Dasseville, Ingmar; Van Der Hallen, Matthias; Bogaerts, Bart; Janssens, Gerda; Denecker, Marc

A compositional typed higher-order logic with definitions

Published in:
Technical Communications of the 32nd International Conference on Logic Programming

DOI:
10.4230/OASIcs.ICLP.2016.14

Published: 01/11/2016

Document Version
Publisher's PDF, also known as Version of record

A Compositional Typed Higher-Order Logic with Definitions*

Ingmar Dasseville[1], Matthias van der Hallen[†1], Bart Bogaerts[‡3], Gerda Janssens[1], and Marc Denecker[1]

1 **KU Leuven – University of Leuven, Celestijnenlaan 200A, Leuven, Belgium**
`ingmar.dasseville@cs.kuleuven.be`
2 **KU Leuven – University of Leuven, Celestijnenlaan 200A, Leuven, Belgium**
`matthias.vanderhallen@cs.kuleuven.be`
3 **Helsinki Institute for Information Technology HIIT, Aalto University, Aalto, Finland; and**
KU Leuven – University of Leuven, Celestijnenlaan 200A, Leuven, Belgium
`bart.bogaerts@cs.kuleuven.be`
4 **KU Leuven – University of Leuven, Celestijnenlaan 200A, Leuven, Belgium**
`gerda.janssens@cs.kuleuven.be`
5 **KU Leuven – University of Leuven, Celestijnenlaan 200A, Leuven, Belgium**
`marc.denecker@cs.kuleuven.be`

Abstract

Expressive KR languages are built by integrating different language constructs, or extending a language with new language constructs. This process is difficult if non-truth-functional or non-monotonic constructs are involved. What is needed is a compositional principle.

This paper presents a compositional principle for defining logics by modular composition of logical constructs, and applies it to build a higher order logic integrating typed lambda calculus and rule sets under a well-founded or stable semantics. Logical constructs are formalized as triples of a syntactical rule, a semantical rule, and a typing rule. The paper describes how syntax, typing and semantics of the logic are composed from the set of its language constructs. The base semantical concept is the *infon*: mappings from structures to values in these structures. Semantical operators of language constructs operate on infons and allow to construct the infons of compound expressions from the infons of its subexpressions. This conforms to Frege's principle of compositionality.

1998 ACM Subject Classification I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases Logic, Semantics, Compositionality

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.14

1 Introduction

Expressive knowledge representation languages consist of many different language constructs. New KR languages are often built by adding new (possibly nestable) language constructs to

* This research was supported by the project GOA 13/010 Research Fund KU Leuven and projects G.0489.10, G.0357.12 and G.0922.13 of FWO (Research Foundation – Flanders).

† Matthias van der Hallen is supported by a Ph.D. fellowship from the Research Foundation – Flanders (FWO – Vlaanderen).

‡ Bart Bogaerts is supported by the Finnish Center of Excellence in Computational Inference Research (COIN) funded by the Academy of Finland (grant #251170).

existing logics. Principled compositional methods are desired that allow to construct logics from language constructs, or incrementally extend an existing logic with a new construct, while preserving the meaning of the remaining language constructs. This is known as Frege's compositionality principle.

In classical monotone logics it is common practice to extend a logic with new language constructs or connectives by specifying an additional pair of a syntactical and semantical rule. E.g., we can add a cardinality construct to classical first order logic (with finite structures) by defining:

- syntactical rule: $\#(\{x : \varphi\})$ is a (numerical) term if x is a variable and φ a formula;
- semantical rule: $(\#(\{x : \varphi\}))^I = \#(\{d \in \mathcal{D}^I \mid I[x : d] \models \varphi\})$, the cardinality of the set of domain elements that correspond to the set expression. Here, $\mathcal{D}^I$ is the domain of I.

The ease and elegance of this is beautiful. In the context of nonmonotonic languages such as logic programming and extensions such as answer set programming [18, 17, 20] and the logic FO(ID) (classical logic with inductive definitions) [7], the situation is considerably more complex. For example, adding aggregates to these logics required, and still requires a serious effort [15, 25, 10, 22, 21, 9, 11] and resulted in a great diversity of logics.

In this paper, we propose a compositional principle to build logics, and apply it to build a logic $\mathbb{L}$ integrating typed higher order lambda calculus with definitions represented as rule sets under well-founded semantics. The two main contributions of this work are:

- It introduces a compositional principle to build and integrate logics and puts it to the test: by building an expressive logic including rule sets, with aggregates, lambda expressions, higher order rules, rule sets to express definitions by monotone, well-founded and iterated induction, definitions nested in rules, ... The semantical basis is the concept of *infon* which provides a semantical abstraction of the meaning of expressions and is related to intensional objects in intensional logics [14] .
- The logic itself brings together the logics of logic programming and descendants such as answer set programming and FO(ID), and the logic of typed lambda calculus which has become the foundational framework for formal specification languages and functional programming. We illustrate the application of the resulting logic to build simple and elegant theories that express complex knowledge.

2 Related Work

2.1 Logics

Our paper on templates [5] introduced a simpler version of the framework from the current paper, using informal notions. There, the framework was used to construct a logic permitting inductive definitions within the body of other inductive definitions. In that logic, *templates* are (possibly inductive) second order definitions that allow nesting inductive definitions; this nesting is required to build, for instance, templates defining one predicate parameter as the transitive closure of another parameter. In this paper, we present the framework with a more formal basis, using the concept of *infons* as the mathematical object corresponding to the semantics of a language construct, and identify the notion of *Frege's principle of compositionality* as the underlying goal of the framework.

This paper explicitly allows the construction of *higher-order* logics. In the context of meta-programming [1], some logics with higher-order syntax already exist. One such example is HiLog [4], which combines a higher-order syntax with first-order semantics. HiLogs main motivation for this is to introduce a useful degree of higher order while maintaining decidability of the deduction inference. Another example is λprolog [19], which extends

Prolog with (among others) higher-order functions, λ-terms, higher-order unification and polymorphic types. To achieve this, λprolog extends the classical first-order theory of Horn-clauses to the intuitionistic higher-order theory of *Hereditary Harrop* formulas [16].

The algebra of modular system (AMS) [23, 24] is a framework in which arbitrary logics with a model semantics can be combined. The difference with our work is that in AMS, connectives from the different logics cannot be combined arbitrarily. Instead, there is a fixed set of connectives (a "master" logic) that can be used to combine expressions from different logics. Compared to our logic, this has advantages and disadvantages. One advantage is that AMS only requires a two-valued semantics (an infon) to be specified for a given logic, making it more easily applicable to a wide range of logics. A disadvantage is that it does not allow for interactions between the different connectives.

2.2 Infons

The concept of infon in the sense used in this paper is related to intensional objects in Montague's intensional logic [14]. Intensional logic studies the dichotomy between the designation and the meaning of expressions. Intensional objects are represented by lambda expressions and model functions from states to objects similar to our infons. The term "infon" was used by other authors in other areas. In situation semantics [2], infons intuitively represent "quantums of information" [8]. Although such an infon has a different mathematical form than an infon in our theory, it determines a characteristic function from *situations* (which are approximate representations of states, similar to approximate structures) to true, false (or undetermined), which intuitively corresponds to an infon. Situation semantics, the semantics supported by situation theory, provides a foundation for reasoning about real world situations and the derivations made by common sense. In [13], infons are "statements viewed as containers of information" and an (intuitionistic) logic of infons is built for the specific purpose of modelling distributed knowledge authorization.

3 Preliminaries

3.1 Cartesian product, powerset, product, pointwise extension and lifting

The *powerset* operator $\mathcal{P}(\cdot)$ maps a set X to its powerset $\mathcal{P}(X)$. The *power* operator $(\cdot)^{(\cdot)}$ maps pairs (I, Y) of sets to the set Y^I of all functions with domain I and co-domain Y. We denote the function with domain D and co-domain C that maps elements $x \in D$ to the value of a mathematical expression $exp[x]$ in variable x as $\lambda : D \to C : x \mapsto exp[x]$ (using λ as the anonymous function symbol as in lambda calculus). Or, if the co-domain is clear from the context, as $\lambda x \in D : exp[x]$. When $exp[x]$ is Boolean expression, this is also denoted as a set comprehension $\{x \in D \mid exp[x]\}$.

We define the set of truth values $\mathcal{T}wo = \{\mathbf{f}, \mathbf{t}\}$; here $\mathbf{t}$ stands for "true" and $\mathbf{f}$ for "false". For any X, $\mathcal{P}(X)$ is isomorphic to $\mathcal{T}wo^X$, using the mapping from a set to its characteristic function. In the rest of the paper, we will identify $\mathcal{P}(\cdot)$ with $\mathcal{T}wo^{(\cdot)}$.

We frequently use $\langle x_i \rangle_{i \in I}$ to denote the function $\lambda : I \to \{x_i \mid i \in I\} : i \to x_i$. We call this an indexed set (with index set I). Let $\langle V_i \rangle_{i \in I}$ be an indexed set of sets, i.e., each V_i is a set. Its *product set*, denoted $\times_{i \in I} V_i$, is the set of all indexed sets $\langle x_i \rangle_{i \in I}$ such that $x_i \in V_i$ for each $i \in I$. This generalizes Cartesian product $V_1 \times \cdots \times V_n$ (taking $I = \{1, \ldots, n\}$).

Let $\langle \leq_i \rangle_{i \in I}$ be an indexed set of partial order relations $\leq_i$ on sets V_i for each $i \in I$. The *product order* of $\langle \leq_i \rangle_{i \in I}$ is the binary relation

$$\{(\langle v_i \rangle_{i \in I}, \langle w_i \rangle_{i \in I}) \in (\times_{i \in I} V_i)^2 \mid \forall i \in I : v_i \leq_i w_i\}.$$

It is a binary relation on $\times_{i \in I} V_i$. Written differently, it is the Boolean function:

$$\lambda : (\times_{i \in I} V_i)^2 \to \mathcal{T}wo : (\langle v_i \rangle_{i \in I}, \langle w_i \rangle_{i \in I}) \mapsto \wedge_{i \in I}(v_i \leq_i w_i)\,.$$

A special case is if all V_i and $\leq_i$ are the same, i.e., for some V and $\leq$, it holds that $V_i = V$ and $\leq_i = \leq$ for each $i \in I$. Then the product relation $\times_{i \in I} \leq$ will be called the *pointwise extension* of $\leq$ on $V^I = \times_{i \in I} V$. Taking products of orders preserves many good properties of its component orders. It is well-known that the product order is a partial order. The product order of chain complete orders is chain complete order and the product order of complete lattice orders is a complete lattice order.

Let $\langle O_i \rangle_{i \in I}$ be an indexed set of operators $O_i \in X_i^{V_i}$. Then we define the lift operator $\uparrow_{i \in I} O_i$ as the operator in $(\times_{i \in I} X_i)^{(\times_{i \in I} V_i)}$ that maps elements $\langle v_i \rangle_{i \in I}$ to $\langle O_i(v_i) \rangle_{i \in I}$. In another notation, it is the function:

$$\lambda : \times_{i \in I} V_i \to \times_{i \in I} X_i : \langle v_i \rangle_{i \in I} \mapsto \langle O_i(v_i) \rangle_{i \in I}\,.$$

A special case arises when all O_i are the same operator $O : V \to V$. In this case, $\uparrow_{i \in I} O$ is a function in $\times_{i \in I} V = V^I$ mapping $\langle v_i \rangle_{i \in I}$ to $\langle O(v_i) \rangle_{i \in I}$. That is, it is the function $\lambda : V^I \to V^I : f \mapsto O \circ f$. We call this the *lifting* of $O : V \to V$ to the product V^I.

3.2 (Approximation) Fixpoint Theory

A binary relation $\leq$ on set V is a *partial order* if $\leq$ is reflexive, transitive and asymmetric. In that case, we call the mathematical structure $\langle V, \leq \rangle$ a *poset*. $\leq$ is *total* if for every $x, y \in V$, $x \leq y$ or $y \leq x$. The partial order $\leq$ is a *complete lattice order* if for each $X \subseteq V$, there exists a least upperbound $lub(X)$ and a greatest lower bound $glb(X)$. If $\leq$ is a complete lattice order of V, then V has a least element $\bot$ and a greatest element $\top$.

Let $\langle V, \leq \rangle, \langle W, \leq \rangle$ be two posets. An operator $O : V \to W$ is monotone if it is order preserving; i.e. if $x \leq y \in V$ implies $O(x) \leq O(y)$.

Let $\langle V, \leq \rangle$ be complete lattice with least element $\bot$ and greatest element $\top$. Its bilattice is the structure $\langle V^2, \leq_p, \leq \rangle$ with $(v_1, v_2) \leq_p (w_1, w_2)$ if $v_1 \leq w_1, v_2 \geq w_2$ and $(v_1, v_2) \leq (w_1, w_2)$ if $v_1 \leq w_1, v_2 \leq w_2$. The latter is the pointwise extension of $\leq$ to the bilattice. Both orders are known to be lattice orders. $\leq_p$ is called the *precision order*. The least precise element is $(\bot, \top)$ and most precise element is $(\top, \bot)$. An *exact* pair is of the form (v, v). A *consistent* pair (v, w) is one such that $v \leq w$. We say that (v, w) approximates $u \in V$ if $v \leq u \leq w$. The set of values approximated by (v, w) is $[v, w]$. This set is non-empty iff (v, w) is consistent. Exact pairs (V, V) are the maximally consistent pairs and they approximate a singleton $\{X\}$. We view the exact pairs as the embedding of V in V^2. Abusing this, we sometimes write v where (v, v) should be written. Pairs $(v, w) \in V^2$ are written as $\mathfrak{v}$, with $(\mathfrak{v})_1 = v, (\mathfrak{v})_2 = w$.

We define $V^c = \{(v, w) \in V^2 \mid v \leq w\}$. It is the set of consistent pairs. We will call such a pair an approximate value, and we call V^c the *approximate value space* of V. It can be shown that any non-empty set $X \subseteq V^c$ has a greatest lower bound $glb_{\leq_p}(X)$ in V^c, but not every set $X \subseteq V^c$ has a least upperbound in V^c. In particular, the exact elements are exactly the maximally precise elements. Hence, V^c is not a complete lattice. However, if X has an upperbound in V^c, then $lub(X)$ exists. Also, V^c is chain complete: every totally ordered subset $X \subseteq V^c$ has a least upperbound. It follows that each sequence $\langle (v_i, w_i) \rangle_{i < \alpha}$ of increasing precision has a least upperbound $lub(\langle (v_i, w_i) \rangle_{i < \alpha})$, called its limit. This suffices to warrant the existence of a least fixpoint for every $\leq_p$-monotone operator $O : V^c \to V^c$.

▶ **Example 1.** Consider the lattice $\mathcal{T}wo = \{\mathbf{t}, \mathbf{f}\}$ with $\mathbf{f} \leq \mathbf{t}$. The four pairs of its billatice $\mathcal{F}our$ correspond to the standard truth values of four-valued logic. The pairs $(\mathbf{t}, \mathbf{t})$ and $(\mathbf{f}, \mathbf{f})$

are the embeddings of true ($\mathbf{t}$) and false ($\mathbf{f}$) respectively. The pair $(\mathbf{f}, \mathbf{t})$ represents unknown ($\mathbf{u}$) and $(\mathbf{t}, \mathbf{f})$ represents the inconsistent value ($\mathbf{i}$). Here, the set $\mathcal{T}wo^c$ is the set of consistent pairs and is denoted $\mathcal{T}hree$. The precision order is $\mathbf{u} \leq_p \mathbf{t} \leq_p \mathbf{i}, \mathbf{u} \leq_p \mathbf{f} \leq_p \mathbf{i}$ and the product order is $\mathbf{f} \leq \mathbf{u} \leq \mathbf{t}, \mathbf{f} \leq \mathbf{i} \leq \mathbf{t}$.

For any lattice $\langle V, \leq \rangle$ and domain D, the pointwise extension of $\leq$ to V^D is a lattice order, also denoted as $\leq$. The lattice V^D has a bilattice $(V^D)^2$ and approximate value space $(V^D)^c$ which are isomorphic to $(V^2)^D$, respectiely $(V^c)^D$.

▶ **Example 2.** The billattice of $\mathcal{T}wo^D$ and the approximation space $(\mathcal{T}wo^D)^c$ are isomorphic to $\mathcal{F}our^D$, respectively $\mathcal{T}hree^D$ under the pointwise extensions of $\leq_p$ and $\leq$ of $\mathcal{F}our$ and $\mathcal{T}hree$. Elements of $\mathcal{F}our^D$ and $\mathcal{T}hree^D$ correspond to four and three valued sets.

Let D, C be complete lattices.

▶ **Definition 3.** For any function $f : D \to C$, we say that $A : D^c \to C^c$ is an *approximator* of f if (1) ($\leq_p$-monotonicity) A is $\leq_p$-monotone and (2) (exactness) for each $v \in D$, $A(v) \leq_p f(v)$. We call A *exact* if A preserves exactness. The projections of $A(v, w)$ on first and second argument are denoted $A(v, w)_1$ and $A(v, w)_2$.

Approximators of f allow to infer approximate output from approximate input for f. The co-domain of an approximator is equipped with a precision order which can be pointwise extended on $(C^c)^{D^c}$.

▶ **Definition 4.** We say that F is the *ultimate approximator* of f if F is the $\leq_p$-maximally precise approximator of f. We denote F as $\lceil f \rceil$.

One can prove that $\lceil f \rceil(\mathfrak{v}) = glb_{\leq_p}(\{f(v) \mid \mathfrak{v} \leq_p v \in D\})$.

▶ **Example 5.** The ultimate approximators of the standard Boolean functions $\wedge, \neg, \vee, \ldots,$ correspond to the standard 3-valued Boolean extensions known from the Kleene truth assignment. E.g. $\lceil \wedge \rceil$:

$\lceil \wedge \rceil$	$\mathbf{f}$	$\mathbf{u}$	$\mathbf{t}$
$\mathbf{f}$	$\mathbf{f}$	$\mathbf{f}$	$\mathbf{f}$
$\mathbf{u}$	$\mathbf{f}$	$\mathbf{u}$	$\mathbf{u}$
$\mathbf{t}$	$\mathbf{f}$	$\mathbf{u}$	$\mathbf{t}$

Let $\langle V, \leq \rangle$ be complete lattice with least element $\perp$ and greatest element $\top$. With an operator $O : V \to V$, many sorts of fixpoints can be associated: the standard fixpoints $O(x) = x$ and the *grounded* fixpoints of O [3]. For any approximator $A : V^c \to V^c$, more sorts of fixpoints can be defined:

- The A-Kripke-Kleene fixpoint is the $\leq_p$-least fixpoint of A.
- A partial A-stable fixpoint is a pair (x, y) such that
  - $A(x, y) = (x, y)$,
  - (x, y) is prudent, i.e., for all $z \leq y$, $A(z, y)_1 \leq y$ implies $x \leq z$.
  - there is no $z \in [x, y[$ such that $A(x, z)_2 \leq z$.
- The well-founded fixpoint of A is the least precise A-partial stable fixpoint.
- An A-stable fixpoint is an element $v \in L$ such that (v, v) is a partial A-stable fixpoint.

Assume A approximates O. It is well-known that the KK-fixpoint of A approximates all fixpoints of O and all partial stable fixpoints of A, hence also the well-founded fixpoint of A and the (exact) stable fixpoints of A. It can be shown that the three-valued immediate consequence operator of logic programs is an approximator of the two-valued one, and that the above sorts of fixpoints induce the different sorts of semantics of logic programming [6].

With a lattice operator $O : V \to V$, we define the ultimate well-founded fixpoint and the ultimate (partial) stable fixpoints as the well-founded fixpoint and the (partial) stable fixpoints of $\lceil O \rceil$. Compared with other approximators A of O, the ultimate approximator has the most precise KK-fixpoint and well-founded fixpoint, and -somewhat surprisingly- the most (exact) stable fixpoints. That is, the set of exact stable fixpoints of any approximator A of O is a subset of that set of $\lceil O \rceil$. Notice that the ultimate well-founded fixpoint of O is an element of the bilattice, but it may be (and often is) exact.

4 A typed higher order logic $\mathbb{L}$ with (nested) definitions

4.1 Type system

A typed logic $\mathbb{L}$ contains a type system, offering a method to expand arbitrary sets $\mathbb{B}$ of (user-defined) type symbols to a set $\mathbb{T}(\mathbb{B})$ of types, together with a method to expand a type structure $\mathcal{A}$ assigning sets of values to the symbols of $\mathbb{B}$, to an assignment $\bar{\mathcal{A}}$ of sets of values to all types in $\mathbb{T}(\mathbb{B})$. We formalize these concepts.

▶ **Definition 6.** A type vocabulary $\mathbb{B}$ is a (finite) set of type symbols. A type structure $\mathcal{A}$ for $\mathbb{B}$ is an assignment of sets $\tau^{\mathcal{A}}$ to each $\tau \in \mathbb{B}$.

▶ **Definition 7.** A *type constructor* is a pair (tc, Sem_{tc}) of a type constructor symbol tc of some arity $n \geq 0$ and its associated semantic function Sem_{tc} which maps n-tuples of sets to sets such that Sem_{tc} preserves set isomorphism. [1]

Given a set $\mathbb{B}$ of type symbols and a set of type constructor symbols, a set of (finite) type terms τ can be built from them. In general, the set $\mathbb{T}(\mathbb{B})$ of types of a logic theory form a subset of the set of these type terms.

▶ **Definition 8.** A type system consists of a set of type constructors and a function mapping any set $\mathbb{B}$ of type symbols to a set $\mathbb{T}(\mathbb{B})$ of type terms formed from $\mathbb{B}$ and the type constructor symbols such that for any bijective renaming $\theta : \mathbb{B} \to \mathbb{B}'$, $\mathbb{T}(\mathbb{B})$ and $\mathbb{T}(\mathbb{B}')$ are identical modulo the renaming θ. An element of $\mathbb{T}(\mathbb{B})$ is called a *type*. A *compound type* is an element of $\mathbb{T}(\mathbb{B}) \setminus \mathbb{B}$.

For a given type system, it is clear that any type structure $\mathcal{A}$ for $\mathbb{B}$ can be expanded in a unique way to all type terms by iterated application of the semantic functions Sem_{tc}.

▶ **Definition 9.** Given a type system and a type structure $\mathcal{A}$ for a set $\mathbb{B}$ of type symbols, we define $\bar{\mathcal{A}}$ as the unique expansion of $\mathcal{A}$ to $\mathbb{T}(\mathbb{B})$ defined by induction on the structure of type terms and using the semantic type constructor functions Sem_{tc} of type constructors.

By slight abuse of notation, we write $\tau^{\bar{\mathcal{A}}}$ as $\tau^{\mathcal{A}}$.

▶ **Definition 10.** We call a type system *type closed* if for every $\mathbb{B}$, $\mathbb{T}(\mathbb{B})$ is the set of all type terms built over $\mathbb{B}$ and the type constructors of the system.

▶ **Example 11.** The type system of the logic that we will define below is type closed. Its type constructor symbols and corresponding semantic type operators are:

[1] That is, if there exists bijections between S_1 and S'_1, ..., S_n and S'_n, then there is a bijection between $Sem_{tc}(S_1, \ldots, S_n)$ and $Sem_{tc}(S'_1, \ldots, S'_n)$.

- the 0-ary Boolean type constructor symbol BOOL with $Sem_{\mathsf{BOOL}} = \mathcal{T}wo$;
- the 0-ary natural number constructor type symbol NAT with $Sem_{NAT} = \mathbb{N}$;
- the n-ary Cartesian product type constructor symbol $\times_n$; we write $\times_n(\tau_1, \dots, \tau_n)$ as $\tau_1 \times \cdots \times \tau_n$ and $\times_n(\tau, \dots, \tau)$ as τ^n. The semantic operator $Sem_{\times_n}$ maps tuples of sets $(S_1, \dots, S_n)$ to the Cartesian product $S_1 \times \cdots \times S_n$;
- the function type constructor $\rightarrow$ with $Sem_{\rightarrow}$ mapping pairs of sets (X, Y) to the function set Y^X.

In typed lambda calculus, Cartesian product is often not used (it can be simulated using higher order functions and *currying*). Here, we keep it in the language to connect easier with FO.

▶ **Example 12.** The type system of typed classical first order logic uses the type constructors corresponding to BOOL, $\times_n$ and $\rightarrow$ in the previous example. $\mathbb{T}(\mathbb{B})$ is the set $\{\tau_1 \times \cdots \times \tau_n \rightarrow \mathsf{BOOL}, \tau_1 \times \cdots \times \tau_n \rightarrow \tau \mid \tau_1, \dots, \tau_n, \tau \in \mathbb{B}\}$. It consists of first order predicate types $\tau_1 \times \cdots \times \tau_n \rightarrow \mathsf{BOOL}$ and first order function types $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$. The type system of untyped classical first order logic is obtained by fixing $\mathbb{B} = \{U\}$, where U represents the universe of discourse. Clearly, (typed) FO is not type closed.

From now on, we assume a fixed type system. We also assume an infinite supply of type symbols, and for all types τ that can be constructed from this supply and the given type constructor symbols, an infinite supply of symbols σ of type τ. We write $\sigma : \tau$ to denote that τ is the type of σ.

▶ **Definition 13.** A *vocabulary* (or *signature*) Σ is a tuple $\langle \mathbb{B}, Sym \rangle$ with $\mathbb{B}$ a set of type symbols, Sym a set of symbols σ of type $\tau \in \mathbb{T}(\mathbb{B})$.

We write $\mathbb{T}(\Sigma)$ to denote $\mathbb{T}(\mathbb{B})$.

Let Σ be a vocabulary $\langle \mathbb{B}, Sym \rangle$.

▶ **Definition 14.** An *assignment* to Sym in type structure $\mathcal{A}$ for $\mathbb{B}$ is a mapping $A : Sym \rightarrow \{\tau^{\mathcal{A}} \mid \tau \in \mathbb{T}(\Sigma)\}$ such that for each $\sigma : \tau \in Sym$, $\sigma^A \in \tau^{\mathcal{A}}$. That is, the value of σ^A is of type τ in $\mathcal{A}$. The set of assignments to Sym in $\mathcal{A}$ is denoted $Assign^{\mathcal{A}}_{Sym}$.

▶ **Definition 15.** A Σ-*structure* I is a tuple $\langle \mathcal{A}, (\cdot)^I \rangle$ of a type structure $\mathcal{A}$ for $\mathbb{B}$, and $(\cdot)^I$ an *assignment* to all symbols $\sigma \in Sym$ in type structure $\mathcal{A}$. We denote the value of σ as σ^I. The class of all Σ-structures is denoted $\mathbb{S}(\Sigma)$.

We frequently replace $\mathcal{A}$ by I; e.g., we may write τ^I for $\tau^{\mathcal{A}}$.

Let Σ be a vocabulary with type symbols $\mathbb{B}$, I a Σ-structure. Let Sym be a set of symbols with types in $\mathbb{T}(\mathbb{B})$ (it may contain symbols not in Σ). For any assignment $A \in Assign^I_{Sym}$ to Sym in (the type structure of) I, we denote by $I[A]$ the structure that is identical to I except that for every $\sigma \in Sym$, $\sigma^{I[A]} = \sigma^A$. This is a structure of the vocabulary $\Sigma \cup Sym$. As a shorthand notation, let σ be a symbol of type τ and v a value of type τ in I, then $[\sigma : v]$ is the assignment that maps σ to v, and $I[\sigma : v]$ is the updated structure.

▶ **Definition 16.** A Σ-*infon* $\mathfrak{i}$ of type $\tau \in \mathbb{T}(\Sigma)$ is a mapping that associates with each Σ-structure I a value $\mathfrak{i}(I)$ of type τ in I. The class of Σ-infons is denoted $\mathfrak{I}_\Sigma$. Each symbol $\sigma \in \Sigma$ of type τ defines the Σ-infon $\mathfrak{i}_\sigma$ of type τ that associates with each Σ-structure I the value σ^I.

Infons of type τ are similar to intensional objects in Montague's intensional logic [14]. An infon of type BOOL provides an abstract syntax independent representation of a *quantum of information*. It maps a structure representing a possible state of affairs in which the information holds to true, and other structures to false. It will be the case that two sentences are logically equivalent in the standard sense iff they induce the same infon.

4.2 Language constructs

▶ **Definition 17.** A language construct $\mathfrak{C}$ consists of an arity n representing the number of arguments, a typing rule $Type_{\mathfrak{C}}$ specifying the allowable argument types and the corresponding expression type, and a semantic operator $Sem_{\mathfrak{C}}$. A *typing rule* $Type_{\mathfrak{C}}$ is a partial function from n argument types $\tau_1, \ldots, \tau_n$ to a type $Type_{\mathfrak{C}}(\tau_1, \ldots, \tau_n) = \tau$ that preserves renaming of type symbols; i.e., if θ is a bijective renaming of type symbols, then $Type_{\mathfrak{C}}(\theta(\tau_1), \ldots, \theta(\tau_n)) = \theta(\tau)$. If $Type_{\mathfrak{C}}$ is defined for $\tau_1, \ldots, \tau_n$, we call $\tau_1, \ldots, \tau_n$ an argument type for $\mathfrak{C}$. The semantic operator $Sem_{\mathfrak{C}}$ is a partial mapping defined for all tuples of infons $\mathfrak{i}_1, \ldots, \mathfrak{i}_n$ of all argument types $\tau_1, \ldots, \tau_n$ for $\mathfrak{C}$ to an infon of the corresponding expression type τ.

A language construct $\mathfrak{C}$ takes a sequence of expressions $e_1, \ldots, e_n$ as argument and yields the compound expression $\mathfrak{C}(e_1, \ldots, e_n)$. This determines the abstract syntax of expressions. We often specify a concrete syntax for $\mathfrak{C}$ (which often disagrees with the abstract syntax).

Let $\tau_1, \ldots, \tau_n$ be an argument type for $\mathfrak{C}$ yielding the expresson type τ. Then for well-typed expressions $e_1, \ldots, e_n$ of respectively types $\tau_1, \ldots, \tau_n$, the (abstract) compound expression $\mathfrak{C}(e_1, \ldots, e_n)$ is well-typed and of type τ. Some language constructs are polymorphic and apply to expressions of many types. Others have unique type for each argument.

▶ **Example 18.** The tupling operator TUP is a polymorphic language construct that maps expressions $e_1, \ldots, e_n$ of arbitrary types $\tau_1, \ldots, \tau_n$ to the compound expression $TUP(e_1, \ldots, e_n)$ of type $\tau_1 \times \cdots \times \tau_n$. The concrete syntax is $(e_1, \ldots, e_n)$.

The conjunction $\wedge$ maps expressions e_1, e_2 of type BOOL to $\wedge(e_1, e_2)$ of type BOOL. The concrete syntax is $e_1 \wedge e_2$.

The set of language constructs of a logic $\mathbb{L}$ together with a vocabulary Σ uniquely determines the set $Exp^{\mathbb{L}}_{\Sigma}$ of well-typed expressions over Σ, as well as a function $Type_{\mathbb{L}} : Exp^{\mathbb{L}}_{\Sigma} \to \mathbb{T}(\Sigma)$. Formally, consider the set of (finite) labeled trees with nodes labeled by language constructs of $\mathbb{L}$ and symbols of Σ. Within this set, the function $Type_{\mathbb{L}}$ is defined by induction on the structure of expressions

- $Type_{\mathbb{L}}(\sigma) = \tau$ if $\sigma \in \Sigma$ is a symbol of type τ;
- $Type_{\mathbb{L}}(\mathfrak{C}(e_1, \ldots, e_n)) = Type_{\mathfrak{C}}(Type_{\mathbb{L}}(e_1), \ldots, Type_{\mathbb{L}}(e_n))$.

This mapping $Type_{\mathbb{L}}$ is a partial function, the domain of which is exactly $Exp^{\mathbb{L}}_{\Sigma}$.

Furthermore, the set of language constructs of $\mathbb{L}$ determines for each well-typed expression $e \in Exp^{\mathbb{L}}_{\Sigma}$ of type τ a unique infon $Sem_{\mathbb{L}}(e)$ of that type. The function $Sem_{\mathbb{L}}$ is defined by induction on the structure of expressions by the following equation:

- $Sem_{\mathbb{L}}(\sigma) = \mathfrak{i}_{\sigma}$ if $\sigma \in \Sigma$;
- $Sem_{\mathbb{L}}(\mathfrak{C}(e_1, \ldots, e_n)) = Sem_{\mathfrak{C}}(Sem_{\mathbb{L}}(e_1), \ldots, Sem_{\mathbb{L}}(e_n))$.

This property warrants a strong form of Frege's compositionality principle.

We call a logic *substitution closed* if every expression of some type may occur at any argument position of that type. E.g., propositional logic and first order logic are substitution closed, but CNF is not due to the syntactical restrictions on the format of CNF formulas.

4.2.1 Simply typed lambda calculus with infon semantics

Below, we introduce a concrete substitution closed logic $\mathbb{L}$ with a type closed type system. We specify the main language constructs.

- $TUP(e_1, \ldots, e_n)$:
  - concrete syntax is $(e_1, \ldots, e_n)$;
  - typing rule: for arguments of types $\tau_1, \ldots, \tau_n$ respectively, the compound expression is of type $\tau_1 \times \cdots \times \tau_n$;

- Sem_{TUP} maps finite tuples $\bar{\mathfrak{i}}$ to the infon $\lambda I \in \mathbb{S}(\Sigma) : (\mathfrak{i}_1(I), \dots, \mathfrak{i}_n(I))$.

- $APP(e, e_1)$:
  - concrete syntax $e(e_1)$;
  - typing rule: for arguments of type $\tau_1 \to \tau, \tau_1$, the expression is of type τ;
  - Sem_{APP}: maps well-typed infons $\mathfrak{i}, \mathfrak{i}_1$ to $\lambda I \in \mathbb{S}(\Sigma) : \mathfrak{i}(I)(\mathfrak{i}_1(I))$.
- $Lambda(\bar{\sigma}, e)$: here $\bar{\sigma}$ is a finite sequence $\sigma_1, \dots, \sigma_n$ of symbols (not expressions);
  - concrete syntax $\lambda\sigma_1 \dots \sigma_n : e$; if e is Boolean, then $\{\sigma_1 \dots \sigma_n : e\}$;
  - typing rule: if the symbols $\sigma_1, \dots, \sigma_n$ are of types $\tau_1, \dots, \tau_n$ and the second argument is of type τ, the expression is of type $(\tau_1 \times \dots \times \tau_n) \to \tau$;
  - Sem_{Lambda} maps an $\Sigma \cup \{\sigma_1, \dots, \sigma_n\}$-infon $\mathfrak{i}$ of type τ to the Σ-infon $\lambda I \in \mathbb{S}(\Sigma) : F_I$, where F_I is the function $\lambda\bar{x} \in {\tau_1}^I \times \dots \times {\tau_n}^I : \mathfrak{i}(I[\bar{\sigma} : \bar{x}])$.

Equality, connectives and quantifiers are introduced using interpreted symbols, symbols with a fixed interpretation in each structure.

The logical symbols $\wedge, \vee : \mathsf{BOOL} \times \mathsf{BOOL} \to \mathsf{BOOL}$ and $\neg : \mathsf{BOOL} \to \mathsf{BOOL}$ have the standard Boolean functions as interpretations in every structure.

Quantifiers and equality are polymorphic. We introduce instantiations of them for all types τ. For every type τ, $\forall_\tau, \exists_\tau$ are symbols of type $(\tau \to \mathsf{BOOL}) \to \mathsf{BOOL}$. For concrete syntax, for $\forall_\tau(Lambda(\sigma, e))$ with e a Boolean expression and $\sigma : \tau$, we write $\forall\sigma : e$ (we dropped the underscore from $\forall_\tau$ since τ is the type of σ). It also corresponds to a quantified set comprehension $\forall_\tau(\{\sigma : \varphi\})$. In any structure I, ${\forall_\tau}^I$ is the Boolean function $\lambda X \in (\tau \to \mathsf{BOOL})^I : (X = \tau^I)$ that maps a set X with elements of type τ to $\mathbf{t}$ if X contains all elements of this type in I. Likewise, ${\exists_\tau}^I$ is the Boolean function $\lambda X \in (\tau \to \mathsf{BOOL})^I : (X = \emptyset)$.

Equality is a polymorphic interpreted predicate. For each τ, introduce a symbol $=_\tau$ of type $\tau \times \tau \to \mathsf{BOOL}$. The concrete syntax is $e = e_1$. Its interpretation in an arbitrary structure I is the identity relation of type τ^I.

Likewise, standard aggregate functions such as cardinality and sum are introduced as interpreted higher order Boolean functions. E.g., we introduce the interpreted symbol

$$Card_\tau : ((\tau \to \mathsf{BOOL}) \times NAT) \to \mathsf{BOOL}$$

interpreted in each structure I as the function

$$ {Card_\tau}^I : ((\tau \to \mathsf{BOOL})^I \times \mathbb{N}) \to \mathcal{T}wo : (S, n) \mapsto (\#(S) = n)).$$

We have chosen here to define $Card_\tau$ as a binary predicate symbol rather than as a unary function, because it is a partial function defined only on finite sets and our logic is not equipped for partial functions.

4.3 The definition construct DEF for higher order and nested definitions

So far, we have defined typed lambda calculus under an infon semantics. In this section, we extend the language with higher order versions of definitions as in the logic FO(ID). There, definitions are conventionally written as finite set of rules $\forall\bar{\sigma}(P(\bar{\sigma}) \leftarrow \varphi)$ where $P : (\bar{\tau} \to \mathsf{BOOL})$ is a predicate symbol, $\bar{\sigma} : \bar{\tau}$ a (sequence of) symbol(s), and φ a Boolean expression. E.g.,

Listing 1 The transitive closure of G.

```
{
∀x ∀y: Reach(x,y)← G(x,y).
∀x ∀z: Reach(x,z)← G(x,y)∧ Reach(y,z).
}
```

In the abstract syntax, a rule $\forall\bar{\sigma}(P(\bar{\sigma}) \leftarrow \varphi)$ will be represented as a pair $(P, \{\bar{\sigma} : \varphi\})$.

In general, an abstract expression of the definition construct DEF is of the form $DEF(\bar{P}, \bar{e})$ where $\bar{P}$ is a finite sequence $(P_1, \ldots, P_n)$ $(n > 0)$ of predicate symbols and $\bar{e}$ an equally long sequence of expressions. We write $(P, e) \in \Delta$ to denote that for some $i \leq n$, $P_i = P$ and $e_i = e$. Let $DP(\Delta)$ be $\{P_1, \ldots, P_n\}$, the set of *defined symbols* of Δ. It is possible that the same symbol P has multiple rules in Δ (as in the above example). Below, we use the mathematical variable Δ to denote definition expressions.

- For the concrete syntax, $DEF(\bar{P}, \bar{e})$ represents a definition with n rules corresponding to the pairs (P_i, e_i). If e_i is the set comprehension $\{\bar{\sigma} : \varphi\}$, the corresponding rule in concrete syntax is $\forall\bar{\sigma}(P(\bar{\sigma}) \leftarrow \varphi)$.
  Due to the substitution closedness of the logic, new abstract rules are allowed. E.g., $(Reach, G)$ is an abstract representation that is equivalent to the first rule in the *Reach* example, and it is an alternative way to represent the base case of the reachability relation.
- Typing rule: if for each $i \in [1, n]$, P_i, e_i are of the same type $\tau_i \to \mathsf{BOOL}$ then the definition expression is of type BOOL. It follows that the value of a definition in a structure is true or false. Note that defined symbols are predicate symbols.
- Sem_{DEF}: this operator maps tuples $((P_1, \ldots, P_n), (\mathfrak{i}_1, \ldots, \mathfrak{i}_n))$ where each $\mathfrak{i}_i$ is an infon of type τ_i to an infon $\mathfrak{i}$ of type BOOL. This operator will be applied to the infons $\mathfrak{i}_i$ of the expressions e_i. To define the infon $\mathfrak{i}$ from the input, we construct for each $I \in \mathbb{S}(\Sigma)$ the immediate consequence operator Γ^I_Δ.
  The operator Γ^I_Δ is an operator on $Assign^I_{DP(\Delta)}$, the lattice of $DP(\Delta)$-assignments in I. Note that for a rule $(P, e) \in D$, the value e^I of e in a structure I is exactly the set that this rule produces for P in I. The total produced value for P is then obtained by taking the union of all rules defining P. Formally, for each $P \in DP(\Delta)$, let $INF_P = \{\mathfrak{i}_i \mid P_i = P\}$. That is, INF_P is the set of infons amongst $\mathfrak{i}_1, \ldots, \mathfrak{i}_n$ that correspond to rules with P in the head. Then Γ^I_Δ maps an assignment $A \in Assign^I_{DP}$ to an assignment B such that for each $P \in DP$:

  $$P^B = lub_{\leq}(\{\mathfrak{i}_i(I[A]) \mid \mathfrak{i}_i \in INF_P\}$$

  That is, P^B is the union of what each rule of P produces in the structure $I[A]$.
  The operator Γ^I_Δ is well-defined, and indeed, it is the immediate consequence operator of Δ in structure I. This is a lattice operator on the lattice of assignments of the defined symbols $DP(\Delta)$ in I. Consequently, this operator will have an ultimate well-founded fixpoint UWF^I_Δ, the well-founded fixpoint of the ultimate approximator $\lceil \Gamma^I_\Delta \rceil$. This fixpoint may be exact or not. We define the truth value Δ^I of Δ in I as $(I = UWF^I_\Delta)$, that is, $\Delta^I = \mathbf{t}$ if I is the exact ultimate well-founded fixpoint of the operator, and $\Delta^I = \mathbf{f}$ otherwise. The infon $Sem_{\mathbb{L}}(\Delta)$ is the Boolean infon $\lambda I \in \mathbb{S}(\Sigma) : (I = UWF^I_\Delta)$.

The semantic operator $Sem_{\mathbb{L}}$ associates with each expression an infon, and with each theory T a Boolean infon $\mathfrak{i}$. This induces a model semantics, in particular $M \models T$ if $\mathfrak{i}(M) = \mathbf{t}$.

▶ **Theorem 19.** *The logic FO(ID) equipped with the ultimate well-founded semantics for definitions is a fragment of $\mathbb{L}$. That is, any theory T of FO(ID) corresponds syntactically to one T' of $\mathbb{L}$ and T and T' have the same models (taking the ultimate well-founded semantics for definitions).*

4.4 Applications for Higher Order Definitions

Higher order definitions are natural representations for some complex concepts. A standard example is a definition of winning positions in two-player games as can be seen in Listing 2. This definition of win and lose is a monotone second order definition that uses simultaneous definition and has a two-valued well-founded model.

Listing 2 cur is a winning position in a two-player game.

```
{
∀cur ∀Move ∀IsWon: win(cur, Move, IsWon) ← IsWon(cur) ∨
   ∃ nxt : Move(cur,nxt) ∧ lose(nxt,Move,IsWon).
∀cur ∀Move ∀IsWon: lose(cur,Move, IsWon) ← ¬IsWon(cur) ∧
   ∀ nxt : Move(cur,nxt) ⇒ win(nxt,Move,IsWon).
}
```

4.5 Templates

In [5], a subclass of higher order definitions were defined as templates. These templates allow us to define an abstract concept in an isolation, so that it can be reused multiple times. This prevents code duplication and results in more readable specifications. In the same context, we identified applications for nested definitions. An example of this can be seen in Listing 3. In that example a binary higher order predicate tc is defined, such that tc(P,Q) holds iff Q is defined as the transitive closure of P.

Listing 3 This template TC expresses that Q is the transitive closure of P.

```
{
∀Q ∀P: tc(P,Q) ←
   {∀x ∀y: Q(x,y) ← P(x,y) ∨(∃ z: Q(x,z)∧Q(z,y))}.
}
```

Note that using this definition of tc, the definition in Listing 1 can simply be replaced with the atom tc(Reach,G). This demonstrates the abstraction power of these definitions.

4.6 Graph Morphisms

A labeled graph is a tuple of a set of vertices, a set of edges between these vertices, and a labeling function on these vertices. Many applications work with labeled graphs: one example is the graph mining problem [12], which requires the notion of homomorphisms and isomorphisms between graphs. As other applications require these same concepts, these concepts lend themselves to a definition in isolation.

To achieve this, we first define the graph type as an alias for the higher order type $\mathcal{P}(node) \times \mathcal{P}(node \times node) \times \mathcal{P}(node \to label)$, where the components of the triple are called Vertex, Edge and Label respectively. To define when two graphs are homomorph and isomorph, we first define a helper predicate homomorphism. This predicate takes a function and two graphs, and is true when this function represents a homomorphism from the first graph to the second. We then define homomorph and isomorph in terms of the homomorphism predicate. In Listing 4, these higher order predicates are defined using higher order definitions. The higher order arguments of these definitions are either decomposed into the different tuple elements using matching (Line 2) or accepted as a single entity (Line 6).

Listing 4 Defining `homomorph` and `isomorph`.

```
{
homomorphism(F, (V1, Edge1, Label1), (V2, Edge2, Label2)) ←
     (∀ x, y [V1] : Edge1(x, y)  ⟹  Edge2(F(x), F(y))) ∧
     (∀ x : Label1(x) = Label2(F(x)).

homomorph(G1,G2) ←
    ∃ F [G1.Vertex:G2.Vertex] :  homomorphism(F, G1, G2).

isomorph(G1, G2) ←
    (∃ F [G1.Vertex:G2.Vertex], G [G2.Vertex:G1.Vertex] :
      (∀ x [G1.Vertex] : G(F(x)) = x) ∧
      homomorphism(F, G1, G2) ∧ homomorphism(G, G2, G1) ).
}
```

5 Conclusion

We defined a logic integrating typed higher order lambda calculus with definitions. The logic is type closed and substitution closed, allows definitions of higher order predicates and nested definitions. The logic satisfies a strong form of Frege's compositionality principle. The principles that we used allow also to define rules under other semantics (e.g., stable semantics). For future work, one question is how to define standard well-founded semantics for definitions in $\mathbb{L}$ rather than the ultimate well-founded semantics. It is well-known that both semantics often coincide, e.g., always when the standard well-founded model is two-valued, which is frequently the case when rule sets are intended to express definitions of concepts. Nevertheless, standard well-founded semantics is computationally cheaper and seems easier to implement. This provides a good motivation. Another question is how to extend definitions for arbitrary symbols, that is, for functions.

References

1. H. Abramson and H. Rogers. *Meta-programming in logic programming.* MIT Press, 1989.
2. Jon Barwise and John Etchemendy. Information, infons, and inference. *Situation theory and its applications*, 1(22), 1990.
3. Bart Bogaerts. *Groundedness in logics with a fixpoint semantics.* PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, June 2015. Denecker, Marc (supervisor), Vennekens, Joost and Van den Bussche, Jan (cosupervisors). URL: `https://lirias.kuleuven.be/handle/123456789/496543`.
4. Weidong Chen, Michael Kifer, and David S Warren. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming*, 15(3):187–230, 1993.
5. Ingmar Dasseville, Matthias van der Hallen, Gerda Janssens, and Marc Denecker. Semantics of templates in a compositional framework for building logics. *TPLP*, 15(4-5):681–695, 2015. `doi:10.1017/S1471068415000319`.
6. Marc Denecker, Victor Marek, and Mirosław Truszczyński. Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, volume 597 of *The Springer International Series in Engineering and Computer Science*, pages 127–144. Springer US, 2000. `doi:10.1007/978-1-4615-1567-8_6`.
7. Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log.*, 9(2):14:1–14:52, April 2008. `doi:10.1145/1342991.1342998`.

8 Keith Devlin. *Logic and information.* Cambridge University Press, 1991.

9 Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell.*, 175(1):278–298, 2011. doi:10.1016/j.artint.2010.04.002.

10 Paolo Ferraris. Answer sets for propositional theories. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 119–131, 2005. doi:10.1007/11546207_10.

11 Michael Gelfond and Yuanlin Zhang. Vicious circle principle and logic programs with aggregates. *TPLP*, 14(4-5):587–601, 2014. doi:10.1017/S1471068414000222.

12 Tias Guns. Declarative pattern mining using constraint programming. *Constraints*, 20(4):492–493, 2015.

13 Yuri Gurevich and Itay Neeman. Logic of infons: The propositional case. *ACM Trans. Comput. Log.*, 12(2):9, 2011. doi:10.1145/1877714.1877715.

14 Jerry R Hobbs and Stanley J Rosenschein. Making computational sense of montague's intensional logic. *Artificial Intelligence*, 9(3):287–306, 1977.

15 David B. Kemp and Peter J. Stuckey. Semantics of logic programs with aggregates. In Vijay A. Saraswat and Kazunori Ueda, editors, *ISLP*, pages 387–401. MIT Press, 1991.

16 Javier Leach, Susana Nieva, and Mario Rodríguez-Artalejo. Constraint logic programming with hereditary harrop formula. *CoRR*, cs.PL/0404053, 2004.

17 Vladimir Lifschitz. Answer set planning. In Danny De Schreye, editor, *Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 – December 4, 1999*, pages 23–37. MIT Press, 1999.

18 Victor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In Krzysztof R. Apt, Victor Marek, Mirosław Truszczyński, and David S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999. URL: http://arxiv.org/abs/cs.LO/9809032.

19 Gopalan Nadathur and Dale Miller. An overview of LambdaProlog. In *Fifth International Conference and Symposium on Logic Programming. MIT Press*, 1988.

20 Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273, 1999. doi:10.1023/A:1018930122475.

21 Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *TPLP*, 7(3):301–353, 2007. doi:10.1017/S1471068406002973.

22 Tran Cao Son, Enrico Pontelli, and Islam Elkabani. An unfolding-based semantics for logic programming with aggregates. *CoRR*, abs/cs/0605038, 2006. URL: http://arxiv.org/abs/cs/0605038.

23 Shahab Tasharrofi and Eugenia Ternovska. A semantic account for modularity in multi-language modelling of search problems. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings*, volume 6989 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2011. doi:10.1007/978-3-642-24364-6_18.

24 Shahab Tasharrofi and Eugenia Ternovska. Three semantics for modular systems. *CoRR*, abs/1405.1229, 2014. URL: http://arxiv.org/abs/1405.1229.

25 Allen Van Gelder. The well-founded semantics of aggregation. In *PODS*, pages 127–138. ACM Press, 1992. doi:10.1145/137097.137854.