Extensionality of simply typed logic programs

M.A. Bezem

Probability, Networks and Algorithms (PNA)

PNA-R9907 September 30, 1999

Report PNA-R9907
ISSN 1386-3711

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Extensionality of Simply Typed Logic Programs

Marc Bezem
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
and
Dept. of Philosophy, Utrecht University
P.O. Box 80126, 3508 TC Utrecht, The Netherlands
bezem@phil.uu.nl

ABSTRACT

We set up a framework for the study of extensionality in the context of higher-order logic programming. For simply typed logic programs we propose a novel declarative semantics, consisting of a model class with a semi-computable initial model, and a notion of extensionality. We show that the initial model of a simply typed logic program, in case the program is extensional, collapses into a simple, set-theoretic representation. Given the undecidability of extensionality in general, we develop a decidable, syntactic criterion which is sufficient for extensionality. Some typical examples of higher-order logic programs are shown to be extensional.

1991 Mathematics Subject Classification: 68N05, 68N17, 68Q60
1991 ACM Computing Classification System: F.3.1, F.4.1
Keywords and Phrases: Logic programming, higher-order logic, simple types.
Note: Work carried out under project PNA1.2, Constraint and Integer Programming. To appear in Proceedings ICLP99.

1. Introduction

Higher-order logic programming is an extension of logic programming where variables for predicates (and predicates of predicates, and so on) are allowed and predicates can be used in terms.

Some restricted forms of higher-order logic programming are already supported by Prolog, but in the more recent logic languages, following functional languages, the higher order has been incorporated in the design from the very beginning: Lambda Prolog [8], Hilog [2], Gödel [5], and Mercury [11]. Good references are [10] and [9].

In this paper we address the problem of assigning meaning to higher-order logic programs, and study in particular the phenomenon of extensionality. The basis of the proposed semantics is the theory of simple types, founded by Church [3]. In order to get a clear picture of the notion of extensionality itself, we keep the terms and the types as simple as possible. For this reason we do not yet consider lambda abstraction, higher-order unification, polymorphism, subtyping and so on, but we are well aware of the importance of these features for the languages cited above.

Let us start with an instructive example. Consider a higher-order logic program P, which includes the following clauses, besides the definition of a number of first-order predicates.

```
stepwise(R,[]).
stepwise(R,[_]).
stepwise(R,[X,Y|Z]):- R(X,Y),stepwise(R,[Y|Z]).
```

These three clauses show one of the key features of higher-order logic programming: the variable `R` stands for an arbitrary binary predicate and is used in a term position of `stepwise`, which is a

predicate of (binary) predicates and lists of individuals. The meaning of `stepwise` is clear: if `r` is a binary predicate and `l` is a list, then `stepwise(r,l)` holds iff `r` holds between every two successive elements of the list `l`. Note that `stepwise(r,l)` only depends on the *extension* of `r`, that is, on its set of pairs, and not on the *intension*, its name `r`. We call the program P *extensional*, as its meaning can be explained in terms of extensions. See also Example 4.3.

Next, assume we want to use the predicate stepwise to define when a list is ordered with respect to an ordering relation. It is easy to add the following clause to the program.

```
ordered(R,L):- ordering(R),stepwise(R,L).
```

But then we have to face the problem of defining the predicate (of binary predicates) `ordering`. As the extensions of binary predicates can be recursively enumerable (Σ_1), cf. [1], deciding whether such an extension is irreflexive is co-enumerable (Π_1) and transitivity has already the complexity of the universal halting problem (Π_2). In fact we can do no better than, for a number of known orderings involved, state explicitly so:

```
ordering(r1).
ordering(r2).
...
```

Now the meaning of the predicate `ordered` has become: if `r` is a binary predicate and `l` is a list, then `ordered(r,l)` holds iff `ordering(r)` and the list `l` is ordered with respect to the extension of `r`. The subtle point here is that `ordering(r)` does not depend on the extension of `r`, but on its intension, the name `r` itself. If some other binary predicate `s` has the same extension as `r` (verifying this has the complexity of the universal halting problem), then it is not warranted that `ordering(s)` iff `ordering(r)` holds, and hence we can have mismatches between `ordered(r,l)` and `ordered(s,l)`, although `r` and `s` have the same extension. As a consequence, the meaning of the extended program cannot be explained in terms of extensions only, whence we call such programs *intensional*, as opposed to the *extensional* program above.

Many properties of extensions are highly undecidable. Due to the unification process, logic programming seems to be inherently intensional. Completeness of the operational behaviour seems incompatible with an extensional semantics. Is this hopeless?

On the positive side, extensions of predicates are important to describe their semantics, and extensionality, if it holds, is an important structural property of a predicate, which simplifies its semantics and thus contributes to a better understanding of the program. Let us see how far we can come.

2. Terms and types

Under the restrictions we adopted, the terms of higher-order logic programming are in fact extremely simple: they are built up from constants and variables by application. To arrive at this simplicity, however, we must adopt a fully curried version of the syntax (named after Curry). This means that a function of more than one argument is viewed as a function of one argument to functions of one argument, and so on. In other words, not the tuple is the argument, but its components are successive arguments of the function (values). Then we do not need cartesian products like in $(D\times D)\to D$, but we can do with the type constructor $\to$ only, such as in $D\to(D\to D)$.

For compliance with the usual syntax we practise a liberal way of currying. For example, the following denotations are all identified:

$$pxa \equiv (px)a \equiv p(x,a) \equiv p(x)a \equiv p(x)(a)$$

This means that brackets and argument lists can be used freely, but serve only as alternative denotations of curried terms.

The (untyped) terms of logic programming are given by the following abstract syntax:

$$\mathcal{T} ::= \mathcal{C} \mid \mathcal{V} \mid (\mathcal{T}\mathcal{T})$$

Here $\mathcal{C}$ and $\mathcal{V}$ are sets of constants and variables, respectively, and $\mathcal{TT}$ are terms obtained by application. In principle $\mathcal{C}$ and $\mathcal{V}$ contain all correct Prolog constants and variables, respectively, but in most examples we will tacitly assume that $\mathcal{C}$ contains only those constants that are explicitly mentioned in the example. We use x, y, z, u, v, w to denote variables, a, b, c, d, p, q, r as constants, and s, t to denote arbitrary terms. Application is taken to be left associative, so $p(xa)$ is notably different from the denotations above. The *head symbol* of a term is its leftmost symbol, either a variable or a constant.

Example 2.1 Examples of (untyped) terms are $c(x, a), c(c), xxy, x(xy)$. The first and the last will turn out to be typable, the others are not. □

A *clause* is an expression of the form $t_0 \leftarrow t_1, \ldots, t_n$, where all t_i are terms $(0 \leq i \leq n)$. A program is a finite set of clauses. We use C for clauses and P for programs. In the examples we use Prolog notation for clauses and programs.

Example 2.2 Consider the following clauses.

```
call/2(X,Y1,Y2):- X(Y1,Y2).
foo(Z):- Z(Z).
X(Y1,Y1).      % every binary relation is reflexive
```

The first and the last clause will turn out to be typable, the second is not. Note that we allow a variable as head symbol of the head of a clause. □

In order to single out the well-typed higher-order logic programs we use types. The fragment consisting of first-order terms can be typed with types given by the following abstract syntax:

$$\mathbb{D} ::= D \mid (D \rightarrow \mathbb{D})$$

Here D is the base type of a (fixed) domain of individuals and $(D \rightarrow D)$, $(D \rightarrow (D \rightarrow D))$, ... are the types of unary functions, binary functions, and so on. Here and below we let $\rightarrow$ associate to the right and we drop outermost brackets.

The types for the higher-order objects are given by the following abstract syntax:

$$\mathbb{T} ::= A \mid (\mathbb{T} \rightarrow \mathbb{T}) \mid (D \rightarrow \mathbb{T})$$

Here A is the base type of atoms. The types $D \rightarrow A$, $D \rightarrow D \rightarrow A$, ... are the types of unary predicates, of binary predicates, and so on. The types $A \rightarrow A$, $(D \rightarrow A) \rightarrow A$, ... are the types of predicates on atoms, on unary predicates, and so on. More complicated types can easily be constructed, for example, $((A \rightarrow A) \rightarrow A) \rightarrow (D \rightarrow D \rightarrow A) \rightarrow A$.

The set of types for higher-order logic programming is the union of $\mathbb{D}$ and $\mathbb{T}$. The former will only play a minor role in this paper, and we will focus attention on the latter. We use σ, τ to denote arbitrary types.

Lemma 2.3 *Every type σ in $\mathbb{T}$ is of the form*

$$\sigma \equiv \sigma_1 \rightarrow \cdots \rightarrow \sigma_k \rightarrow A$$

for some $k \geq 0$, with σ_i either D or in $\mathbb{T}$, for every $1 \leq i \leq k$.

Proof: By induction on σ. In the base case we have $\sigma \equiv A$ and we take $k = 0$. There are two induction steps, namely $D \rightarrow \tau$ with $\tau \in \mathbb{T}$ and $\tau' \rightarrow \tau$ with $\tau, \tau' \in \mathbb{T}$. Both are settled by applying the induction hypothesis to τ. Assume $\tau \equiv \tau_1 \rightarrow \cdots \rightarrow \tau_k \rightarrow A$ as in the lemma. Now the lemma is proved for σ by incrementing k, putting $\sigma_{i+1} \equiv \tau_i$ and $\sigma_1 \equiv D$ or $\sigma_1 \equiv \tau'$. □

Due to the absence of abstraction, the typing system for terms is very simple and consists of one single typing rule:

$$\frac{t : \sigma{\to}\tau \quad s : \sigma}{ts : \tau}\ (\to)$$

We will now formally define when a higher-order logic program is typable.

Definition 2.4 A *declaration* is an expression either of the form $x : \sigma$ with x a variable, or of the form $c : \sigma$ with c a constant, stating that x (respectively c) has type σ. The variable (constant) on the left hand side is called the *declarandum* of the declaration.

A *context* is a finite list of declarations with different declaranda. Contexts are denoted by Γ.

The *typing relation* $\Gamma \vdash t : \sigma$ is defined inductively as the smallest relation which holds whenever $t : \sigma$ is a declaration in Γ and which is closed under the typing rule $(\to)$ above.

A term t is *typable* by Γ if there exists σ such that $\Gamma \vdash t : \sigma$.

A clause $t_0 \leftarrow t_1, \ldots, t_n$ is *typable* by Γ if $\Gamma \vdash t_i : A$ for all $0 \leq i \leq n$. In that case we write $\Gamma \vdash t_0 \leftarrow t_1, \ldots, t_n : A$.

A program P consisting of clauses $C_1, \ldots, C_n$ is *typable* by Γ if $\Gamma \vdash C_i : A$ for all $1 \leq i \leq n$. In that case we write $\Gamma \vdash P : A$.

We call a term (clause, program) *typable* if it is typable by Γ for suitable Γ. If we speak of a clause (term) in relation to a typable program, then we implicitly assume the clause (term) to be typable by the same context. □

Intuitively, $\Gamma \vdash P : A$ means that the declarations in Γ ensure that each atom in P is of the base type A. Note that one and the same variable may occur in different clauses of P, but always with the same type as declared in Γ. In cases in which different types are required, the program clauses should be standardized apart.

An alternative characterization of the typing relation is the following: $\Gamma \vdash t : \sigma$ holds if and only if there exists a derivation tree according to $(\to)$ with root $t : \sigma$ and leaves in Γ.

Due to the absence of abstraction, the typing system is in fact a subsystem of that for simply typed combinatory logic. We rely on the well-established techniques on principal type schemes for type checking and type synthesis. Space limitations prevent us from entering this important subject, instead we refer to the original source [6], or [7] for a modern exposition, and to the ML literature.

To give the reader at least some idea, the principal type scheme of the term xy is $x : \alpha{\to}\beta,\ y : \alpha$, with α, β arbitrary types. The principal type scheme of the atom xy is $x : \alpha{\to}A,\ y : \alpha$, and of the clause $xy \leftarrow y$ it is the context $x : A{\to}A,\ y : A$.

Example 2.5 Recall the program from Example 2.2. The first and the last clause are typable by the context:

```
call/2: (D->D->A)->D->D->A
X: D->D->A
Y1,Y2: D
```

This means that `call/2` takes three arguments of types `D->D->A`, `D` and `D`, respectively. The second clause of Example 2.2 is not typable, since `Z` cannot have both type σ and type $\sigma{\to}A$. □

3. Operational semantics

The operational semantics is in fact an extension of the usual one for first-order logic programming. Space limitations prevent us from entering soundness and completeness issues. We treat only some key points needed for a proper understanding of the sequel, namely unification, well-typed substitution and the immediate consequence operator. The latter will play a role in inductive proofs.

We extend the Martelli-Montanari algorithm such as described in [1]. Unification of two terms can only succeed if they have the same type, say τ, in a context Γ. In unifying them, we first write the terms in the form $t_0 t_1 \dots t_k$ and $s_0 s_1 \dots s_l$, where t_0 and s_0 are the respective head symbols. If the head symbols have the same type, then $k = l$ and the algorithm proceeds in the usual way. If $t_0 : \tau_1 \to \dots \to \tau_k \to \tau$ and $s_0 : \sigma_1 \to \dots \to \sigma_l \to \tau$ have different types, then the unification fails in all but the following two symmetric cases.

- $l < k$, $\sigma_i \equiv \tau_{i+k-l}$ for $1 \leq i \leq l$ and s_0 is a variable. Now the algorithm proceeds by binding s_0 to $t_0 t_1 \dots t_{k-l}$ and by unifying s_i with t_{i+k-l} for all $1 \leq i \leq l$.
- $k < l$, etcetera, symmetric to the previous case.

Type persistence under well-typed substitution (and hence under resolution) is ensured by the following lemma.

Lemma 3.1 *If* $\Gamma, x : \sigma \vdash t : \tau$ *and* $\Gamma, \Gamma' \vdash s : \sigma$, *then* $\Gamma, \Gamma' \vdash t[x/s] : \tau$.

Proof: By induction on the derivation of $\Gamma, x : \sigma \vdash t : \tau$. □

We recall the familiar notions of Herbrand Base, Herbrand Universe, immediate consequence operator and its least fixed point. These notions are now slightly more general as the terms involved stem from the higher-order syntax. By convention all terms are assumed to be typable in the context of the program.

Definition 3.2 Let P be a typable higher-order logic program. We define the *Herbrand Base* B_P (resp. the *Herbrand Universe* U_P) to be the set of all closed terms of type A (resp. D). For every $S \subseteq B_P$ we define $T_P(S) \subseteq B_P$ by $t \in T_P(S)$ iff there exists a closed instance of a program clause in P with head t and all body atoms in S. The operator T_P is called the *immediate consequence operator* of P. As usual, $T_P{\uparrow}0 = \emptyset$, $T_P{\uparrow}(n{+}1) = T_P(T_P{\uparrow}n)$ and $M_P = T_P{\uparrow}\omega = \bigcup_{n \geq 0} T_P{\uparrow}n$. □

4. Declarative semantics

Some introductory remarks may help to grasp the intuition behind the proposed declarative semantics. Objects of type $\sigma \in \mathbb{D}$ are never defined in a logic program. The reason is that such objects only occur in argument positions in atoms. Hence we simply put the interpretation $[\![\sigma]\!]_P$ of types $\sigma \in \mathbb{D}$ to be the set of closed terms of type σ. This includes $[\![D]\!]_P = U_P$.

For types from $\mathbb{T}$, in particular for the base type A, the situation is entirely different. At first thought one may expect $[\![A]\!]_P$ to be the set of truth values $\{\mathbf{T}, \mathbf{F}\}$, indeed the domain of interpretation of atoms in classical, extensional logic. However, intensionality makes the applicative behaviour of the higher-order objects non-truth-functional. The right interpretation of $[\![A]\!]_P$ turns out to be the graph of the characteristic function of the least fixed point M_P (see also [9]). This graph contains all information about the names as well as about the truth values of the atoms. In other words, the interpretation of atoms consists of pairs of intensions and extension. This will also be the case for objects of the other types in $\mathbb{T}$, i.e. for predicates, predicates of predicates, and so on.

For every typable program P we shall define, successively,

- $|\sigma|_P$ the set of all intensions of type σ,
- $|t|_P$ the intension of closed term t of type σ,
- $[\sigma]_P$ the set of (possible) extensions of type σ,
- $[t]_P$ the extension of closed term t of type σ.
- $[\![\sigma]\!]_P$ the interpretation of type σ,

- $[\![t]\!]_P$ the interpretation of closed term t of type σ.

For every type σ, $[\![\sigma]\!]_P$ will be the set consisting of all $[\![t]\!]_P$ with t a closed term of type σ. For t of type σ in $\mathbb{D}$, $[\![t]\!]_P$ will be the intension $|t|_P$. For t of type σ in $\mathbb{T}$, $[\![t]\!]_P$ will be the pair $(|t|_P, [t]_P)$ consisting of the intension t and the extension of t.

The *intension* of a closed term is simply the closed term itself. The *extension* of a closed term depends on its type and is either

- the term itself (this is the case for terms of type in $\mathbb{D}$), or
- a truth value (extension of type A), or
- a function mapping intensions of type σ to extensions of type τ (extension of type $\sigma{\to}\tau$ in $\mathbb{T}$).

For every type σ, $[\sigma]_P$ will be a set containing at least all $[t]_P$ with t a closed term of type σ.

We are now in a position to give a formal definition of the interpretations of types and terms.

Definition 4.1 Let P be a typable higher-order logic program. We shall define the six items listed above, distinguishing between types $\sigma \in \mathbb{D}$ and $\sigma \in \mathbb{T}$.

$\mathbb{D}$: We put $[\![t]\!]_P = [t]_P = |t|_P = t$ for all closed terms of type $\sigma \in \mathbb{D}$, and $[\![\sigma]\!]_P = [\sigma]_P = |\sigma|_P$ consist of all such terms.

$\mathbb{T}$: We put $|t|_P = t$ for all closed terms of type $\sigma \in \mathbb{T}$, and $|\sigma|_P$ consists of all such terms. For $[\sigma]_P$ we proceed by induction on σ, hence we distinguish the base case A and two induction steps. In all these cases $[\![\cdot]\!]_P$ is defined in terms of $|\cdot|_P$ and $[\cdot]_P$.

For the base type A we put $[A]_P = \{\mathbf{T}, \mathbf{F}\}$ and $[\![A]\!]_P = \{[\![t]\!]_P \mid t : A \text{ closed}\}$ with $[\![t]\!]_P = (t, \mathbf{T})$ if $t \in M_P$ and $[\![t]\!]_P = (t, \mathbf{F})$ otherwise.

For the induction steps, let σ be $\sigma'{\to}\tau$ with either $\sigma' \equiv \tau'$ or $\sigma' \equiv D$ and assume $[\cdot]_P$ has already been defined for τ such that $[t]_P \in [\tau]_P$ for every closed term of type $t : \tau$. Define $[\sigma'{\to}\tau]_P$ to be the set of all functions from $|\sigma'|_P$ to $[\tau]_P$. Note that the domain of these functions consists of intensions, whereas the co-domain contains extensions. Next define

$$[\![\sigma'{\to}\tau]\!]_P = \{[\![t]\!]_P \mid t : \sigma'{\to}\tau \text{ closed}\}$$

where (and this is the crucial pairing of intension and extension)

$$[\![t]\!]_P = (t, [t]_P)$$

with $[t]_P \in [\sigma'{\to}\tau]_P$ defined by

$$[t]_P t' = [tt']_P$$

for every closed term t' of type σ'. Observe that $[t]_P$ is well-defined since tt' is a closed term of type τ, so that indeed $[tt']_P \in [\tau]_P$. □

We shall omit the subscript P in $[\![\cdot]\!]_P$, $[\cdot]_P$, $|\cdot|_P$ when no confusion can arise.

Lemma 4.2 *Let P be a typable higher-order logic program. For all types σ and all $\sigma_i \in \mathbb{T}$ we have:*

1. *$[t] \in [\sigma]$ for all closed $t : \sigma$;*
2. *$[\sigma_1{\to}\ldots{\to}\sigma_n{\to}A]$ is isomorphic to the powerset of $|\sigma_1| \times \ldots \times |\sigma_n|$.*

Proof: By a simple induction on σ and n, respectively. □

It follows from Definition 4.1 that the interpretation of a higher-order logic program is completely determined by the extensions of all closed terms. In our examples we often give only these extensions, leaving it to the reader to pair the extensions with the intensions into interpretations. We shall also refer frequently and implicitly to Lemma 4.2, clause 2, by presenting extensions as subsets rather than as characteristic functions of these subsets.

Example 4.3 Consider the following typable higher-order logic program P.

```
p(a).               % a: D, p: D->A
holds(X,Y):- X(Y). % Y: D, X: D->A, holds: (D->A)->D->A
```

We have the following extensions (and one intension) relative to P.

```
[D]={a}
[D->A]={{},{a}}
|D->A|={p,holds(p),holds(holds(p)),...}
[p]=[holds(p)]=[holds(holds(p))]=...={a}
[holds]={(p,a),(holds(p),a),(holds(holds(p)),a),...}
```

The extension of `holds` is already quite complicated. The notion of model proposed in [9] labours similar complications.

Anticipating the introduction of extensionality, we state that P is extensional and that by virtue of this property it is possible to collapse the extension `[holds]` into `{(a,a)}`, that is, a subset of the product of the *extensions* `[D->A]` and `[D]`, instead of the (generally more complicated) *intensions*. See also Lemma 4.2 and Corollary 6.5. □

Definition 4.4 Let P be a typable higher-order logic program. We define the declarative semantics $\mathcal{M}_P$ of P to be the structure consisting of sets $[\![\sigma]\!]_P$ for all types σ, with application mappings

$$ap_{\sigma,\tau} : [\![\sigma{\to}\tau]\!]_P \times [\![\sigma]\!]_P \to [\![\tau]\!]_P$$

defined by

$$ap_{\sigma,\tau}([\![t]\!]_P, [\![t']\!]_P) = [\![tt']\!]_P.$$

We denote application by juxtaposition and associate to the left. Another way of phrasing $[\![t]\!]_P[\![t']\!]_P = [\![tt']\!]_P$ is that $[\![\cdot]\!]_P$ is a homomorphism with respect to syntactic and semantic application. Note that, for $t : \sigma{\to}\tau \in \mathscr{T}$ and $t' : \sigma$, we have

$$[\![t]\!]_P[\![t']\!]_P = (t, [t]_P)(t', [t']_P) = (tt', [tt']_P) = (tt', [t]_P t').$$

5. Models for higher-order logic programs

The interpretation $\mathcal{M}_P$ from Section 4 raises the more general question about the nature of models for higher-order logic programs. In this section we propose a large model class for higher-order logic programs. The idea is to separate the applicative behaviour of higher-order objects from the logical behaviour. Although this semantical framework may seem overly general at first sight, there are strong reasons in favour of this generality:

- The extensional collapse, see Example 4.3, to be introduced in the next section, can be carried out within the model class. This does not seem to be the case in the set-up of [9, Section 3.4]. [1]

[1] [9, page 521] states that all models are extensional, but extensionality is used there in a weaker sense of non-truth-functional behaviour: $D_o = [\![A]\!]$ and not $[A]$.

- A greater flexibility with respect to future extensions of the typing system is achieved.
- The larger the model class is, the more applications there are.

Definition 5.1 A *type structure* $\mathcal{D}$ consists of sets D_σ for every type σ and application mappings

$$ap_{\sigma,\tau} : D_{\sigma\to\tau} \times D_\sigma \to D_\tau$$

for all types σ, τ. We denote application by juxtaposition and associate to the left.

A type structure is extended to an interpretation for higher-order logic programs in the following way. First we add an interpretation function I which assigns an element of D_σ to every constant of type σ. Next we add a valuation function V assigning a truth value (**T** or **F**) to every element of D_A. What follows now is a standard development of the interpretation of terms, but with application according to the given type structure. An *assignment* is a function mapping variables to domain elements of the corresponding types. Given an assignment α, the interpretation function I can be extended to an *interpretation* $[\![t]\!]_\alpha$ for all terms t in the following way:

- $[\![c]\!]_\alpha = I(c)$ for every constant c,
- $[\![x]\!]_\alpha = \alpha(x)$ for every variable x, and
- $[\![tt']\!]_\alpha = [\![t]\!]_\alpha [\![t']\!]_\alpha$.

Thus defined the interpretation function is a homomorphism with respect to syntactic and semantic application.

A term t of type A (that is, an atom) is *true* (*false*) under an assignment α if $V([\![t]\!]_\alpha) = \mathbf{T}$ ($\mathbf{F}$). The valuation V is extended to formulas according to the usual meaning of the logical connectives and quantifiers.

A type structure $\mathcal{D}$ with interpretation functions I and V is called a *model* of higher-order logic program P if it makes true every clause of P under any assigment α. □

As an example, we proceed by a simple lemma stating that the construct $\mathcal{M}_P$ from the previous section indeed yields an initial model of P.

Lemma 5.2 *Let P be a typable higher-order logic program. Consider the type structure $\mathcal{M}_P$ from Definition 4.4, with interpretation function $I_P(c) = [\![c]\!]_P$ for every constant c, and with valuation function $V_P([\![t]\!]_P) = [t]_P$ for every closed term t of type A. Then $\mathcal{M}_P$ is a model of P. Moreover, $\mathcal{M}_P$ is initial in the following sense: for every closed term t of type A, if t is true in $\mathcal{M}_P$, then t is true in every model of P.*

Proof: Use that $[\![\cdot]\!]_P$ is a homomorphism and that $[\![A]\!]_P$ is the characteristic function of M_P. Initiality is proved for $t \in T_P{\uparrow}n$ by induction on n. □

6. The extensional collapse

In this section we define a notion of extensionality for higher-order logic programs and show that for extensional programs the semantics can be considerably simplified. This so-called extensional collapse originates from the model theory of finite type arithmetic and is described and attributed to Zucker in [12].

Definition 6.1 Let P be a typable higher-order logic program, with $|\cdot|_P$ and $[\cdot]_P$ as in Definition 4.1. We define relations $\approx^\sigma_P$ on $|\sigma|_P$, expressing extensional equality of type σ.

$\mathbb{D}$: We put $\approx^\sigma_P$ to be $=$, equality on $|\sigma|_P$, for every $\sigma \in \mathbb{D}$.

$\mathbb{T}$: By induction on $\sigma \in \mathbb{T}$.

For the base type A we put $t \approx_P^A s$ if and only if $[t]_P = [s]_P$.

For the induction steps $\tau' \to \tau$ with $\tau' \equiv D$ or $\tau \in \mathbb{T}$ we define $t \approx_P^{\tau' \to \tau} s$ if and only if $tt' \approx_P^{\tau} ss'$ for all t', s' such that $t' \approx_P^{\tau'} s'$.

We will often omit type superscripts and the subscript P. A closed term t is called *extensional* if $t \approx t$. We call P *extensional* if all closed terms are extensional. □

We give some examples and counterexamples of extensional programs. Proofs are postponed till after Theorem 6.7.

Example 6.2 The following clauses form an extensional program:

```
R(a,b).  % (a,b) in every binary relation D->D->A
call(X):- X.
or(X,Y):- X.
or(X,Y):- Y.
tc(R,XD,YD) :- R(XD,YD).              % tc = transitive closure
tc(R,XD,ZD) :- R(XD,YD),tc(R,YD,ZD). % XD,YD,ZD: D
```

Counterexamples, i.e., examples of non-extensionality, are:

```
eq(Y,Y).  % Y: A
apply(F,Z,F(Z)).  % F: (D->D->A)->D->D->A
```

For example, we have `p` $\approx^A$ `or(p,p)` , but not `eq(p,p)` $\approx^A$ `eq(p,or(p,p))`, and hence not `eq` $\approx^{A\to A\to A}$ `eq`. The non-extensionality of the second clause arises when, for example, one considers a transitive relation `r`, so that `r` has the same extension as `tc(r)`, and `apply(tc,r,tc(r))` holds, whereas `apply(tc,r,r)` does not hold. □

Lemma 6.3 *Let P be a typable higher-order logic program. Then $\approx_P^\sigma$ is a partial equivalence relation for every type σ. More precisely, for every type σ, the relation $\approx_P^\sigma$ is symmetric and transitive (whence reflexive only where defined: $t \approx_P^\sigma s \Rightarrow t \approx_P^\sigma t$ for all closed terms $t, s : \sigma$). If $\approx_P^\sigma$ is reflexive for every σ, then the usual form of extensionality holds: if $tr \approx sr$ for all r, then $t \approx s$.*

Proof: Symmetry and transitivity are proved simultaneously by induction on σ. If reflexivity also holds, then $tr \approx sr$ and $r \approx r'$ imply $tr \approx sr'$. □

Theorem 6.4 *Let P be a typable, extensional, higher-order logic program, with $[\cdot]_P$ as in Definition 4.1. Then we have $t \approx_P^\sigma s$ if and only if $[t]_P = [s]_P$, for every type σ and all closed terms $t, s : \sigma$.*

Proof: Let conditions be as above, in particular P is extensional. Thus we have $t \approx_P^\sigma t$ for all types σ and closed terms t of type σ. For types $\sigma \in \mathbb{D}$ there is nothing to prove, and for $\sigma \in \mathbb{T}$ we proceed by induction. For the base type A the result holds by definition. For the induction step $\tau' \to \tau$ with $\tau' \equiv D$ or $\tau' \in \mathbb{T}$, assume the result has been proved for type $\tau \in \mathbb{T}$. Let $t \approx_P^{\tau' \to \tau} s$, then $tt' \approx_P^\tau st'$ for all closed terms t' of type τ', as all terms are extensional. By the induction hypothesis for type τ we have $[tt']_P = [st']_P$, so $[t]_P t' = [s]_P t'$ for all closed terms t' of type τ'. It follows that $[t]_P = [s]_P$. For the converse, assume $[t]_P = [s]_P$, so $[tt']_P = [t]_P t' = [s]_P t' = [st']_P$ for all closed terms t' of type τ'. By the induction hypothesis for type τ we have $tt' \approx_P^\tau st'$ for all closed terms t' of type τ'. In order to prove $t \approx_P^{\tau' \to \tau} s$, assume $t' \approx_P^{\tau'} s'$ and calculate $tt' \approx_P^\tau st' \approx_P^\tau ss'$, using $s \approx_P^{\tau' \to \tau} s$. □

Corollary 6.5 *Generalize the relations $\approx_P^\sigma$ from $|\sigma|_P$ to $[\![\sigma]\!]_P$ by putting $[\![t]\!]_P \approx_P^\sigma [\![s]\!]_P$ if and only if $t \approx_P^\sigma s$. Under the conditions of the theorem, with $\mathcal{M}_P$ as in Definition 4.4, we then have the following:*

1. *The relations $\approx_P^\sigma$ on $[\![\sigma]\!]_P$ are congruences with respect to application.*
2. *The quotient structure $[\![\sigma]\!]_P/\approx_P^\sigma$ is a model of P that is elementarily equivalent to $\mathcal{M}_P$ with respect to the clausal language of P.* [2]
3. *$[\![\sigma_1 \to \ldots \to \sigma_n \to A]\!]_P/\approx_P$ is isomorphic to a subset of the powerset of $[\![\sigma_1]\!]_P/\approx_P \times \ldots \times [\![\sigma_n]\!]_P/\approx_P$.*

Proof: By the assumption that P is extensional we have that $\approx_P^\sigma$ is an equivalence relation on closed terms of type σ, and hence on $[\![\sigma]\!]_P$, for all σ. Congruence with respect to application holds by definition. This proves 1.

Since equivalent terms have the same extension, we can identify $[\![\sigma]\!]_P/\approx_P^\sigma$ with $\{[t]_P \mid t : \sigma \text{ closed}\}$. (The latter set should not be confused with its superset $[\sigma]_P$.) In this quotient structure, application, interpretation and valuation can be defined in terms of the extensions: $[t]_P[t']_P = [tt']_P$, $I(t) = [t]_P$, $V(t) = [t]_P$. The elementary equivalence of the quotient structure with the original structure follows easily, since both structures make true the same closed atoms. This proves 2.

For every closed term $t : \sigma_1 \to \ldots \to \sigma_n \to A$ we can view $[t]_P$ as a mapping from $[\![\sigma_1]\!]_P/\approx_P \times \ldots \times [\![\sigma_n]\!]_P/\approx_P$ to $\{\mathbf{T}, \mathbf{F}\}$, so as the characteristic function of a subset of $[\![\sigma_1]\!]_P/\approx_P \times \ldots \times [\![\sigma_n]\!]_P/\approx_P$. This proves 3. □

The quotient structure from Corollary 6.5 is called the *extensional collapse* of $\mathcal{M}_P$, anticipated in Example 4.3. The dramatic simplification of the semantics can be explained by comparing Lemma 4.2 to Corollary 6.5, clause 3.

In the remaining part of this section we develop a syntactic criterion which is sufficient for extensionality. The criterion is not necessary. It is important to stress that detemining extensionality is highly undecidable. There will certainly be room for improvement of the criterion presented below. We conjecture in particular that the conditions imposed on the body atoms of a program clause can be alleviated.

In order to acquire some familiarity with proving extensionality, we start with proving that the clauses defining the transitive closure in Example 6.2 are extensional. This boils down to proving $\mathtt{tc} \approx^{(D\to D\to A)\to D\to D\to A} \mathtt{tc}$, so $[\mathtt{tc}(r,s,t)] = [\mathtt{tc}(r',s',t')]$ for all closed terms r, r', s, s', t, t' satisfying $r \approx^{D\to D\to A} r'$, $s \approx^D s'$, $t \approx^D s'$. We shall prove by induction on n the following proposition (omitting type superscripts):

$$\mathtt{tc}(r,s,t) \in T_P{\uparrow}n \Rightarrow \forall r', s', t' \; (r \approx r' \wedge s \approx s' \wedge t \approx t' \Rightarrow \mathtt{tc}(r',s',t') \in T_P{\uparrow}\omega)$$

which entails $\mathtt{tc} \approx \mathtt{tc}$. For $n = 0$ there is nothing to prove. Assume the result has been proved for n and let $\mathtt{tc}(r,s,t) \in T_P{\uparrow}(n{+}1)$ for some closed terms r, s, t of appropriate types and assume $r \approx r' \wedge s \approx s' \wedge t \approx t'$. Since there are two defining clauses for $\mathtt{tc}$, there are two cases to distinguish.

First we consider the case in which $\mathtt{tc}(r,s,t) \in T_P{\uparrow}(n{+}1)$ since $r(s,t) \in T_P{\uparrow}n$. By $r \approx r' \wedge s \approx s' \wedge t \approx t'$ we have $[r(s,t)] = [r'(s',t')]$, so $r'(s',t') \in T_P{\uparrow}\omega$. It follows that $\mathtt{tc}(r',s',t') \in T_P{\uparrow}\omega$.

In the second case we have $\mathtt{tc}(r,s,t) \in T_P{\uparrow}(n{+}1)$ since for some closed term t_s we have $r(s,t_s) \in T_P{\uparrow}n$ and $\mathtt{tc}(r,t_s,t) \in T_P{\uparrow}n$. By $r \approx r' \wedge s \approx s' \wedge t \approx t'$, as well as $t_s \approx t_s$ since t_s is of type D, we have $[r(s,t_s)] = [r'(s',t_s)]$, so $r'(s',t_s) \in T_P{\uparrow}\omega$. Moreover, by the induction hypothesis, we have $\mathtt{tc}(r',t_s,t') \in T_P{\uparrow}\omega$. Again it follows that $\mathtt{tc}(r',s',t') \in T_P{\uparrow}\omega$. This completes the induction step, and hence the proof of $\mathtt{tc} \approx \mathtt{tc}$.

The following definition and theorem generalize the above result.

[2] The quotient structure is in fact an extensional general model in the sense of Henkin [4], cf. the previous footnote.

Definition 6.6 Let P be a typable higher-order logic program. We define a syntactic criterion for atoms, clauses and programs.

- an *atom* $t_0 t_1 \dots t_n : A$ is *good* if t_0 is either a variable or a constant, and for $0 < i \leq n$, if t_i has a type in $\mathbb{T}$, then t_i is a variable.
- a *clause* is *good* if it consists of good atoms, all variables of type in $\mathbb{T}$ in the head are distinct, every variable that occurs in the body but not in the head has type D or the type of a first-order predicate, and if the head atom is of the form $xt_1 \dots t_n$, then the variable x occurs in the body atoms only as head symbol.
- a *program* is *good* if it consists of good clauses. □

Examples of good atoms (clauses, programs) can be found in Example 6.2. The counterexamples there provide reasons for some aspects of the definition of the notions of good atom (clause, program).

Theorem 6.7 *Let P be a typable higher-order logic program. If P is good, then P is extensional.*

Proof: Let P be a typable, good program. We will prove $c \approx c$ for every constant c. Then the extensionality of P follows immediately, since $\approx$ is closed under application.

First we remark that $t \approx t$ holds by definition for all closed terms $t : D$. Furthermore, as $\approx$ for type D coincides with identity, $t \approx t$ also holds for closed terms having the type of a first-order predicate, that is, type $D \to A$, $D \to D \to A$, and so on.

We shall prove by induction on n the following proposition:

$$ct_1 \dots t_k \in T_P{\uparrow}n \Rightarrow \forall t'_1, \dots, t'_k \; (t_1 \approx t'_1 \wedge \dots \wedge t_k \approx t'_k \Rightarrow ct'_1 \dots t'_k \in T_P{\uparrow}\omega),$$

for all constants c simultaneously. For convenience, we will simply ignore arguments of type D in the proof. For $n = 0$ there is nothing to prove. Assume the result has been proved for n and let $ct_1 \dots t_k \in T_P{\uparrow}(n{+}1)$ for some closed terms t_i of appropriate types. Assume $t'_1, \dots, t'_k$ such that $t_1 \approx t'_1 \wedge \dots \wedge t_k \approx t'_k$. For $ct_1 \dots t_k \in T_P{\uparrow}(n{+}1)$, there are basically two possibilities.

The first is a good program clause

$$cx_1 \dots x_k \leftarrow x_i(\vec{x}, \vec{y}), \dots, d(\vec{x}, \vec{y})$$

where $\vec{x}, \vec{y}$ are sequences of variables x_i and y_j that may occur as (multiple) arguments in the good atoms of the body, and d is a constant, possibly $d \equiv c$. The variables y_j occur in the body but not in the head. Atoms with head symbol y_j are unproblematic. The two body atoms shown correspond exactly to the two forms of a good atom. Let $t_i(\vec{t}, \vec{s}), \dots, d(\vec{t}, \vec{s}) \in T_P{\uparrow}n$ for suitable closed terms $\vec{s}$. By the assumption on the types of the variables y_j it follows that $\vec{s} \approx \vec{s}$ (componentwise). Since we also have $\vec{t} \approx \vec{t'}$, it follows that $t_i(\vec{t}, \vec{s}) \approx t'_i(\vec{t'}, \vec{s})$, so $t'_i(\vec{t'}, \vec{s}) \in T_P{\uparrow}\omega$. Moreover, by the induction hypothesis, it follows that $d(\vec{t'}, \vec{s}) \in T_P{\uparrow}\omega$, and hence $ct'_1 \dots t'_k \in T_P{\uparrow}\omega$.

The second possibility is a good program clause

$$x_p \dots x_k \leftarrow x_i(\vec{x}, \vec{y}), \dots, d(\vec{x}, \vec{y})$$

with $\vec{x} \equiv x_{p+1} \dots x_k$ and $\vec{y}$ as above, and $0 \leq p \leq i \leq k$. Now the argument for $ct'_1 \dots t'_k \in T_P{\uparrow}\omega$ is very similar as above, with x_p bound to $ct_1 \dots t_p$ and $ct'_1 \dots t'_p$, respectively. □

Acknowledgements
I am grateful to Krzysztof Apt for providing the opportunity to work on the subject of higher-order logic programming at CWI (Amsterdam), and for many discussions on earlier versions of this paper.

Thanks to an anonymous referee the paper [13] by Wadge came to my notice. Its result may be rendered as follows: every *definitional* higher-order logic program has a minimal *standard* model. In a standard model, see also [4], $[\![A]\!]$ is $\{\mathbf{T}, \mathbf{F}\}$ and $[\![\sigma{\rightarrow}\tau]\!]$ is the full function space $[\![\sigma]\!] \rightarrow [\![\tau]\!]$ (hence standard models are extensional, but uncountable if $[\![D]\!]$ is infinite). The notions 'definitional' and 'good' are very similar in spirit, but mathematically different. For example, the body of a good clause may contain local higher-order variables, provided their types do not surpass the type of a first-order predicate. Hence the transitive closure of a binary predicate of unary predicates is good but not definitional. All Wadge's examples of definitional programs are good programs in our sense, but his definition appears to include programs beyond this. Furthermore, Wadge's argument is model theoretic, whereas ours is proof theoretic. We are happy to take over his plea for extensionality and its relevance for software engineering. For future research we conjecture that the result of [13] can be extended from definitional programs to extensional programs as defined here.

References

1. K. R. Apt. Logic programming. In J. van Leeuwen (ed.) *Handbook of Theoretical Computer Science*, Vol. B, pp. 493–574. Elsevier, Amsterdam, 1990.
2. W. Chen, M. Kifer and D.S. Warren. Hilog: a foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.
3. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
4. L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15:81–91, 1950.
5. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* MIT Press, Cambridge, Massachusetts, 1994.
6. J.R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the AMS*, 146:29–60, 1969.
7. J.R. Hindley. *Basic simple type theory.* Cambridge tracts in TCS 42, CUP, 1997.
8. G. Nadathur and D.A. Miller. An overview of λProlog. In K. Bowen and R. Kowalski (eds.) *Proceedings of the Fifth International Conference on Logic Programming*, pp. 810–827, Seattle. MIT Press, Cambridge, Massachusetts, 1988.
9. G. Nadathur and D.A. Miller. Higher-order logic programming. In D. Gabbay e.a. (eds.) *Handbook of logic in artificial intelligence*, Vol. 5, pp. 499–590. Clarendon Press, Oxford, 1998.
10. F. Pfenning (ed.). *Types in Logic Programming.* MIT Press, Cambridge, Massachusetts, 1992.
11. Z. Somogyi, F.J. Henderson and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.
12. A.S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis.* Number 344 in Lecture Notes in Mathematics. Springer-Verlag, Berlin, 1973.
13. W.W. Wadge. Higher-order Horn logic programming. In V. Saraswat and K. Ueda (eds.) *Proceedings of the 1991 International Symposium on Logic Programming*, pp. 289–303. MIT Press, Cambridge, Massachusetts, 1991.