming language corresponding to various subrecursive classes? How can we modify our realizability calculus to capture non-local control more efficiently and clearly? Perhaps most importantly of all, our work suggests that a major task ahead of us is to understand equation and constraint-solving (together with feasibility questions) over applicative structures along the lines initiated by Statman and Tronci ([34, 35]). This work must combine type inference and simultaneous constraint solving. This issue is discussed in more depth in [19].

In particular, higher order programming is likely to require the simultaneous solution of domain equations and constraint sets over continuous or computable functions on the indicated domain.
Appendix

The disjunctive calculus

Following the example computed earlier, we can extend the term structure of the lambda calculus or of a partial combinatory calculus such as APP to include disjunctive terms $[t_1 \mid t_2]$. We also consider the addition of generic or logic variables $X$ (which can also be thought of as labelled contexts) standing for all possible (non-disjunctive) instances. We consider two developments here: pure disjunctive calculus

$$t ::= x \mid X \mid (t_1 t_2) \mid \lambda x.t \mid [t_1 \mid t_2]$$

indexed disjunctive calculus

$$t ::= x \mid (X)^\sigma \mid (t_1 t_2) \mid \lambda x.t \mid [t_1 \mid t_2]$$

The second calculus allows us to reason about disjunctive lambda terms with a “generic” evaluation index $\sigma$ which is assumed to select one of the disjuncts, which admits full-blown $\beta$ reduction. In the pure calculus we must lift disjunctions to the top (using rule 22 and rule 30 prior to $\beta$ reducing with rule 23). We now give reduction rules for both calculi. The optional rules are included to better handle the simplification of functions computed from logic programs, but do not seem essential for the lambda calculi themselves.

<table>
<thead>
<tr>
<th>pure disjunctive calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[s \parallel t] u \leadsto [su \parallel tu]$</td>
</tr>
<tr>
<td>$(\lambda x.s)u \leadsto s[x/u]$ non-disjunctive</td>
</tr>
<tr>
<td>$\lambda x.[s \parallel t] \leadsto [\lambda x.s \parallel \lambda x.t]$</td>
</tr>
<tr>
<td>$(\lambda x.s)[u \parallel v] \leadsto [(\lambda x.s)u \parallel (\lambda x.s)v]$</td>
</tr>
<tr>
<td>$\langle u \parallel v, z \rangle \leadsto [(u, z) \parallel (v, z)]$</td>
</tr>
<tr>
<td>$\langle z, [u \parallel v] \rangle \leadsto [(z, u) \parallel (z, v)]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>indexed disjunctive calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[s \parallel t] u \leadsto [su \parallel tu]$</td>
</tr>
<tr>
<td>$(\lambda x.s)u \leadsto s[x/u]$</td>
</tr>
<tr>
<td>$\langle u \parallel v, z \rangle \leadsto [(u, z) \parallel (v, z)]$</td>
</tr>
<tr>
<td>$\langle z, [u \parallel v] \rangle \leadsto [(z, u) \parallel (z, v)]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>optional rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f[u \parallel v] \leadsto [fu \parallel fv]$</td>
</tr>
<tr>
<td>$[u \parallel \text{fail}] \leadsto u$</td>
</tr>
<tr>
<td>$[\text{fail} \parallel u] \leadsto u$</td>
</tr>
<tr>
<td>$[u \parallel u] \leadsto u$</td>
</tr>
</tbody>
</table>

---

3The author has recently learned of substantial work in disjunctive calculi by Piperno, Liguori, and Dezani, among others, which may provide a better framework for nondeterministic term extraction from logic programs.
References


[34] Statman, R., [1982], “Completeness, Invariance and lambda-definability”, JSL.

