A Modular Logical and Semantic Framework for Higher-Order Declarative Programming with Lambda Abstractions

Rafael del Vado Vírseda, Fernando Pérez Morente

TECHNICAL REPORT 04/12

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain

Abstract

Declarative programming paradigms allow to write programs with a very high level of abstraction that are close to specification languages, with the key difference that programs are executable. This makes these languages a very useful tool in the early stages of the development of complex software systems, being a natural choice for prototyping and quick testing of different high-level alternatives before proceeding to further stages of the design, implementation and distribution of a software product. One of the main characteristics of declarative languages is its higher-order nature. The addition of λ-abstractions improves dramatically their higher-order expressivity and allows to model easily systems that manipulate higher-order objects, such as advanced mathematical frameworks and logical circuits synthesis tools. In this work we propose a modular logical and semantic framework for higher-order declarative programming with λ-abstractions as an extension of pattern rewriting systems, based on λ-calculus to add advanced higher-order features in the form of λ-abstractions and higher-order pattern matching to standard multiparadigm declarative programming languages. We present a rewriting logic with an associated proof calculus that formally specifies derivability from the logic. Then we present declarative semantic concepts in the form of classic model-theoretic semantics and fixed-point semantics from this logic. Finally, we present the modular construction of higher-order programs, defining the properties of compositionality and full abstraction with respect to the most classical operations defined over program modules.

Keywords:
Declarative Programming, Higher-Order Patterns, Model-Theoretical Semantics, Fixed-Point Semantics, Modular Semantics, Rewriting Logics

1. Introduction

Declarative programming languages constitute a very interesting matter of research motivated by the fact that a higher programming level using powerful abstraction facilities leads to more reliable and maintainable software. Declarative programs allow to easily build models of an

application domain with high level of abstraction where non explicit properties can be inferred from a description of the domain. This makes these languages a very useful tool in the early stages of the development of complex software systems, being a natural choice for prototyping and quick testing of different high-level alternatives before proceeding to further stages of the design, implementation and distribution of the software product.

The most prominent declarative programming paradigms are *functional programming* [1, 2], where a program is a set of function definitions close to the notion of mathematical functions that allows to evaluate expressions by means of an equational logic, and *logic programming* [3, 4], where a program is a set of logical rules that allow to evaluate goals as inferences with respect to a logic. Even though both paradigms share the main features of the declarative paradigm (i.e., high level of abstraction, solid theoretical foundations in the form of formal denotational and operational semantics, etc.), there are several features that characterizes and differentiates each of them among the declarative paradigms. The main characteristics of the functional paradigm are: a computational model based on *rewriting* of expressions into equivalent ones with respect to a program [5], where the possibility of *lazy evaluation* (i.e., expressions are evaluated only when its value is demanded) opens the door to support the manipulation of infinite data structures; a *higher-order* nature of the functions, that are allowed to operate with functional elements as inputs or output; and, usually, the presence of a type system to denote the domain and range of the functions, together with the possibility of user-defined *data types* to model complex domains. The main features that characterize the logic programming paradigm are: a model of evaluation based on *resolution* and *unification* [6] (i.e., solving equations between terms with variables); the presence of logical variables in goals that can be instantiated to particular objects of the domain in the goal-solving process; and the possibility of *non-determinism*, that allows multiple valid solutions to a single goal and usually a mechanism to compute them all.

The *functional logic programming paradigm* [7, 8, 9, 10] represents a conservative combination of both the functional and the logic programming paradigms (being classical surveys [7, 8], and modern ones [9, 10]), in the sense that programs using the resources of only one of them are expected to have the same behavior in the combined declarative paradigm. In addition to the convenience of having the features of the most relevant declarative programming paradigms, functional logic programming languages have a greater expressivity (e.g., allowing a non deterministic manipulation of an infinite data structure), and can establish the basis of efficient goal-solving strategies [11, 12, 9]. The high level of expressivity of these languages makes necessary to have a sound and complete computational mechanism with respect to a suitable declarative semantics, based on rewriting, that traditionally lacks efficiency with respect to other lower level operational models. Even though, there are two main proposals for the operational semantics of functional logic programming languages that are at the basis of efficient programming languages and systems. The most commonly accepted proposal is *narrowing* [13, 14], a technique originally introduced in automatic theorem proving [15, 16, 17, 18] and adapted for declarative programming in [19], that is based in *rewriting with unification*, that is similar to rewriting in functional languages but substituting pattern matching (i.e., unification of a term with a *pattern*, that is a term that contains no variables) with unification to generate bindings for variables in the goal solving process. The other one is *residuation* [11], that is similar to resolution [6] in logic programming but suspending parts of the goal until enough information to perform a deterministic computation is generated. Considering these two main proposals, narrowing is usually preferred for the operational semantics of functional logic languages due to the lack of complete-

ness of resolution [11].

One of the main characteristics of functional logic languages, inherited from their functional component, is their *higher-order* nature. That means that it is possible to define abstract entities that can have other entities of the same kind as inputs or outputs; for instance, it is possible to have mathematical functions that operate with other mathematical functions, or transformations over program transformations; for example, we can have a transformation that suitable combines other program transformations in order to successively or alternatively apply them. This higher order nature, together with the high level of expressiveness of the declarative paradigm, makes functional logic programming languages a natural tool to model, and even implement, systems that manage objects of higher-order nature; this leads to programming languages that are useful in early stages of the software development cycle for prototyping and testing of high-level solutions to complicated problems. For example, the kernel of the advanced mathematical software system *Mathematica*^{TM} [20] is a particular application of higher-order rewriting techniques with numeric constraints to solve complex symbolic computations.

Most functional programming languages allow programming with λ-abstractions, with a syntax borrowed from the λ-*calculus* [21, 22], that stands for anonymous functions that are not explicitly declared and abstract a variable from a functional expression. These λ-abstractions are a straightforward way of having a syntactic representation of a huge space of functions without providing specific definitions and with a very compact and simple syntax. Functional languages operate by pattern matching and λ-abstractions are easily handled by the reduction model. On the other hand, there have been proposed several higher-order extensions to the logic programming paradigm. In [23] was supported the idea that logic languages are expressive enough to support several higher-order application domains without additional extensions, via a specially defined *apply symbol* and *flattening* (i.e., a specific predicate that represents the application of any predicate to any number of arguments, represented as a list), but at the same time recognizing some lacks of expressiveness of such an approach. Even though, it was also argued that the addition of λ-abstractions and higher-order unification (i.e., unification of terms with variables that can be instantiated to functions) to the paradigm would lead to a more expressive and adequate approach for many domains [24]. There were attempts to add λ-terms to logic languages to support higher-order features; unfortunately, it was known that higher-order unification with λ-*terms* is undecidable in the general case. In 1991 it was proposed a subset of λ-terms with decidable higher-order unification [25], the so called *higher-order patterns*. The λ-*prolog* language [26] incorporates these patterns, an approach that has been adapted to other paradigms as constraint programming [27], and that has lead to a very efficient implementation of higher-order logic programming [28]. Another extension to add functional notation to the logic language *Ciao-Prolog*, which is translated by a pre-processor to the logic language Prolog, has been proposed in [29].

In functional logic languages, the combination of unification with λ-abstractions is not straightforward, and has proved to be a complicated task. As far as we know, neither language, nor any implementation beyond a prototype of any functional logic existing system, supports λ-abstractions in its syntax and *higher-order unification* into its operational semantics while preserving solid declarative semantics foundations. The addition of those advanced higher-order features to functional logic languages would increase notably their expressivity; for example, a mathematical framework could infer functions that were not already defined to solve higher-order equations [30], a program transformation could be deduced from a specification to satisfy some

properties, or a logical circuit can be calculated from the specification of the logical function it should compute.

The aim of this work is to develop a modular logical and semantic framework for higher-order functional logic programming with λ-abstractions as an extension of the first-order functional logic programming framework presented in [31], that considers non-deterministic lazy functions with semantics based on a rewriting logic and an operational model based on demand-driven lazy narrowing [32]. As a first impression of our higher-order framework, the following example illustrates the kind of problems that a higher-order declarative programming framework with λ-abstractions makes easy to solve; it shows how the framework allows to implement a simple logical circuits design tool, where circuits representation and computation of logical functions are implemented straightaway with λ-abstractions. We also show how a language based on the framework can be used to compute logical circuits from an specification in a simple way.

Example 1.1. *We represent logical circuits as λ-abstractions and show how we can, from this representation, do actual logical circuit design following a specification of the behavior of the circuit. Circuits operate with signals that can take two values from a type 'val', represented as '0' and '1', and are represented as λ-abstractions with the input signals of the circuit being its abstracted variables. Generic logical functions (defined from their respective truth tables) are represented with a single function 'truth_table' with type '$[val] \to [val] \to val$', that has two lists of values as inputs and a value as output; the first list represents the ordered sequence of input values of the logical function and the second one represents the truth table that the function computes, as a list with the ordered sequence of values corresponding to the codification of the truth table of the function; the output value is the result of the logical function with the truth table of the second argument for the binary input configuration in the first one. The set of rewriting rules defining the behavior of generic logical functions with up to three inputs are partially defined as follows:*

$$
\begin{array}{lcl}
\mathit{truth\_table}([0],[x_0,x_1]) & \to & x_0 \\
\mathit{truth\_table}([1],[x_0,x_1]) & \to & x_1 \\
 & & \\
\mathit{truth\_table}([0,0],[x_0,x_1,x_2,x_3]) & \to & x_0 \\
\mathit{truth\_table}([0,1],[x_0,x_1,x_2,x_3]) & \to & x_1 \\
\mathit{truth\_table}([1,0],[x_0,x_1,x_2,x_3]) & \to & x_2 \\
\mathit{truth\_table}([1,1],[x_0,x_1,x_2,x_3]) & \to & x_3 \\
 & & \\
\mathit{truth\_table}([0,0,0],[x_0,x_1,x_2,x_3,x_4,x_5,x_6,x_7]) & \to & x_0 \\
\mathit{truth\_table}([0,0,1],[x_0,x_1,x_2,x_3,x_4,x_5,x_6,x_7]) & \to & x_1 \\
\vdots & \vdots & \vdots
\end{array}
$$

A circuit is represented as a λ-abstraction with its inputs represented as abstracted variables; for example, we can easily model typical logic gates such as two input 'and' and 'or' gates and a one input inverter (the classical 'not' gate) as follows:

$$
\begin{array}{lcl}
\mathit{and} & \rightarrow & \lambda u.v.\mathit{truth\_table}([u,v],[0,0,0,1]) \\
\mathit{or} & \rightarrow & \lambda u.v.\mathit{truth\_table}([u,v],[0,1,1,1]) \\
\mathit{not} & \rightarrow & \lambda u.\mathit{truth\_table}([u],[1,0])
\end{array}
$$

Now we are ready to do some actual design of logical circuits. For example, suppose that we want to design an unknown part in a logical circuit, so the whole circuit computes a certain logical function; in this particular example, we consider the circuit represented as '$\lambda u.v.w.F(\mathit{and}(u,v),\ \mathit{and}(\ u,w))$*', with 'F' representing an unknown circuit, to compute the logical function represented by the term '*$\lambda u.v.w.\mathit{truth\_table}([u,v,w],[1,1,1,1,1,1,\ 0,0])$*'; in order to do so, we can apply an adequate solving system [32] compatible with respect to the semantics described in our framework based on higher-order narrowing, getting, for example, the solution '*$\lambda u.v.\mathit{truth\_table}($ $[u,v],[1,0,0,0])$*' for 'F', that represents a 'nand' gate.*

Our logical and semantic framework provides formal tools to reason about this kind of higher-order programs and solutions. In this work we are going to describe the programming language that allows to represent complex domains of higher-order nature such as we do in this example, and we are going to provide a semantic characterization of it based on a rewriting logic, that is a valuable tool to reason about the programs and to support modern verification and debugging techniques [33]. For example, with the semantic characterization of our framework, we can build a formal proof of the fact that the computed solution '$\lambda u.v.\mathit{truth\_table}([u,\ v],[1,0,0,0])\}$*' for 'F' is effectively a solution of the problem. The construction of that formal proofs can be used as a "software certificate" and can be implemented in theorem provers like* Isabelle *[34] or* Coq *[35] in order to be automatically generated. Moreover, formal proofs in our logical framework can be used for the declarative debugging of programs; if the computational model obtains an incorrect answer, we can analyze its corresponding formal proof to identify erroneous formalizations of the rules in the program. For instance, if we obtain the obviously wrong solution '*$\lambda u.u$*' for 'F' for the previous circuit, we can identify the particular program rule that is producing this wrong answer.*

□

This work is an improved and extended version of our previous papers [32, 33, 36, 37], where many of the ideas to develop it were partially presented in their own context. The framework was initially introduced in [32] to present a suitable and well-founded operational semantics in the form of a higher-order demand-driven narrowing calculus (i.e., a formal calculus based on narrowing that performs only evaluation steps that are mandatory to reach a solution) with the help of *definitional trees*, a hierarchical structure of program rules that guides the selection and application of rewrite rules according to the syntactic structure of a function call. The rewriting logic and the main semantic concepts were originally presented in [33], and the essential concepts related to modular semantics were introduced in [37], but developed in the context of hybrid constraint domains, a different approach to the one taken in this work. In this work we compile and put together all these works, while presenting the foundational concepts in a simpler and more elegant way than they were originally published, and completing them with exhaustive mathematical proofs of the theoretical results.

The most relevant contributions of this work with respect to already published works [32, 33, 36, 37] can be summarized as follows:

- A complete higher-order conditional pattern rewriting framework based on λ-calculus.

This framework represents an extension to first-order term rewriting systems based on simply-typed λ-calculus for higher-order features. Most of the classic concepts on term rewriting systems are revisited, as well as basic notions in the context of declarative programming languages. These include a carefully revised notion of higher-order value that plays a role analogous to the one of *c-term* in first-order functional logic programming [31].

- A higher-order conditional rewriting logic suitable for our framework that encapsulates the behavior of the language in a few concise but powerful semantic rules. This leads to a proof calculus powerful enough to support correctness proofs about programs that are suitable to be implemented in a theorem prover, such as Isabelle/HOL [34] or the Coq proof assistant [35].
- A declarative semantics in the form of a model theoretic and a fixed-point semantics that extend analogous notions for first-order frameworks in a simple and a straightforward way, thanks to the definition of the programming language and the higher-order rewriting logic.
- A modular semantics specifically adapted for the framework based on λ-calculus that fulfills the properties of compositionality and full abstraction with respect to a simple module system defined on the framework. These two properties guarantee the applicability of our approach in application fields such as modular programming [38, 39], abstract interpretation [40], and to support sound program transformations [41].

The rest of this paper is organized as follows. In Section 2 we present the higher-order functional logic programming language with λ-abstractions that is proposed in this work. In Section 3 we present the higher-order rewriting logic that characterizes provability in our framework. Sections 4 and 5 define classical declarative semantics, while in Section 6 we define the modular framework along with suitable semantics for it. Finally, in Section 7 we compare our work to the most closely related frameworks and in Section 8 we summarize the basic conclusions and future work lines from this work. Complete and detailed proofs for the theoretical results of the work are provided; in the case of the most laborious and technical proofs we provide a sketch of them in the main text while routinary and technical details are relegated to an Appendix.

2. A Higher-Order Declarative Programming Language with λ-Abstractions

The higher-order functional logic programming language with λ-abstractions that is introduced in this section is an extension of the general term rewriting framework [42] on simply-typed λ-expressions [22, 43] that is closely related to the higher-order extension to logic programming firstly introduced by Miller [25, 44] and adapted in the context of functional logic programming in [30]. The goal of this section is to define the programs that we consider in our framework in Definition 2.5, for which are necessary some related notions of simply-typed λ-calculus and term rewriting systems; the most important of them are summarized in this section, together with other ones more specific of this framework that will be useful in the rest of this work.

The set of *λ-terms* is constructed from a signature Σ that contains function symbols, and an infinite countable set of variables $\mathcal{V}$ disjoint to Σ. Function symbols and variables are typed either with a *base type* (e.g., *nat*, *int*, *bool*, etc.) from a set of base types $\mathcal{B}$ or a functional type constructed with the left-associative functional types constructor '$\rightarrow$' (such as $nat \rightarrow nat$,

nat $\rightarrow$ (*nat* $\rightarrow$ *bool*), etc.). We will use the letter τ possibly subscripted or primed to denote generic types. The arity of a symbol f is its number of arguments and is represented as $arity(f)$. We also have a bottom constant for each base type $b \in \mathcal{B}$, denoted as $\perp_b$, that represents an undefined value that belongs to every base type. Terms are built by means of abstraction and composition operations applied to terms: $t ::= \perp \mid f \mid v \mid (t_1\ t_2) \mid (\lambda v.t_1)$, with $f \in \Sigma$ and $v \in \mathcal{V}$. We write $t :: \tau$ to denote that term t has type τ. *Total terms* are terms that do not have any occurrence of the bottom constant (e.g., *truth_table*$([0,0],[x_0,x_1,x_2,x_3])$) and we denote the set of all total terms built from a signature Σ and a set of variables $\mathcal{V}$ as $T(\Sigma,\mathcal{V})$; by contrary, *partial terms* are those that contain at least one occurrence of the bottom constant (e.g., $\perp$, $f(X,\perp)$, etc.), and the set that contains all partial and total terms is denoted as $T(\Sigma_\perp,\mathcal{V})$. The set of *bound variables* of a term t is denoted as $\mathcal{BV}(t)$ and contains each variable that appears under the scope of an abstraction of it in the term t; the set of *free variables* is denoted as $\mathcal{FV}(t)$ and contains each variable that occurs outside the scope of an abstraction of it. For example, in term $t \equiv \lambda u.v.truth\_table([u,v],[x_3,x_2,x_1,x_0])$, $\mathcal{BV}(t) = \{u,v\}$ and $\mathcal{FV}(t) = \{x_3,x_2,x_1,x_0\}$. A term is said to be *linear* if no free variable appears twice on it.

For brevity, we use the following notations and conventions related to terms: a sequence of syntactic objects $x_1, x_2, \ldots, x_n$, where $n \geq 0$, is denoted as $\overline{x_n}$. Successive applications of terms $(((a\,t_1)\,t_2)\cdots t_n)$ are denoted as $a(t_1,t_2,\ldots,t_n)$ or $a(\overline{t_n})$, where a is a variable or a symbol from Σ such that $arity(a) \geq n$. The successive abstraction of variables $\lambda x_1.\lambda x_2.\cdots.\lambda x_n.t$ is denoted as $\lambda x_1.x_2.\cdots.x_n.t$ or simply $\lambda\overline{x_n}.t$. We will use uppercase letters possibly subscripted such as F, G, X, Y for free variables and lower case ones for abstracted variables (such as u, v, w, x, ...). We will use other lowercase letters such as a,b, ... and lowercase identifiers for elements from Σ and $T(\Sigma_\perp,\mathcal{V})$. We will use π, π', π_1, π_2, ... for terms of base type. *Substitutions* are finite type-preserving mappings from variables to terms, that we denote by $\{\overline{x_n \mapsto t_n}\}$, with x_i either a free or bound variable, and extend homomorphically from terms to terms; we denote as $Subst(\Sigma_\perp,\mathcal{V})$ the set of substitutions that map variables to any term in $T(\Sigma_\perp,\mathcal{V})$; we use lowercase greek letters γ, σ, ... for substitutions. The restriction $\sigma\restriction_W$ of a substitution σ to a set W of variables is defined by $x\,\sigma\restriction_W = x\,\sigma$ if $x \in W$ and $x\,\sigma\restriction_W = x$ otherwise.

Now we summarize some basic concepts from the λ-calculus that are essential in this framework; we start with the three classical reduction rules in the context of simply-typed λ-calculus [22, 45]: α-*conversion*, β-*reduction* and η-*reduction*, whose reflexive and transitive closures are denoted respectively as $\rightarrow_\alpha$, $\rightarrow_\beta$ and $\rightarrow_\eta$, and whose one-step definitions are $\lambda u.t \rightarrow_\alpha \lambda v.t\{u \mapsto v\}$ if v is not a free variable in t, $(\lambda v.t)s \rightarrow_\beta t\{v/s\}$ and $\lambda v.tv \rightarrow_\eta t$ if $v \notin \mathcal{FV}(t)$. Since β-reduction and η-reduction are confluent (i.e., convergence of reduction sequences to the same term) and terminating (i.e., there are not infinite reduction sequences) in the context of simple types, any term t is guaranteed to have a *normal form* with respect to each of them that are denoted respectively $t\downarrow_\beta$ and $t\downarrow_\eta$. The union of β-reduction and η-reduction defines a confluent and terminating relation called $\beta\eta$-*reduction* whose reflexive and transitive closure we represent with $\rightarrow_{\beta\eta}$ and we denote the $\beta\eta$-normal form of any term t as $t\downarrow_{\beta\eta}$. For example, $\lambda u.v.truth\_table([u,v],[x_3,x_2,x_1,x_0]) \rightarrow_\alpha \lambda x.y.truth\_table([x,y],[x_3,x_2,x_1,x_0])$, $((\lambda u.v.truth\_table([u,v],[x_3,x_2,x_1,x_0]))0)1 \rightarrow_\beta truth\_table([u,v],[x_3,x_2,x_1,x_0])$ and $\lambda u.v.and(u,v) \rightarrow_\eta and$.

An important related concept for our framework, in order to properly define semantic key concepts, is the *long* $\beta\eta$-*normal form* of a term t, that is the η-*expanded* form of the β-normal form of t and is essential since those key semantic concepts will be defined only for terms in *long*

$\beta\eta$-normal form. The concept of long $\beta\eta$-normal form is defined as follows:

Definition 2.1 (Long $\beta\eta$-normal form).

- *Let $t \in T(\Sigma_\perp, \mathcal{V})$ be in β-normal form such that $t = \lambda\overline{x_n}.a(\overline{t_m})$ and $t :: \overline{\tau_{n+k}} \to \tau$. The η-expanded form of t is represented as $t\uparrow_\eta$ and is defined as $t\uparrow_\eta = \lambda\overline{x_{n+k}}.a(\overline{t_m}\uparrow_\eta, \overline{x_{n+k}}\uparrow_\eta)$ such that $\overline{x_{n+k}} \notin \mathcal{FV}(\overline{t_m})$.*
- *Let $t \in T(\Sigma_\perp, \mathcal{V})$. The long $\beta\eta$-normal form of t is denoted as $t\updownarrow_\beta^\eta$ and is defined as the η-expanded form of the β-normal form of t: $t\updownarrow_\beta^\eta = (t\downarrow_\beta)\uparrow_\eta$.*

For example, we have that $\lambda u.and(u)0\updownarrow_\beta^\eta = \lambda v.and(0, v)$; it is obvious to see that a term in long $\beta\eta$-normal form is not always in $\beta\eta$-normal form (e.g., $\lambda v.and(0, v)$ is in long $\beta\eta$-normal form but can be η-reduced to $and(0)$, so it is not a $\beta\eta$-normal form). It is well-known that $s \to_{\alpha\beta\eta} t$ if $s\updownarrow_\beta^\eta \to_\alpha t\updownarrow_\beta^\eta$ for any given terms s and t and, when considering simply-typed λ-terms, every term is guaranteed to have a $\beta\eta$-normal form, and consequently, a long $\beta\eta$-normal form [22]. The evaluation of this long $\beta\eta$-normal form is assumed to be an implicit operation in our framework (i.e., every term is expected to be normalized with respect to that relation at any point). We assume that terms are identified modulo α-conversion, and bound-variable clashes are avoided keeping free and bound variables disjoint using implicit α-conversion operations and assuming that bound variables with different binders have different names. For brevity, we may write variables and constants from Σ in η-normal form (i.e., X instead of $\lambda\overline{x_k}.X(\overline{x_k})$). With these conventions, every considered term t has a unique long $\beta\eta$-normal form with the syntactic form $\lambda\overline{x_k}.a(\overline{t_n})$, where $a \in \Sigma_\perp \cup \mathcal{V}$ and $a()$ coincides with a; we refer to symbol a as the *head* or *root* of the term t and denote it as $hd(t)$.

In our framework, *positions* in terms are represented as sequences of natural numbers useful to label and traverse all the subterms of a given term; we represent as ϵ the empty sequence; the sequence of n 1's is represented as 1^n, and the concatenation of sequences is represented by the symbol '$\cdot$'. We consider terms of the generic form $\lambda\overline{x_k}.a(\overline{t_n})$ and we use a sequence of 1's to traverse the sequence of abstracted variables $\lambda\overline{x_k}$; in positions that have k 1's, the next symbol represents the order of the subterm in the application that the position denotes. A prefix order relation among positions $\preceq$ is defined as "$p \preceq q$ if and only if there exists r such that $p \cdot r = q$". Formally:

Definition 2.2 (Positions and subterms). *Let $t \equiv \lambda\overline{x_k}.a(\overline{t_n}) \in T(\Sigma_\perp, \mathcal{V})$:*

- *The set of positions $Pos(t)$ in t is defined as $Pos(\lambda\overline{x_k}.a(\overline{t_n})) = \{1^i \mid 0 \le i \le k\} \cup \{1^k \cdot j \cdot q \mid 1 \le j \le n, q \in Pos(t_j)\}$.*
- *The subterm of t at position $p \in Pos(t)$ is denoted as $t|_p$ and is defined as:*
  - *$\lambda x_{i+1}.\cdots.x_k.a(\overline{t_n})$ if $p = 1^i$ with $0 \le i \le k$.*
  - *$t_i|_q$ if $p = 1^k \cdot i \cdot q$ with $1 \le i \le n$.*
- *A position p is maximal in t if $t|_p$ is of base type in $\mathcal{B}$. The set of maximal positions in a term t is denoted as $MPos(t)$.*

As an example of subterms and positions, the set of positions of $t \equiv \lambda u.v.w.F(and(u,v), and(u, w))$ is $Pos(t) = \{\epsilon, 1, 11, 111, 1111, 1112, 11111, 11112, 11121, 11122\}$ and some of their corresponding subterms are $\lambda u.v.w.F(and(u,v), and(u,w))$ corresponding to position ϵ, $F(and(u,v), and(u,w))$ to 111 and $and(u,w)$ to 1112. The set of maximal positions of t is $MPos(t) = \{111, 1111, 1112, 11111, 11112, 11121, 11122\}$.

The ordered sequence of variables abstracted on the path to a position p is denoted as $seq_{\mathcal{BV}}(t,p)$, and from that sequence we define the set $\mathcal{BV}(t,p)$ of all its elements; the set of bound variables $\mathcal{BV}(t)$ of a term t that was described before can be defined as the set that contains the variables abstracted into the path to any position, that is $\mathcal{BV}(t) = \bigcup_{p \in Pos(t)} \mathcal{BV}(t,p)$. Formally:

Definition 2.3 (Abstracted variables). *Let $x \in \mathcal{V}$, $s,t \in T(\Sigma_\perp, \mathcal{V})$, and $p,q \in Pos(t)$:*

- *The sequence of variables abstracted on the path to position p is denoted as $seq_{\mathcal{BV}}(t,p)$, and is defined as follows:*
  - $seq_{\mathcal{BV}}(t,\epsilon) = \epsilon$.
  - $seq_{\mathcal{BV}}(\lambda x.s, 1 \cdot q) = x \cdot seq_{\mathcal{BV}}(s,q)$.
  - $seq_{\mathcal{BV}}(h(\overline{t_n}), i \cdot q) = seq_{\mathcal{BV}}(t_i, q)$, *with* $1 \leq i \leq n$.
- *The set of variables abstracted on the path to position p is denoted as $\mathcal{BV}(t,p)$, and is defined as $\mathcal{BV}(t,p) = \{x \mid x$ is an element of $seq_{\mathcal{BV}}(t,p)\}$.*

The notation $t\restriction_p$ is used to denote the subterm of t at position p with every bound variable until that position abstracted, that is: $t\restriction_p = \lambda\overline{x_k}.(t|_p)$ where $\overline{x_k} = seq_{\mathcal{BV}}(t,p)$.

After these preliminaries, we can introduce programs as *Conditional Pattern Rewriting Systems* (shortly, *CPRS*), a higher-order extension of term rewriting systems [46, 47, 42] based on simply-typed λ-calculus. Since the problem of *higher-order unification* (i.e., solving equations in term structures in the presence of variables that can be instantiated to function symbols) for generic simply-typed λ-terms is undecidable (indeed, it is undecidable for the second-order case [48]), we restrict the kind of terms that appear in programs, considering a subset of simply-typed λ-terms where higher-order unification and *pattern matching* (i.e., unification of a generic term with a term containing no free variables) are both decidable. This subset is the one of the so called higher-order fully-extended *patterns* [25]. This restriction is quite standard in higher-order languages based on λ-abstractions with a logic programming component (see e.g., [25, 30, 28]).

Definition 2.4 (Higher-Order patterns).

- *A term $t \in T(\Sigma_\perp, \mathcal{V})$ is a higher-order pattern if and only if for all $p \in MPos(t)$ such that $hd(t|_p) \in \mathcal{FV}(t)$ and $t|_p = X(\overline{t_n})$ holds that $t_1{\downarrow_\eta} \in \mathcal{BV}(t,p), \ldots, t_n{\downarrow_\eta} \in \mathcal{BV}(t,p)$ is a sequence of distinct elements.*
- *A term $t \in T(\Sigma_\perp, \mathcal{V})$ is a higher-order fully-extended pattern if and only if it is a pattern, and for all $p \in MPos(t)$ such that $hd(t|_p) \in \mathcal{FV}(t)$ and $t|_p = X(\overline{t_n})$, $\mathcal{BV}(t,p) \setminus \{t_1{\downarrow_\eta}, \ldots, t_n{\downarrow_\eta}\} = \emptyset$.*

Examples of higher-order patterns are $\lambda x.y.\ F(x,y)$, $\lambda x.\ f(G(\lambda z.\ x(z)))$, and $\lambda x.y.z.\ g(F(x,y))$, where only the latter is not fully-extended. Non-patterns are for instance $\lambda x.y.\ F(a,y)$ and $\lambda x.\ G(H(x))$. It is well known that the matching of higher-order patterns is decidable and unitary [25]. Therefore, for every $t \in T(\Sigma_\perp, \mathcal{V})$ and pattern π, there exists at most one matcher between t and π, which we denote as '$matcher(t,\pi)$'.

In our higher-order programming framework, *programs* are conditional rewriting systems over fully-extended linear patterns, with conditional equations between total terms. In our context, an *equation* is a set $\{s,t\}$ written $s == t$ (or equivalently, $t == s$), where $s,t \in T(\Sigma_\perp, \mathcal{V})$ are of the same type.

Definition 2.5 (Conditional Pattern Rewriting System). *A conditional pattern rewriting system (or CPRS for brevity) is a finite set of conditional rewrite rules of the form $f(\overline{l_n}) \to r \Leftarrow C$ with $f :: \overline{\tau_n} \to \tau$, $\overline{l_n} :: \overline{\tau_n}$, and $r :: \tau$ with τ a base type, where*

- *$f(\overline{l_n})$ and r are total terms.*
- *$f(\overline{l_n})$ is a higher-order fully-extended linear pattern.*
- *C is a (possibly empty) finite sequence of equations between total terms.*

In a rule of the form $f(\overline{l_n}) \to r \Leftarrow C$, $f(\overline{l_n})$ is called the *left-hand side* (*lhs* for brevity) of the rule, r is called the *right-hand side* (*rhs* for brevity) and C is called the *conditional* part of the pattern rewrite rule. Each *CPRS* induces a partition of the signature Σ underlying the program rules into the set of *defined function symbols* Σ_d that have rules defining them (i.e., rules with the symbol at the head of the lhs), and the set of *data constructor symbols* Σ_c with no rule defining them.

Example 2.6. *One of the most interesting application fields of higher-order declarative programming is the design of symbolic calculus systems to deal with mathematical functions; for instance, the kernel of Mathematica™ [20] is a particular application of higher-order rewriting techniques with numeric constraints. In this example we present a higher-order program that evaluates the symbolic differentiation of a small set of functions defined by rules and some generic ones represented with λ-abstractions over natural numbers, which are represented in the Peano style with a constructor symbol 'zero' with arity 0 representing the natural number zero and a successor function 's' with arity 1 to build the rest of the natural numbers.*

$zero :: nat$
$s :: nat \to nat$

$add :: nat \to nat \to nat$
$add(x, zero) \to x$
$add(x, s(y)) \to s(add(x, y))$

$mult :: nat \to nat \to nat$
$mult(x, zero) \to zero$
$mult(x, s(y)) \to add(x, mult(x, y))$

$diff :: nat \to nat \to nat$
$diff(\lambda u.F, x) \to zero$
$diff(\lambda u.u, x) \to s(zero)$
$diff(\lambda u.add(F(u), G(u)), x) \to add(diff(\lambda u.F(u), x), diff(\lambda u.G(u), x))$
$diff(\lambda u.mult(F(u), G(u)), x) \to add(mult(diff(\lambda u.F(u), x), G(x)),$
$mult(diff(\lambda u.G(u), x), F(x)))$

As we can see in this example, higher-order patterns are allowed in the lhs of the rules, such as $diff(\lambda u.add(F(u), G(u)), x)$*, which is a desiderable feature expected in a higher-order framework that deals with higher-order objects.*

□

An auxiliary operation needed for defining a suitable declarative semantics for this class of higher-order programs is the notion of *lifter*, that deals with a problem that arises with bound variables when considering subterms of a given term; that is, considering that terms in the framework are in long $\beta\eta$-normal form, when considering a subterm of a term, some bindings of variables may be lost and a variable that was bound in the term will usually be free when considering a subterm. Also, when trying to match two terms it may be needed that both of them have the same names of variables as arguments of free variables so unification can be soundly performed. To solve these problems, we define the $\overline{x_k}$*-lifter* of a term t with respect to a set of variables V that adds to every free variable in t all the variables $\overline{x_k}$ given as arguments, and also adds the abstracted variables in the path to the position of the subterm.

Definition 2.7 (Lifter). *Let* $t \in T(\Sigma_\perp, \mathcal{V})$ *and* $V \subseteq \mathcal{V}$*:*

- *The* $\overline{x_k}$*-lifter of* t *with respect to* V *is a term denoted as* $t^{\uparrow \overline{x_k} \restriction_V}$ *that is:*
  - *If* $t \equiv \lambda \overline{y_l}.\pi$ *then* $(\lambda \overline{y_m}.\pi)^{\uparrow \overline{x_k} \restriction_V} = \lambda \overline{y_l}.(\pi^{\uparrow \overline{y_m}, \overline{x_k} \restriction_V})$ *for every* $\overline{y_m} \notin \mathcal{FV}(\pi)$ *in* y_l*.*
  - *If* $t \equiv a(\overline{t_n})$ *with* $a \notin V$ *then* $(a(\overline{t_n}))^{\uparrow \overline{x_k} \restriction_V} = a(\overline{t_n^{\uparrow \overline{x_k} \restriction_V}})$*.*
  - *If* $t \equiv X(\overline{t_n})$ *with* $X \in V$ *then* $(X(\overline{t_n}))^{\uparrow \overline{x_k} \restriction_V} = X(\overline{x_k}, \overline{t_n^{\uparrow \overline{x_k} \restriction_V}})$*.*
- *The* $\overline{x_k}$*-lifter of* t *is a term denoted as* $t^{\uparrow \overline{x_k}}$ *and is defined as* $t^{\uparrow \overline{x_k}} = t^{\uparrow \overline{x_k} \restriction_{\mathcal{FV}(t)}}$*. The notation* $t^{\downarrow \overline{x_k}}$ *denotes the term* $\lambda \overline{x_k}.(t^{\uparrow \overline{x_k}})$*.*
- *If* $C \equiv \overline{s_n == t_n}$ *is a sequence of equations, then* $C^{\updownarrow \overline{x_k}}$ *denotes the sequence* $\overline{s_n^{\updownarrow \overline{x_k}} == t_n^{\updownarrow \overline{x_k}}}$*.*

For example, $(\lambda x.Y(f(x, Z(x))))^{\uparrow y,z} = \lambda x.Y(y, z, f(x, Z(y, z, x)))$ whereas $(\lambda x.Y(x, Z(x)))^{\updownarrow y,z} = \lambda y.z.x.Y(y, z, f(x, Z(y, z, x)))$.

In order to deal with a suitable declarative semantics in the next sections, we are interested in terms that play a role similar to normal forms in term rewriting systems, that is, terms that cannot be further reduced with respect to the rules of a program. This is an extension of the concept of *c-term* usual in first-order programming frameworks (see e.g., [49]): in that first-order context, a c-term is a term that contains only data constructor and variable symbols; intuitively, those terms cannot be reduced since rules can only be applied to positions with a function symbol on it. In our higher-order framework, we allow higher-order patterns and we have to deal with higher-order variables, so we extend the concept of c-term to the concept of *value*; values do not have subterms that match with the lhs of any program rule of a given *CPRS*, so no rewrite steps can be performed from them. For example, terms $s(s(zero))$ and $\lambda u.s(s(u))$ are values with respect to the *CPRS* from Example 2.6, while $s(X)$, $\lambda x.y.add(s(x), s(y))$, and $\lambda x.y.F(s(x), s(y))$ are not values with respect to that *CPRS*.

Definition 2.8 (Values). *Let $t \in T(\Sigma_\perp, \mathcal{V})$ and $\mathcal{R}$ a CPRS:*

- *A term t is a value if and only if, for all $p \in MPos(t)$ and $(\pi \to r \Leftarrow C) \in \mathcal{R}$ such that $\mathcal{FV}(t) \cap \mathcal{FV}(\pi) = \emptyset$, there not exists 'matcher$(t|_p, \pi^{\updownarrow seq_{\mathcal{BV}}(t,p)})$'. The set of all values is denoted as $Val(\Sigma_\perp, \mathcal{V})$.*
- *A total term t that is a value is a total value. The set of all total values is denoted as $Val(\Sigma, \mathcal{V})$.*
- *A value substitution is a substitution $\{\overline{x_n \mapsto t_n}\}$, where $t_1, \ldots, t_n$ are values. The set of all value substitutions is denoted as $VSubst(\Sigma_\perp, \mathcal{V})$.*

With $[\mathcal{R}]_\perp$ we denote the set of all instances of a rule affected by a total value substitution, that is $[\mathcal{R}]_\perp = \{(l \to r \Leftarrow C)\,\theta \,|\, (l \to r \Leftarrow C) \in \mathcal{R} \text{ and } \theta \in VSubst(\Sigma_\perp, \mathcal{V})\}$.

3. The Higher-Order Rewriting Logic

In this section we provide a logical characterization of the semantics of *CPRS* programs by means of the *Goal-oriented Higher-order Rewriting Calculus* (*GHRC*), that allows to build logical proofs for *reduction* and *equality* statements and is a valuable tool to reason about programs for *debugging* and *verification* [33, 50]. Rewriting logics are considered an adequate and powerful tool to characterize the semantics of functional logic programs and they elegantly captures the characteristics of our higher-order programming language. The *GHRC* logic is an extension of the first-order *constructor-based conditional rewriting logic* (*CRWL* for short) schema described in [31] to deal now with higher-order conditional rewrite rules; it intends to model the evaluation of λ-terms in a constructor-based language involving non-strict lazy functions. As in [31], we do not impose non-ambiguity conditions, so non-deterministic functions are supported.

With the proof calculus underlying the *GHRC* logic we can build derivations for two kind of statements:

B	**Bottom**	$\lambda\overline{x_k}.\,\pi \twoheadrightarrow \lambda\overline{x_k}.\,\bot$
RF	**ReFlexivity**	$s \twoheadrightarrow s$
MN	**MoNotonicity**	$\dfrac{\lambda\overline{x_k}.\,s_1 \twoheadrightarrow \lambda\overline{x_k}.\,t_1 \;\cdots\; \lambda\overline{x_k}.\,s_n \twoheadrightarrow \lambda\overline{x_k}.\,t_n}{\lambda\overline{x_k}.\,a(\overline{s_n}) \twoheadrightarrow \lambda\overline{x_k}.\,a(\overline{t_n})}$ for all $a \in \Sigma \cup \mathcal{V}$.
OR	**Outermost Reduction**	$\dfrac{\lambda\overline{x_k}.\,s_1 \twoheadrightarrow l_1^{\updownarrow\overline{x_k}} \;\cdots\; \lambda\overline{x_k}.\,s_n \twoheadrightarrow l_n^{\updownarrow\overline{x_k}} \;\; C^{\updownarrow\overline{x_k}} \;\; r^{\updownarrow\overline{x_k}} \twoheadrightarrow u}{\lambda\overline{x_k}.\,f(\overline{s_n}) \twoheadrightarrow u}$ if $(f(\overline{l_n}) \to r \Leftarrow C) \in [\mathcal{R}]_\bot$, for any $u \neq \lambda\overline{x_k}.\bot$.
J	**Join**	$\dfrac{s \twoheadrightarrow u \qquad t \twoheadrightarrow u}{s == t}$ if $u \in \mathit{Val}(\Sigma, \mathcal{V})$.

Figure 1: The *GHRC* proof calculus.

- *Reduction statements* $s \twoheadrightarrow t$, where $s, t \in T(\Sigma_\bot, \mathcal{V})$ and s and t have the same type, that represent sequences of reductions of terms to terms in the underlying rewriting framework by applying several rewriting steps.
- *Equality statements* $s == t$, where $s, t \in T(\Sigma, \mathcal{V})$ and s and t have the same type, that represent equality as reduction to a common total value, that is $s \twoheadrightarrow u$ and $t \twoheadrightarrow u$ with $u \in \mathit{Val}(\Sigma, \mathcal{V})$. Notice that "equality" is only defined for total terms.

Our rewriting framework is based on the semantic notion of *approximation* of terms. We can see terms as partially computed information of their respective types; then total terms are completely evaluated, while the $\bot$ constant of each base type is the term containing less information of that type; among those maximum and minimum elements, there are all the approximations of the total term that can be obtained by substituting subterms by a $\bot$ constant. The *approximation ordering* among partial terms $\sqsubseteq$, that relates terms with respect to the relation "$s \sqsubseteq t$ if and only if s is an approximation of a more or equally defined term t", is defined as the least partial ordering such that:

$$\lambda\overline{x_k}.\bot \sqsubseteq \lambda\overline{x_k}.t \qquad t \sqsubseteq t \qquad \frac{s_1 \sqsubseteq t_1 \;\cdots\; s_n \sqsubseteq t_n}{\lambda\overline{x_k}.\,a(\overline{s_n}) \sqsubseteq \lambda\overline{x_k}.\,a(\overline{t_n})} \text{ with } a \in \Sigma \cup \mathcal{V}$$

The *GHRC* proof calculus is defined in Figure 1 by means of four recursive inference rules, being two of them base rules and the other two recursive rules; this calculus follows the idea of approximation, so terms can be rewritten to other terms following the approximation ordering,

and also performing rewrite steps applying program rules of the *CPRS* to a suitable subterm. There is only one rule defining equality statements $s == t$, that specifies equality conditions as joinability of s and t to a common total value u. This follows the idea of specifying joinability as a generalization of *strict equality*, where total values in our higher-order framework play a role analogous to total c-terms in the first-order framework [31]. This is easily expressed by the following rule for equality statements:

$$\frac{s \twoheadrightarrow u \quad t \twoheadrightarrow u}{s == t} \text{ if } u \in \mathit{Val}(\Sigma, \mathcal{V}).$$

Now we present the rules for reduction statements. These rules encapsulate the desired specification of the language, that is, non-strict lazy non-deterministic functions with *call-time choice* [31]. With call-time choice semantics, variables replicated after applying a rewrite rule have to be reduced to the same term (e.g., if we apply a rule with the form '$f(X) \to g(X, X)$', then both arguments of g must be reduced to the same term).

The first base rule of the recursive calculus allows to reduce any pattern to a corresponding bottom constant $\bot$ of the same type, reducing it to the minimum comparable element with respect to the approximation order $\sqsubseteq$ between terms, that is a $\bot$ constant. This rule represents non-terminating or unnecessary computations solving a goal in the operational model [32]; this means that a non-strict argument of a function that is not going to be demanded when applying a rewrite rule can be reduced to a $\bot$ constant; at this level of the logic, it is manifested by the fact that any pattern can be reduced to a corresponding constant $\bot$ in one step of the calculus. For example, in the case of applying a function that computes the first element (head) of a list, the computation of the other elements of the list is not required, which allows to avoid a non-termination situation in the case of an infinite list with a lazy operational model; at the level of the logic, the computation of the other elements of the list is reduced to a $\bot$ constant when building a proof tree. Because of that, the *bottom rule* **B** is very simple and corresponds to the first rule of the approximation ordering amongst terms $\lambda\overline{x_k}.\, \bot \sqsubseteq \lambda\overline{x_k}.t$ and is defined as follows:

$$\lambda\overline{x_k}.\, \pi \twoheadrightarrow \lambda\overline{x_k}.\, \bot$$

The bottom rule is one of the two base rules of the proof calculus; the other one is a reflexivity rule, corresponding to the second rule of the approximation ordering amongst terms $t \sqsubseteq t$. This rule allows to stop a proof at its current state; this will be useful, for example, proving an equality statement and reaching a term that is a total value. The *reflexivity rule* **RF** that expresses this behavior is defined simply as:

$$s \twoheadrightarrow s$$

The first inductive rule of the calculus is the *monotonicity rule* **MN**, corresponding to the monotonicity property of the approximation ordering. It starts a proof derivation by a reduction of the arguments of a function or variable symbol:

$$\frac{\lambda\overline{x_k}.\, s_1 \twoheadrightarrow \lambda\overline{x_k}.\, t_1 \;\ldots\; \lambda\overline{x_k}.\, s_n \twoheadrightarrow \lambda\overline{x_k}.\, t_n}{\lambda\overline{x_k}.\, a(\overline{s_n}) \twoheadrightarrow \lambda\overline{x_k}.\, a(\overline{t_n})} \text{ for all } a \in \Sigma \cup \mathcal{V}.$$

The other recursive rule of the calculus defines how terms are reduced applying program rules of a *CPRS* $\mathcal{R}$ to them. The rule is called *outermost reduction* **OR** because the program rule is applied to the outermost function symbol of the term. To do that, the symbol f at the head

of a program rule must be the same that the symbol at the head of the term $t = \lambda\overline{x_k}.f(\overline{s_n})$, the parameters of the term $\lambda\overline{x_k}.\overline{s_n}$ must be reduced to the formal parameters of the program rule that is applied, and the (possible) conditions of the rule must be fulfilled; if that is the case, the rule can be applied and the term can be rewritten to the rhs r of the rule instantiated by the substitution inferred in the process. Formally, the rule that reflects this behavior is the following:

OR

$$\frac{\lambda\overline{x_k}.\, s_1 \twoheadrightarrow l_1^{\updownarrow\overline{x_k}} \;\cdots\; \lambda\overline{x_k}.\, s_n \twoheadrightarrow l_n^{\updownarrow\overline{x_k}} \quad C^{\updownarrow\overline{x_k}} \quad r^{\updownarrow\overline{x_k}} \twoheadrightarrow u}{\lambda\overline{x_k}.\, f(\overline{s_n}) \twoheadrightarrow u}$$

$$\text{if } (f(\overline{l_n}) \to r \Leftarrow C) \in [\mathcal{R}]_\perp, \text{for any } u \neq \lambda\overline{x_k}.\perp.$$

It is interesting to explain the use of the lifter $\updownarrow x_k$ in this rule. Program rules are generic entities that have to deal with many different terms, and any of them can have any number of different abstracted variables; the lifter ensures that the parameters in the program rule have the same bound variables that the term that is being reduced, opening the possibility of applying the rule with independence of the names of the variables abstracted until the subterm at the position where the rule is applied. It is also interesting to notice that an instance $(f(\overline{l_n}) \to r \Leftarrow C) \in [\mathcal{R}]_\perp$ of the program rule is applied, which ensures the call-time choice parameter passing since it is instanced by a common substitution. Finally, requiring $u \neq \lambda\overline{x_k}.\perp$ is useful to avoid overlapping with the bottom rule **B**, that is preferred since generates simpler derivations.

Sometimes it is useful to split the **OR** rule to make explicit in the calculus the argument reduction and the result of the function that is applied. This has applications in verification and debugging (see e.g., [33, 51, 52, 53]). If that is the case, the rule **OR** is replaced by a more expressive version:

OR

$$\frac{\lambda\overline{x_k}.s_1 \twoheadrightarrow l_1^{\updownarrow\overline{x_k}} \;\ldots\; \lambda\overline{x_k}.s_n \twoheadrightarrow l_n^{\updownarrow\overline{x_k}} \qquad \dfrac{C^{\updownarrow\overline{x_k}} \qquad r^{\updownarrow\overline{x_k}} \twoheadrightarrow u}{\boxed{\lambda\overline{x_k}.f(\overline{l_n^{\updownarrow\overline{x_k}}}) \twoheadrightarrow u}}\ \mathbf{RA}}{\lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow u}\ \mathbf{AR}$$

$$\text{if } (f(\overline{l_n}) \to r \Leftarrow C) \in [\mathcal{R}]_\perp, \text{for any } u \neq \lambda\overline{x_k}.\perp.$$

Now the **OR** rule is split in the successive application of a rule to perform the parameter passing, named *argument reduction* and denoted **AR**, and another one for the application of the rule that includes checking the equations on its conditional part (if there is any), named *rule application* and denoted **RA**. The function call enclosed in a box is called a *basic fact*; Example 3.3 illustrates the use of the proof calculus for debugging and verification purposes and gives additional details about this subject. Individually, the rules resulting from the split of the **OR** rule are the following, that have to be successively applied under the same premises of rule **OR**:

AR

$$\frac{\lambda\overline{x_k}.\, s_1 \twoheadrightarrow l_1^{\updownarrow\overline{x_k}} \;\cdots\; \lambda\overline{x_k}.\, s_n \twoheadrightarrow l_n^{\updownarrow\overline{x_k}} \qquad \boxed{\lambda\overline{x_k}.\, f(\overline{l_n^{\updownarrow\overline{x_k}}}) \twoheadrightarrow u}}{\lambda\overline{x_k}.\, f(\overline{s_n}) \twoheadrightarrow u}$$

RA

$$\frac{C^{\updownarrow\overline{x_k}} \qquad r^{\updownarrow\overline{x_k}} \twoheadrightarrow u}{\boxed{\lambda\overline{x_k}.\, f(\overline{l_n}) \twoheadrightarrow u}}$$

From the proof calculus of Figure 1, it is possible to build *proof trees* for reduction and equality statements, as is illustrated in Example 3.1. We denote the *set of proof trees* for a reduction or equality statement φ as $\mathcal{PT}(\varphi)$, and the set of proof trees ending with the application of a rule **R** of the calculus as $\mathcal{PT}_{\mathbf{R}}(\varphi)$. We say that statement φ is provable from the calculus and a *CPRS* $\mathcal{R}$ if there exists a proof tree for it and we denote it as $\mathcal{R} \vdash \varphi$; if there exists a proof tree ending with rule **R** we denote it as $\mathcal{R} \vdash_{\mathbf{R}} \varphi$. We denote the negation of these relations respectively as $\mathcal{R} \nvdash \varphi$ and $\mathcal{R} \nvdash_{\mathbf{R}} \varphi$.

Example 3.1. *In this example we use the proof calculus underlying the GHRC-logic to generate a proof tree of a reduction statement with respect to the CPRS defined in Example 2.6. This example shows that the construction of a proof tree can be laborious and non-trivial even for a simple statement, because it has some redundant parts, and there are many choice points where we have to guess the rule to apply in order to find a solution; the latter problem would be difficult to solve for an automatic method to build proof trees.*

Now we prove that 'diff(zero) ∗ s(s(zero))' is equal to 's(s(zero))' (i.e., the differential of the identity function at the point zero multiplied by two is equal to two) by building a proof tree for the reduction statement 'mult(diff(λx.x, zero), s(s(zero))) ↠ s(s(zero))':

OR $mult(diff(\lambda x.x, zero), s(s(zero))) \twoheadrightarrow s(s(zero))$
- **OR** $diff(\lambda x.x, zero) \twoheadrightarrow s(zero)$
  - **RF** $\lambda x.x \twoheadrightarrow \lambda x.x$
  - **RF** $zero \twoheadrightarrow zero$
  - **RF** $s(zero) \twoheadrightarrow s(zero)$
- **RF** $s(s(zero)) \twoheadrightarrow s(s(zero))$
- **OR** $add(diff(\lambda x.x, zero), mult(diff(\lambda x.x, zero), s(zero))) \twoheadrightarrow s(s(zero))$
  - **OR** $diff(\lambda x.x, zero) \twoheadrightarrow s(zero)$
    - **RF** $\lambda x.x \twoheadrightarrow \lambda x.x$
    - **RF** $zero \twoheadrightarrow zero$
    - **RF** $s(zero) \twoheadrightarrow s(zero)$
  - **OR** $mult(diff(\lambda x.x, zero), s(zero)) \twoheadrightarrow s(zero)$
    - **OR** $diff(\lambda x.x, zero) \twoheadrightarrow s(zero)$
      - **RF** $\lambda x.x \twoheadrightarrow \lambda x.x$
      - **RF** $zero \twoheadrightarrow zero$
      - **RF** $s(zero) \twoheadrightarrow s(zero)$
    - **RF** $s(zero) \twoheadrightarrow s(zero)$
    - **OR** $add(diff(\lambda x.x, zero), mult(diff(\lambda x.x, zero), zero)) \twoheadrightarrow s(zero)$
      - **OR** $diff(\lambda x.x, zero) \twoheadrightarrow s(zero)$

RF $\lambda x.x \twoheadrightarrow \lambda x.x$
RF $zero \twoheadrightarrow zero$
RF $s(zero) \twoheadrightarrow s(zero)$
OR $mult(diff(\lambda x.x, zero), zero) \twoheadrightarrow zero$
OR $diff(\lambda x.x, zero) \twoheadrightarrow s(zero)$
RF $\lambda x.x \twoheadrightarrow \lambda x.x$
RF $zero \twoheadrightarrow zero$
RF $s(zero) \twoheadrightarrow s(zero)$
RF $zero \twoheadrightarrow zero$
RF $zero \twoheadrightarrow zero$
RF $s(zero) \twoheadrightarrow s(zero)$
MN $s(add(diff(\lambda x.x, zero), zero)) \twoheadrightarrow s(s(zero))$
OR $add(diff(\lambda x.x, zero), zero) \twoheadrightarrow s(zero)$
OR $diff(\lambda x.x, zero) \twoheadrightarrow s(zero)$
RF $\lambda x.x \twoheadrightarrow \lambda x.x$
RF $zero \twoheadrightarrow zero$
RF $s(zero) \twoheadrightarrow s(zero)$
RF $zero \twoheadrightarrow zero$
OR $diff(\lambda x.x, zero) \twoheadrightarrow s(zero)$
RF $\lambda x.x \twoheadrightarrow \lambda x.x$
RF $zero \twoheadrightarrow zero$
RF $s(zero) \twoheadrightarrow s(zero)$

□

In order to complete the presentation of our higher-order framework in a declarative programming setting, we need a formal definition for the *goals* from a given *CPRS* $\mathcal{R}$ that we want to solve, and the kind of *solutions* for the goals that we want to obtain by using an appropriate operational semantics based on higher-order narrowing (see e.g., [54, 32]). Goals are defined as sets of equality statements, that must be simultaneously fulfilled in order to solve the goal in the precise way that the logic defines, by reduction to common total values. Solutions are common substitutions to each equality statement in the goal that, when applied, allow to reduce each of them to a common total value by means of the rules of the *GHRC*-proof calculus.

Definition 3.2 (Goals and solutions).

- *Let* $\overline{s_n}, \overline{t_n} \in T(\Sigma, \mathcal{V})$ *such that for all* $i \in \{1, \ldots, n\}$ *both* s_i *and* t_i *has the same type. A goal* G *for a given CPRS* $\mathcal{R}$ *is a set* $\{\overline{s_n == t_n}\}$ *of equations. Equations are view as symmetric:* $s == t \equiv t == s$.
- *A substitution* $\gamma \in Subst(\Sigma_\perp, \mathcal{V})$ *is a solution of a goal* $G \equiv \{\overline{s_n == t_n}\}$ *if* $\gamma \restriction_{\mathcal{FV}(G)} \in VSubst(\Sigma_\perp, \mathcal{V})$*, and for each equation* $s_i == t_i$ *in* G *there exists a proof tree* $\mathcal{P}_i \in \mathcal{PT}(s_i\gamma == t_i\gamma)$*. The proof tree* $\mathcal{P}_i$ *is called a witness that* γ *is a solution of* $s_i == t_i$.
- *We write* $Soln(G)$ *for the set of solutions of a goal* G*, and* $Wtn_\gamma(G)$ *for the set of witnesses that* γ *is a solution of* G.

With the calculus defined in Figure 1, it is possible to build formal proofs for reduction and equality statements that follow a logical structure based on a rewriting logic, that can be at the basis of sound formal methods for verification and debugging of programs (see Example 3.3

below). For instance, it is possible to build a proof tree that verifies that a solution is a correct one of a goal (see Example 3.1) or a debugging tree to find a portion of the code of the program that is producing an undesired behavior by means of an incorrect or incomplete computed answer discovered from an error symptom.

Example 3.3. *In this example, we show how the GHRC logic helps to verify non-trivial higher-order properties about CPRS-programs and to deal with the declarative debugging of incorrect programs. We want to verify the following property regarding the standard function over lists 'foldr', that evaluates the successive application of a binary function to the elements of a list:* $(+1) \circ foldr\,(+)\,0 = foldr\,(+)\,1$. *We use the following definition for the 'foldr' function by means of rewrite rules:*

$$\begin{array}{lcl} foldr\,(\lambda u.v.\,G(u,v), E, [\,]) & \rightarrow & E \\ foldr\,(\lambda u.v.\,G(u,v), E, [X|Xs]) & \rightarrow & G(X, foldr\,(\lambda u, v.\,G(u,v), E, Xs)) \end{array}$$

In order to prove the property, we use the so-called fusion law *[55]:*

$$\lambda x, y.\, f\,(g\,x\,y) = \lambda x.y.\, h\,x\,(f\,y) \Rightarrow f \circ foldr\,g\,z = foldr\,h\,(f\,z)$$

So we need to find functions f, g, h and z such that the property may be satisfied. In order to do so, we try with the following function definitions for f, g and z that we suspect suitable for solving the problem, and try to infer h by applying a correct goal solving system with respect to GHRC [32]:

$$\begin{array}{lcl} f & \rightarrow & \lambda z.z \\ g & \rightarrow & \lambda u.v.u + v \\ z & \rightarrow & 0 \end{array}$$

In order to infer h, we can solve the goal '$\lambda x.y.f(gxy) == \lambda x.y.Hx(fy)$' expecting to get the value of the function h as a value assigned to the higher-order variable 'H'. After applying the goal solving system [32], we get a solution '$\gamma = \{H \mapsto \lambda u.v.u + v\}$', that we check by applying the fusion law. However, we obtain the trivial identity $foldr\,(+)\,0 = foldr\,(+)\,0$, that is not the expected result, so an error has occurred in the program rules defining functions f, g and z. In order to find the error, we can start a debugging session examining a simplified proof tree of the statement that only contains the function calls performed in the evaluation of the goal. After removing all the nodes not corresponding to a function call, the debugging tree is the one in Figure 2.

In order to find the erroneous function, we have to look for a so-called buggy node*, that is a node containing an erroneous statement whose children contain correct statements with respect to the expected behavior of the program, so we can ensure that the error is produced in the node and is not being propagated from its children nodes. In Figure 2, we can see that the node with the double-line cover contains an erroneous statement, and it is a buggy node because it has no children (so we can ensure that the error is not being propagated from a deeper level). So that, we can conclude that function f is erroneously defined.*

Therefore, the user has to reformulate the definition of f, which is done defining it with the rule '$f \rightarrow \lambda z.z + 1$' and, after repeating the process, we can apply the fusion law to get

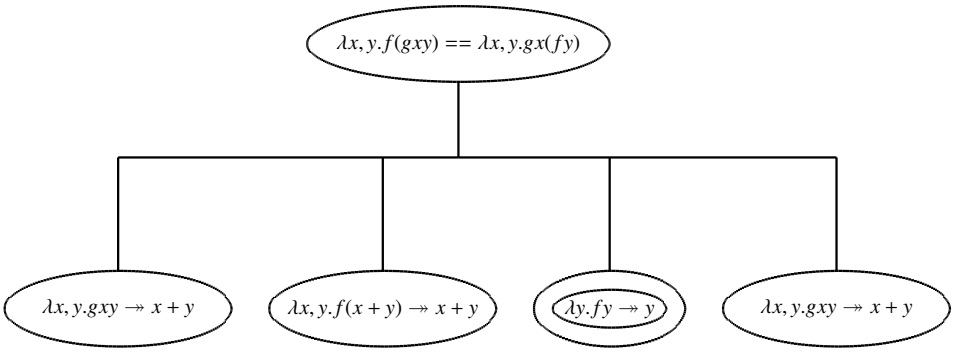

Figure 2: Debugging tree corresponding to the goal $\lambda x, y.f(gxy) == \lambda x, y.Hx(fy)$

$(+1) \circ$ *foldr* $(+)$ 0 $=$ *foldr* $(+)$ 1*, that is what we expected, so we have formally proved the property.*

□

In the next section we are going to provide a denotational semantics for the framework, defining the mathematical object that denotes the semantics of each *CPRS*-program. That denotational semantics is closely related to the rewriting logic presented in this section.

4. Model-Theoretic Semantics

Model-theoretic semantics are widely adopted to support denotational semantics for declarative languages [56, 31]. In this section, we define *models* for higher-order programs formalized as *CPRS*s in Section 2 and we establish soundness and completeness results of provability for the *GHRC*-logic presented in Section 3 with respect to validity in such models. Moreover, we prove that every *CPRS* $\mathcal{R}$ has a canonic *pattern model* $\mathcal{M}_{\mathcal{R}}$, which can be seen as a generalization of the canonic $\mathcal{C}$*-semantics* for logic programming as defined in [56, 57].

To be able to define model-theoretic semantics, we need some technical preliminary notions about *Domain Theory* [58] which are at the basis of denotational semantics. The primary motivation for the study of domains, which was initiated by *Dana Scott* in the late 1960s, was the search for a denotational semantics of the λ-calculus [59]. We only summarize here the most relevant notions to understand this work and we refer the interested reader to [59, 58], that extend these concepts.

First we give the definition of a *partially ordered set with bottom* and some auxiliary notions about them, which are at the first basis of the semantics of our framework since terms follow that structure; this is essential to establish the equivalence between *GHRC*-derivations and semantic validity in Theorem 4.8 at the end of this section.

Definition 4.1 (Partially ordered sets with bottom).

- *A partially ordered set (also called poset for short) with bottom (generically denoted as $\bot$) is a set S with a partial order $\sqsubseteq$ defined between elements of S such that $\bot \sqsubseteq x$ for all $x \in S$.*
- *Let S be a poset with bottom and $D \subseteq S$. Then D is a directed set if and only if for all $x, y \in D$ there exists $z \in D$ such that $x \sqsubseteq z$ and $y \sqsubseteq z$.*
- *Let S be a poset with bottom and $x \in S$. The element x is totally defined in S if it is a maximal element of S (i.e., there not exists $y \in S$ such that $y \neq x$ and $y \sqsubseteq x$).*

An example of poset with bottom is the set of partial values $Val(\Sigma_\bot, \mathcal{V})$ defined in Definition 2.8 with the approximation ordering $\sqsubseteq$ defined in Section 3 and the bottom element $\bot$ corresponding to each base type, where maximal elements are total values.

The adequate structure to denote defined function and constructor symbols in the signature of a program are *cones* and *ideals*. Cones are partially ordered sets with bottom that contain all the approximations with respect to $\sqsubseteq$ of elements that belong to them. *Ideals* are cones that satisfy the additional property that for each couple of elements there exists a least upper bound of both in the ideal. Formally:

Definition 4.2 (Cones and ideals). *Let S be a partially ordered set with bottom $\bot$:*

- *The set $A \subseteq S$ is a cone if and only if $\bot \in A$ and for all $x \in A, y \in S$ if $y \sqsubseteq x$ then $y \in A$ (the latter condition will be referred as 'A is downclosed').*
- *The set of cones of S is denoted as $\mathcal{C}(S)$ and is defined as $\mathcal{C}(S) = \{A \subseteq S \mid A \text{ is a cone}\}$.*
- *A set $I \subseteq S$ is an ideal if is a cone and is a directed set.*
- *The set of ideals of S is denoted as $\mathcal{I}(S)$ and is defined as $\mathcal{C}(S) = \{A \subseteq S \mid A \text{ is an ideal}\}$.*
- *The ideal completion of S is denoted as $\bar{S}$ and is defined as a poset $\mathcal{I}(S)$ with the 'set inclusion partial ordering' $\subseteq$ and bottom element $\emptyset$ (the empty set).*
- *Let S be a poset with partial order $\sqsubseteq$ and $x \in S$. The ideal generated by x is denoted as $\langle\, x \,\rangle$ and is defined as $\langle\, x \,\rangle = \{y \in S \mid y \sqsubseteq x\}$.*

For example, a subset of the set of partial values consisting on a value and all its possible approximations is a cone; also, the cone consisting on a total value and all its possible approximations is the ideal generated by the value. Following [60], we interpret a *CPRS*-program over structures of posets with bottom as carriers, whose elements are thought of finite approximations of possibly infinite values in the poset's ideal completion. Moreover, monotonic mappings from elements to cones represent defined function symbol denotations reflecting possible nondeterminism, and ideals in the case of constructor symbols, that are always deterministic.

Definition 4.3 (*GHRC*-algebras). *Given a signature $\Sigma = \Sigma_c \cup \Sigma_d$, a GHRC-algebra $\mathcal{A}$ is a structure of the form*

$$\mathcal{A} = \langle\, D_{\mathcal{A}}, \{c^{\mathcal{A}}\}_{c \in \Sigma_c}, \{f^{\mathcal{A}}\}_{f \in \Sigma_d}, \circ^{\mathcal{A}} \,\rangle$$

where:

- *The carrier set $D_{\mathcal{A}}$ is a poset with partial order $\sqsubseteq_{D_{\mathcal{A}}}$ and bottom element $\bot_{\mathcal{A}}$.*
- *For each $c \in \Sigma_c$ with arity$(c) = n$, $c^{\mathcal{A}}$ is a monotonic mapping $D_{\mathcal{A}}^n \to \mathcal{I}(D_{\mathcal{A}})$.*
- *For each $f \in \Sigma_d$ with arity$(f) = n$, $f^{\mathcal{A}}$ is a monotonic mapping $D_{\mathcal{A}}^n \to \mathcal{C}(D_{\mathcal{A}})$ that verifies for all total $\gamma \in Subst(\Sigma_\bot, \mathcal{V})$:*

  $$(f^{\mathcal{A}}(\overline{t_n}))\gamma = \{d\gamma \,|\, d \in f^{\mathcal{A}}(\overline{t_n})\} \subseteq f^{\mathcal{A}}(\overline{t_n}\gamma)$$

  This condition is the so-called consistency condition*, close to the notion of c-interpretation considered in [57].*
- *The apply operation $\circ^{\mathcal{A}}$ is a non-deterministic binary mapping $D_{\mathcal{A}} \times D_{\mathcal{A}} \to \mathcal{C}(D_{\mathcal{A}})$, written in infix notation, such that $\bot_{\mathcal{A}} \circ^{\mathcal{A}} d = \langle \bot_{\mathcal{A}} \rangle$ for all $d \in D_{\mathcal{A}}$.*

Now we show how to *evaluate* λ-terms in *GHRC*-algebras; evaluation of terms is dependant on an actual instantiation of variables, that we define by mean of *valuations*, that are mappings from variables to elements in the carrier set of a *GHRC*-algebra.

Definition 4.4 (Valuations). *Let $\mathcal{A}$ be a GHRC-algebra and $\mathcal{V}$ a set of variables:*

- *A valuation over $\mathcal{A}$ is any mapping $\eta : \mathcal{V} \to D_{\mathcal{A}}$.*
- *A valuation η is totally defined if and only if for all $v \in \mathcal{V}$, $\eta(v)$ is a totally defined element of $D_{\mathcal{A}}$.*

Given a valuation η we can evaluate each λ-term in $\mathcal{A}$ as follows:

Definition 4.5 (Evaluation of terms). *Let $\mathcal{A}$ be a GHRC-algebra over a signature $\Sigma = \Sigma_c \cup \Sigma_d$, $t \in T(\Sigma_\bot, \mathcal{V})$, and η a valuation of $\mathcal{V}$. The evaluation of t in $\mathcal{A}$ under η is denoted as $[\![t]\!]_\eta^{\mathcal{A}}$ and is defined inductively as follows:*

- $[\![\bot]\!]_\eta^{\mathcal{A}} = \langle \bot_{\mathcal{A}} \rangle$.
- $[\![X]\!]_\eta^{\mathcal{A}} = \langle \eta(X) \rangle$, *if* $X \in \mathcal{V}$.
- $[\![c]\!]_\eta^{\mathcal{A}} = \langle c^{\mathcal{A}} \rangle$, *if* $c \in \Sigma_c$.
- $[\![f]\!]_\eta^{\mathcal{A}} = f^{\mathcal{A}}$, *if* $f \in \Sigma_d$ *and* arity$(f) = 0$.
- $[\![f]\!]_\eta^{\mathcal{A}} = \langle f^{\mathcal{A}} \rangle$, *if* $f \in \Sigma_d$ *and* arity$(f) > 0$.
- $[\![(t\ t')]\!]_\eta^{\mathcal{A}} = [\![t]\!]_\eta^{\mathcal{A}} \circ^{\mathcal{A}} [\![t']\!]_\eta^{\mathcal{A}}$.
- $[\![\lambda\overline{x_k}.\, a(\overline{t_n})]\!]_\eta^{\mathcal{A}} = [\![\, (\cdots (a\ t_1) \cdots\ t_n) \,]\!]_\eta^{\mathcal{A}}$, *if* $\{\overline{x_k \mapsto d_k}\} \in \eta$ *with* $d_i \in D_{\mathcal{A}}$.

Using *GHRC*-algebras, we are able to extend the model-theoretic results from [31] to our higher-order setting with λ-abstractions. We interpret reduction statements as inclusions between cones, and equality statements as assertions of the existence of at least one common total value. In particular, models in *GHRC* are introduced using the following notions of satisfiability:

Definition 4.6 (Models). *Let $\mathcal{R}$ be a CPRS and $\mathcal{A}$ a GHRC-algebra:*

- *The algebra $\mathcal{A}$ satisfies a reduction statement $s \twoheadrightarrow t$ under a valuation η (denoted as $\mathcal{A} \vDash_\eta (s \twoheadrightarrow t)$), if and only if $[\![s]\!]^{\mathcal{A}}_\eta \supseteq [\![t]\!]^{\mathcal{A}}_\eta$.*
- *The algebra $\mathcal{A}$ satisfies an equality statement $s == t$ under a valuation η (denoted as $\mathcal{A} \vDash_\eta (s == t)$), if and only if $[\![s]\!]^{\mathcal{A}}_\eta \cap [\![t]\!]^{\mathcal{A}}_\eta$ contains a maximal element in $D_{\mathcal{A}}$. The satisfaction of a sequence of equality statements C is denoted as $\mathcal{A} \vDash_\eta C$ and is defined as the satisfaction of $\mathcal{A} \vDash_\eta (s == t)$ for every $s == t$ in C.*
- *The algebra $\mathcal{A}$ satisfies a conditional pattern rewrite rule $(\pi \to r \Leftarrow C) \in \mathcal{R}$ (denoted as $\mathcal{A} \vDash (\pi \to r \Leftarrow C)$), if and only if $\mathcal{A} \vDash_\eta C$ implies $\mathcal{A} \vDash_\eta (\pi \twoheadrightarrow r)$, for every valuation η.*
- *The algebra $\mathcal{A}$ is a model of $\mathcal{R}$ (denoted as $\mathcal{A} \vDash \mathcal{R}$) if and only if $\mathcal{A}$ satisfies all the conditional pattern rewrite rules in $\mathcal{R}$.*

GHRC-provability is *sound* and *complete* with respect to this model-theoretic semantics when we consider totally defined valuations. For each *CPRS* $\mathcal{R}$, we can build a so-called *pattern model* $\mathcal{M}_{\mathcal{R}}$ as a *GHRC*-algebra $\mathcal{A}$ with carrier set $D_{\mathcal{A}}$ the set of values, that is the denotation we are going to choose for programs among all of the algebras that are models of them. The interpretation of function symbols is defined by means of all the possible *GHRC*-derivations and establishes a close relation between the pattern model and the logical calculus.

Definition 4.7 (Pattern model). *Let $\mathcal{R}$ be a CPRS with signature Σ. The pattern model of $\mathcal{R}$ is denoted as $\mathcal{M}_{\mathcal{R}}$ and is defined as follows:*

- *The carrier set $D_{\mathcal{M}_{\mathcal{R}}}$ is the poset $Val(\Sigma_\perp, \mathcal{V})$ of partial values over Σ, with approximation ordering $\sqsubseteq$ (as defined in Section 3) and bottom element $\perp$ for each base type $b \in \mathcal{B}$.*
- *For each $c \in \Sigma_c$ with $arity(c) = n$ and for all $t_i \in Val(\Sigma_\perp, \mathcal{V})$, $c^{\mathcal{M}_{\mathcal{R}}}(\overline{t_n}) = \langle\, c(\overline{t_n}) \,\rangle$.*
- *For each $f \in \Sigma_d$ with $arity(f) = n$ and for all $t_i \in Val(\Sigma_\perp, \mathcal{V})$, $f^{\mathcal{M}_{\mathcal{R}}}(\overline{t_n}) = \{\, t \in Val(\Sigma_\perp, \mathcal{V}) \mid \mathcal{R} \vdash f(\overline{t_n}) \twoheadrightarrow t \,\}$.*
- *The apply operation $t_1 \circ^{\mathcal{M}_{\mathcal{R}}} t_2 = \langle\, (t_1\ t_2) \,\rangle$, whenever $(t_1\ t_2) \in Val(\Sigma_\perp, \mathcal{V})$.*

Finally, we show the adequateness of this notion of pattern model as the semantic object denoting the interpretation of each program, proving that it is actually a model of the *CPRS* $\mathcal{R}$, and relating deducibility by means of the *GHRC* proof calculus and satisfiability from the pattern model of $\mathcal{R}$.

Theorem 4.8 (Adequateness of $\mathcal{M}_{\mathcal{R}}$). *Let $\mathcal{R}$ be a CPRS. The pattern model $\mathcal{M}_{\mathcal{R}}$ is a model of $\mathcal{R}$. Moreover, for any reduction or equality statement φ, the following conditions are equivalent:*

1) $\mathcal{R} \vdash \varphi$.
2) $\mathcal{A} \vDash_\eta \varphi$, *for every* $\mathcal{A} \vDash \mathcal{R}$ *and every totally defined valuation* η.

3) $\mathcal{M}_\mathcal{R} \models_\varepsilon \varphi$, where ε is the identity valuation.

Proof First we prove that $\mathcal{M}_\mathcal{R}$ is a model of $\mathcal{R}$ by proving that $\mathcal{M}_\mathcal{R}$ satisfies all the pattern rewrite rules in $\mathcal{R}$ following Definition 4.6, by applying the characterization of the pattern model given in Definition 4.7 and the definition of the logic from Figure 1. Then we prove the equivalence of 1), 2) and 3) by proving that 1) $\Rightarrow$ 2) $\Rightarrow$ 3) $\Rightarrow$ 1) as follows:

1) $\Rightarrow$ 2) We assume that $\mathcal{R} \vdash \varphi$ for any reduction or equality statement φ and consider $\mathcal{A} \models \mathcal{R}$ an arbitrarily fixed totally defined valuation η. Then we can prove that $\mathcal{A} \models_\eta \varphi$ by induction on the length of the *GHRC*-proof considering the characterization of the pattern model given in Definition 4.7.

2) $\Rightarrow$ 3) Is trivial because $\mathcal{M}_\mathcal{R}$ is a *GHRC*-model of $\mathcal{R}$ and ε is a totally defined valuation.

3) $\Rightarrow$ 1) We prove that, for any approximation or equality statement φ, if $\mathcal{M}_\mathcal{R} \models_\varepsilon \varphi$ then $\mathcal{R} \vdash \varphi$ proceeding by induction on the size of φ, being the size of a valuation the number of symbols occurring on it.

A complete and detailed proof of this result can be found in Appendix.

□

The equivalence between items 1) and 2) shows that the *GHRC*-proof calculus is sound and complete for deriving statements which hold in all models of a given *CPRS* under all possible totally defined valuations:

$$\mathcal{R} \vdash \varphi \Leftrightarrow \mathcal{A} \models_\eta \varphi, \text{ for every } \mathcal{A} \models \mathcal{R} \text{ and every totally defined valuation } \eta.$$

The pattern model defined in this section, that is closely related to the *GHRC*-logic described in Section 3, is not only the mathematical object that denotes the semantics of a given *CPRS*-program, it is a powerful tool whose fixed-point characterization given in Section 5 will allow to *modularize* the framework in order to build modular proofs and to support a modular language on top of the framework, as is described in Section 6.

5. Fixed-Point Semantics

The denotational semantic characterization of a program is usually given in terms of the least fixed-point of an associated continuous transformation. In this section we prove for every *CPRS* $\mathcal{R}$ that its pattern model $\mathcal{M}_\mathcal{R}$ is the least fixed-point of a continuous operator defined over *pattern algebras*, which are *GHRC*-algebras with the set of partial values from $\mathcal{R}$ as carrier set, ordered by the approximation ordering $\sqsubseteq$ defined in Section 2, and a fixed interpretation for data constructors. The pattern model $\mathcal{M}_\mathcal{R}$ of a *CPRS* $\mathcal{R}$ is a particular case of pattern algebra.

The approach we use here is similar to that applied in the field of *logic programming* [3]. However, the notion of interpretation, and the corresponding mathematical aspects, have to be reformulated in the new context of pattern algebras. This approach has been also used in the context of previous formalisms to model *higher-order functional logic programming* (see e.g., [54]). However, [54] does not deal with some relevant aspects (e.g., non-determinism or λ-abstractions) of the *GHRC*-programming approach we are considering here.

Definition 5.1 (Pattern algebra). *Let $\mathcal{A}$ be a GHRC-algebra and $\mathcal{R}$ a CPRS with signature $\Sigma = \Sigma_c \cup \Sigma_d$. The algebra $\mathcal{A}$ is a pattern algebra if and only if:*

- *The carrier set $D_{\mathcal{A}}$ is the set of partial values $Val(\Sigma_\perp, \mathcal{V})$ with partial order $\sqsubseteq_{D_{\mathcal{A}}}$ the approximation ordering $\sqsubseteq$ over terms (as defined in Section 3), and bottom element the bottom constant $\perp$ corresponding to each base type.*
- *For each $c \in \Sigma_c$ with $arity(c) = n$ and for all $t_i \in Val(\Sigma_\perp, \mathcal{V})$, $c^{\mathcal{M}_{\mathcal{R}}}(\overline{t_n}) = \langle\, c(\overline{t_n})\,\rangle$.*
- *The apply operation $t_1 \circ^{\mathcal{M}_{\mathcal{R}}} t_2 = \langle\, (t_1\ t_2)\,\rangle$, whenever $(t_1\ t_2) \in Val(\Sigma_\perp, \mathcal{V})$.*

The set of pattern algebras of Σ associated to $\mathcal{R}$ is denoted as Alg_Σ.

It is interesting to notice that the set of pattern algebras Alg_Σ of signature $\Sigma = \Sigma_c \cup \Sigma_d$ associated to a *CPRS* $\mathcal{R}$ is a poset with bottom when we define an ordering based on interpretation of defined function symbols $f \in \Sigma_d$ as a set inclusion of cones. Also, following this definition there exists a family of pattern algebras that satisfies a fixed *CPRS*-program; in this section we construct a pattern model, that is a particular case of pattern algebra, by applying subsequently an operator over pattern algebras starting with a fixed one that contains minimum information with respect to the behavior of the program; as pattern algebras are defined as a particular case of *GHRC*-algebras, all of those generated in this process will follow the conditions imposed to the interpretation of function symbols in Definition 4.3 (i.e., monotonicity and consistency).

Now we are going to describe the formal construction that allows to build the pattern model by the subsequent application of an operator defined over pattern algebras. To do so, and to be able to guarantee the existence of the pattern model as the least fixed-point of the operator, we have to define a *complete lattice* on pattern algebras and a continuous operator on it. Then, the existence of its least fixed-point is guaranteed, that is characterized by the subsequent application of the operator to the pattern algebras starting from the bottom element of the lattice; we prove that this least fixed-point corresponds to the pattern model $\mathcal{M}_{\mathcal{R}}$ described in Section 4. In order to do so, we start defining the complete lattice of pattern algebras.

Definition 5.2 (Pattern algebra relation). *Let $\mathcal{A}$ and $\mathcal{B}$ be two pattern algebras of Alg_Σ. We define the relation $\sqsubseteq$ as the least relation such that:*

- *$f^{\mathcal{A}}(\overline{t_n}) \sqsubseteq f^{\mathcal{B}}(\overline{t_n})$ for all $f \in \Sigma_d$, when $arity(f) > 0$.*
- *$f^{\mathcal{A}} \sqsubseteq f^{\mathcal{B}}$, when $arity(f) = 0$.*

It is trivial to see that this relation is a partial ordering, and from it we can define the poset Alg_Σ of pattern algebras over a fixed signature Σ with the partial ordering $\sqsubseteq$. This poset has a bottom element and a top element, denoted respectively as $\perp_\Sigma$ and $\top_\Sigma$, that are characterized by the interpretation of function symbols in the algebra as $f^{\perp_\Sigma}(\overline{t_n}) = \langle \perp \rangle$ and $f^{\top_\Sigma}(\overline{t_n}) = Val(\Sigma_\perp, \mathcal{V})$, respectively, for each $f \in \Sigma_d$ with $arity(f) \geq 0$. Now we define the interpretation of the function symbols for a subset of pattern algebras.

Definition 5.3 (Pattern algebra interpretation). *Let $S \subseteq Alg_\Sigma$. Then, for each $f \in \Sigma_d$:*

- *$f^{\sqcup S}(\overline{t_n}) =_{def} \bigcup_{\mathcal{A} \in S} f^{\mathcal{A}}(\overline{t_n})$*

- $f^{\sqcap S}(\overline{t_n}) =_{def} \bigcap_{\mathcal{A}\in S} f^{\mathcal{A}}(\overline{t_n})$

This definition characterizes two pattern algebras, denoted respectively by $\sqcup S$ and $\sqcap S$, since the union and intersection of any number of cones are also cones, and the resulting functions in the above definition are obviously monotonic if $f^{\mathcal{A}}$ is monotonic for all $\mathcal{A} \in S$. Clearly, $\sqcup S$ and $\sqcap S$ are the *least upper bound* and the *greatest lower bound* of S, respectively. So that, the set of Alg_Σ of *GHRC*-pattern algebras defines a *complete lattice*.

The following auxiliary lemma establishes the continuity of valuations in Alg_Σ, that states that pattern algebras with a less defined interpretation of defined function symbols will evaluate terms to equally or less defined cones.

Lemma 5.4 (Continuity of valuations in Alg_Σ). *For each partial term $t \in T(\Sigma_\perp, \mathcal{V})$ and each value substitution $\theta \in VSubst(\Sigma_\perp, \mathcal{V})$:*

1. *If $\mathcal{A} \sqsubseteq \mathcal{B}$ then $[\![t]\!]^{\mathcal{A}}_\theta \subseteq [\![t]\!]^{\mathcal{B}}_\theta$, for $\mathcal{A}, \mathcal{B} \in Alg_\Sigma$.*
2. *$[\![t]\!]^{\sqcup D}_\theta = \bigcup_{\mathcal{A}\in D} [\![t]\!]^{\mathcal{A}}_\theta$, for all directed subsets $D \subseteq Alg_\Sigma$.*

Proof The proof of 1. proceeds by induction on the structure of t, proving each of the cases following the semantics of terms provided in Definition 4.5. To prove 2. we show that we only need to prove the inclusion that states that $[\![t]\!]^{\sqcup D}_\theta \subseteq \bigcup_{\mathcal{A}\in D} [\![t]\!]^{\mathcal{A}}_\theta$ by induction on t, because the inclusion in the other way is a trivial consequence of 1. A complete and detailed proof of this result can be found in Appendix.

□

Given a *CPRS* $\mathcal{R}$ with signature Σ, we can define the *pattern algebra transformer* $\mathbb{T}_\mathcal{R}$, that is a continuous transformation on Alg_Σ similar to the immediate consequences operator used in logic programming [56, 57], by fixing the interpretation of each function symbol $f \in \Sigma_d$ in the transformed pattern algebra $\mathbb{T}_\mathcal{R}(\mathcal{A})$, as the result of one step application of those pattern rewrite rules of $\mathcal{R}$ defining f, satisfied in $\mathcal{A} \in Alg_\Sigma$. Formally:

Definition 5.5 (Pattern algebra transformer). *Let $S \subseteq Alg_\Sigma$. Then, the pattern algebra transformer of Alg_Σ is denoted as $\mathbb{T}_\mathcal{R}$ and for each $f \in \Sigma_d$:*

$$f^{\mathbb{T}_\mathcal{R}(\mathcal{A})}(\overline{t_n}) =_{def} \{\perp\} \cup \{\, t \mid t \in [\![f(\overline{t_n})]\!]^{\mathcal{A}}_\varepsilon \,\} \cup \{\, t \mid \exists (f(\overline{l_n}) \to r \Leftarrow C) \in [\mathcal{R}]_\perp, \overline{l_n} \sqsubseteq \overline{t_n}, \mathcal{A} \vDash_\varepsilon C, t \in [\![r]\!]^{\mathcal{A}}_\varepsilon \,\}$$

This is basically a union of cones. This definition corresponds to a monotonic mapping because all rule instances $(f(\overline{l_n}) \to r \Leftarrow C) \in [\mathcal{R}]_\perp$, applicable to arguments $\overline{t'_n}$, are also applicable to arguments $\overline{t_n}$ such that $t'_i \sqsubseteq t_i$, for $i = 1, \ldots, n$, and so the corresponding interpretation characterizes a pattern algebra. From this definition and the following auxiliary results, we can derive the continuity of the operator $\mathbb{T}_\mathcal{R}$ in Alg_Σ .

Lemma 5.6 (Upper bound). *Let C be a finite sequence of equality statements and D a directed subset of Alg_Σ. Then, $\sqcup D \vDash_\eta C$ implies that there exists $\mathcal{A} \in D$ such that $\mathcal{A} \vDash_\eta C$ for every valuation η.*

Proof We prove the lemma for the case when C consists only of one equality statement $s == t$, because with more statements we shall obtain pattern algebras $\mathcal{A}_1, \ldots, \mathcal{A}_n$, one for each equality statement, and the upper bound $\sqcup\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ will satisfy all equality statements in C. By

definition, $\sqcup D \vDash_\eta (s == t)$ implies that there exists a totally defined $r \in [\![s]\!]_\eta^{\sqcup D} \cap [\![t]\!]_\eta^{\sqcup D}$, and by the second item of Lemma 5.4, if $r \in [\![s]\!]_\eta^{\sqcup D}$ then $r \in [\![s]\!]_\eta^{\mathcal{A}_1}$ for some $\mathcal{A}_1 \in D$, and if $r \in [\![t]\!]_\eta^{\sqcup D}$ then $r \in [\![t]\!]_\eta^{\mathcal{A}_2}$ for some $\mathcal{A}_2 \in D$. By the first item of Lemma 5.4, considering $\mathcal{A} \in D$ such that $\mathcal{A}_i \sqsubseteq \mathcal{A}$, $i = 1, 2$, we have a pattern algebra such that $r \in [\![s]\!]_\eta^{\mathcal{A}} \cap [\![t]\!]_\eta^{\mathcal{A}}$, and consequently $\mathcal{A} \vDash_\eta (s == t)$.

□

Lemma 5.7 (Continuity of $\mathbb{T}_\mathcal{R}$). *For each CPRS $\mathcal{R}$, its associated operator $\mathbb{T}_\mathcal{R}$ is continuous.*

Proof We prove first that $\mathbb{T}_\mathcal{R}$ is monotonic. Given $\mathcal{A}, \mathcal{B} \in Alg_\Sigma$ such that $\mathcal{A} \sqsubseteq \mathcal{B}$, if $\mathcal{A} \vDash_\varepsilon C$ then $\mathcal{B} \vDash_\varepsilon C$ for every sequence C of equality statements, and by the first item of Lemma 5.4, $[\![t]\!]_\varepsilon^{\mathcal{A}} \subseteq [\![t]\!]_\varepsilon^{\mathcal{B}}$ for every term t; hence, every rule instance $(f(\overline{l_n}) \to r \Leftarrow C) \in [\mathcal{R}]_\perp$ applicable to obtain $f^{\mathbb{T}_\mathcal{R}(\mathcal{A})}(\overline{t_n})$ also will be applicable to obtain $f^{\mathbb{T}_\mathcal{R}(\mathcal{B})}(\overline{t_n})$, and therefore $\mathbb{T}_\mathcal{R}(\mathcal{A}) \sqsubseteq \mathbb{T}_\mathcal{R}(\mathcal{B})$. Then we can prove that $\mathbb{T}_\mathcal{R}$ is continuous considering every directed set $D \subseteq Alg_\Sigma$, $\mathbb{T}_\mathcal{R}(\sqcup D) \sqsubseteq \sqcup\{\mathbb{T}_\mathcal{R}(\mathcal{A}) \mid \mathcal{A} \in D\}$ because each rule instance $(f(\overline{l_n}) \to r \Leftarrow C) \in [\mathcal{R}]_\perp$ that is applicable to obtain $f^{\mathbb{T}_\mathcal{R}(\sqcup D)}(\overline{t_n})$, by Lemma 5.4 and Lemma 5.6, is also applicable to obtain $\bigcup_{\mathcal{A} \in D} f^{\mathbb{T}_\mathcal{R}(\mathcal{A})}(\overline{t_n})$, and this expression is $f^{\sqcup\{\mathbb{T}_\mathcal{R}(\mathcal{A}) \mid \mathcal{A} \in D\}}(\overline{t_n})$. The inclusion in the other way is trivial.

□

Now the existence of the least fixed-point of $\mathbb{T}_\mathcal{R}$ is guaranteed (see e.g., [56]) and is characterized as $\sqcup A_\mathcal{R}$, being $A_\mathcal{R}$ the chain of pattern algebras $\mathcal{A}_i$, $i \in \mathbb{N}$, such that $\mathcal{A}_0 = \perp_\Sigma \sqsubseteq \cdots \sqsubseteq \mathcal{A}_{i+1} = \mathbb{T}_\mathcal{R}(\mathcal{A}_i) \sqsubseteq \cdots$. We denote this least fixed-point (that is also the *least pre-fixpoint*) as $\mathfrak{F}_\mathcal{R}$, that is usually denoted also as $\mathbb{T}_\mathcal{R}^\omega(\perp_\Sigma)$ (see e.g., [61]). From these notions we can finally prove that $\mathfrak{F}_\mathcal{R}$ coincides with $\mathcal{M}_\mathcal{R}$ with the help of two auxiliary lemmas. The first one, namely Lemma 5.8, establishes that models of programs are the only pattern algebras that, when applying the operator $\mathbb{T}_\mathcal{R}$ to them, do not incorporate additional information to the interpretation of any $f \in \Sigma_d$. The second auxiliary result, namely Lemma 5.9, establishes that reduction statements that can be performed from a *CPRS* $\mathcal{R}$ to get values are satisfied by some algebra in the chain $A_\mathcal{R}$.

Lemma 5.8 (Model characterization). *Given a CPRS $\mathcal{R}$, a pattern algebra $\mathcal{M}$ is a GHRC-model of $\mathcal{R}$ if and only if $\mathbb{T}_\mathcal{R}(\mathcal{M}) \sqsubseteq \mathcal{M}$.*

Proof First, we prove that $\mathbb{T}_\mathcal{R}(\mathcal{M}) \sqsubseteq \mathcal{M}$ for each pattern algebra $\mathcal{M}$ which is a *GHRC*-model of $\mathcal{R}$. Let us consider $f^{\mathbb{T}_\mathcal{R}(\mathcal{M})}(\overline{t_n})$ for $f \in \Sigma_d$ with $arity(f) = n > 0$ and $t_i \in Val(\Sigma_\perp, \mathcal{V})$ for $i = 1, \ldots, n$. If there exists a rule instance $(f(\overline{l_n}) \to r \Leftarrow C) \in [\mathcal{R}]_\perp$ with $r \neq \perp$, $\overline{l_n} \sqsubseteq \overline{t_n}$, and $\mathcal{M} \vDash_\varepsilon C$, then, as $\mathcal{M}$ is a *GHRC*-model of $\mathcal{R}$, $[\![r]\!]_\varepsilon^\mathcal{M} \subseteq [\![f(\overline{l_n})]\!]_\varepsilon^\mathcal{M}$. Therefore, $[\![f(\overline{l_n})]\!]_\varepsilon^\mathcal{M} = f^\mathcal{M}(\overline{l_n})$, and by $f^\mathcal{M}$ monotonicity, $f^\mathcal{M}(\overline{l_n}) \subseteq f^\mathcal{M}(\overline{t_n})$, and $[\![r]\!]_\varepsilon^\mathcal{M} \subseteq f^\mathcal{M}(\overline{t_n})$. Thus, $f^{\mathbb{T}_\mathcal{R}(\mathcal{M})}(\overline{t_n}) \subseteq f^\mathcal{M}(\overline{t_n})$, and consequently, $\mathbb{T}_\mathcal{R}(\mathcal{M}) \sqsubseteq \mathcal{M}$. For $f \in \Sigma_d$ with $arity(f) = 0$, the proof is simpler.

Now, we prove that every pattern algebra $\mathcal{M}$ such that $\mathbb{T}_\mathcal{R}(\mathcal{M}) \sqsubseteq \mathcal{M}$ is a *GHRC*-model of $\mathcal{R}$. Given a rule $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$, for $\theta \in VSubst(\Sigma_\perp, \mathcal{V})$ such that $\mathcal{M} \vDash_\varepsilon C\theta$, or equivalently, $\mathcal{M} \vDash_\theta C$, we can consider $f^{\mathbb{T}_\mathcal{R}(\mathcal{M})}(\overline{t_n\theta})$, and because of the instance $(f(\overline{l_n}) \to r \Leftarrow C)\theta$, we have $[\![r\theta]\!]_\varepsilon^\mathcal{M} \subseteq f^{\mathbb{T}_\mathcal{R}(\mathcal{M})}(\overline{t_n\theta})$. By initial hypothesis, $f^{\mathbb{T}_\mathcal{R}(\mathcal{M})}(\overline{t_n\theta}) \subseteq f^\mathcal{M}(\overline{t_n\theta})$, $[\![r\theta]\!]_\varepsilon^\mathcal{M} = [\![r]\!]_\theta^\mathcal{M}$, and $f^\mathcal{M}(\overline{t_n\theta}) = [\![f(\overline{t_n})]\!]_\theta^\mathcal{M}$; thus $[\![r]\!]_\theta^\mathcal{M} \subseteq [\![f(\overline{t_n})]\!]_\theta^\mathcal{M}$, which is $\mathcal{M} \vDash_\theta (f(\overline{l_n}) \twoheadrightarrow r)$, and then $\mathcal{M}$ satisfies the rule $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$.

□

Lemma 5.9 (Semantic characterization). *Given a CPRS $\mathcal{R}$, $s \in T(\Sigma_\perp, \mathcal{V})$, and $t \in Val(\Sigma_\perp, \mathcal{V})$, if $\mathcal{R} \vdash s \twoheadrightarrow t$ then $\mathcal{A}_i \models_\varepsilon (s \twoheadrightarrow t)$, for some $\mathcal{A}_i \in A_\mathcal{R}$.*

Proof Since $\mathbb{T}_\mathcal{R}(\sqcup A_\mathcal{R}) = \sqcup A_\mathcal{R}$, by the *Model characterization lemma* (Lemma 5.8), $\sqcup A_\mathcal{R}$ will be a *GHRC*-model of $\mathcal{R}$. Thus, by Theorem 4.8, $\mathcal{R} \vdash s \twoheadrightarrow t$ implies $\sqcup A_\mathcal{R} \models_\varepsilon (s \twoheadrightarrow t)$, or by Definition 4.6, $\langle t \rangle \subseteq [\![s]\!]_\varepsilon^{\sqcup A_\mathcal{R}}$, that is equivalent to $t \in [\![s]\!]_\varepsilon^{\sqcup A_\mathcal{R}}$. By the second item of Lemma 5.4, $[\![s]\!]_\varepsilon^{\sqcup A_\mathcal{R}} = \bigcup_{\mathcal{A}_i \in A_\mathcal{R}} [\![s]\!]_\varepsilon^{\mathcal{A}_i}$, so there will be $\mathcal{A}_i \in A_\mathcal{R}$ such that $t \in [\![s]\!]_\varepsilon^{\mathcal{A}_i}$, that means $\mathcal{A}_i \models_\varepsilon (s \twoheadrightarrow t)$.

□

Finally, we can prove the main result of this section, that is the characterization of the pattern model of a *CPRS* $\mathcal{R}$ as the least fixed-point $\mathfrak{F}_\mathcal{R}$ of the operator $\mathbb{T}_\mathcal{R}$ over pattern algebras.

Theorem 5.10 (Fixed-Point characterization of the least model). *For every given CPRS $\mathcal{R}$, $\mathcal{M}_\mathcal{R}$ is the least fixed-point (and the least pre-fixpoint) of $\mathbb{T}_\mathcal{R}$.*

Proof First, we can prove $\sqcup A_\mathcal{R} \sqsubseteq \mathcal{M}_\mathcal{R}$: from $\mathcal{A}_0 \sqsubseteq \mathcal{M}_\mathcal{R}$, $\mathbb{T}_\mathcal{R}(\mathcal{M}_\mathcal{R}) \sqsubseteq \mathcal{M}_\mathcal{R}$ (because $\mathcal{M}_\mathcal{R}$ is a model of $\mathcal{R}$ and Lemma 5.8), and the continuity of $\mathbb{T}_\mathcal{R}$ from Lemma 5.7 that assures $\mathcal{A}_i \sqsubseteq \mathcal{M}_\mathcal{R}$ for all i. Now, we can prove that $\mathcal{M}_\mathcal{R} \sqsubseteq \sqcup A_\mathcal{R}$ by proving, for each $f \in \Sigma_d$, that $f^{\mathcal{M}_\mathcal{R}}(\overline{t_n}) \subseteq f^{\sqcup A_\mathcal{R}}(\overline{t_n})$, for $t_1, \ldots, t_n \in Val(\Sigma_\perp, \mathcal{V})$. This inclusion is proved as follows: by Definition 4.7, $t \in f^{\mathcal{M}_\mathcal{R}}(\overline{t_n})$ is equivalent to $\mathcal{R} \vdash f(\overline{t_n}) \twoheadrightarrow t$, and this implies $\mathcal{A}_i \models_\varepsilon (f(\overline{t_n}) \twoheadrightarrow t)$ for some $\mathcal{A}_i \in A_\mathcal{R}$ by Lemma 5.9. Taking into account that, $[\![f(\overline{t_n})]\!]_\varepsilon^{\mathcal{A}_i} = f^{\mathcal{A}_i}(\overline{t_n})$, and we obtain $t \in f^{\mathcal{A}_i}(\overline{t_n})$. Then, $t \in f^{\sqcup A_\mathcal{R}}(\overline{t_n})$.
□

Thus, if we consider the meaning of a *CPRS* $\mathcal{R}$ as the least fixed-point of its associated transformer $\mathbb{T}_\mathcal{R}$, then this fixed-point semantics coincides with the model-theoretic semantics as it happens in logic programming [62]. In fact, the fixed-point semantics corresponds to the *C-semantics* given in [57]. This fixed-point characterization of the semantics of a program will be essential in the next section to define appropriate semantics for a modular extension of the framework in order to fulfill essential modular properties of the semantics.

6. Modular Semantics

In this section we enrich our higher-order declarative programming framework with λ-abstractions to support modular programming by means of the definition of program modules and modular operations between them. We also provide an adequate semantics for modular higher-order programming by proving that it is compositional and fully abstract with respect to a basic set of modular operations.

The modularity of the semantics is particularly relevant in declarative programming, because one of the most critical aspects in multiparadigm declarative systems is the possibility of making a separate compilation of modules, and this can only be made in the presence of some kind of compositionality. For instance, $\mathcal{TOY}$ [63] is a constraint functional logic system, designed to support the main declarative programming styles and their combination. The current version provides a module system involving higher-order patterns, a polymorphic type system, constraints with symbolic equations and disequations, linear and non-linear arithmetic constraints over real

numbers, and finite domain constraints [64]. For this reason, our modular semantics for higher-order declarative constraint programming can be applied to the $\mathcal{TOY}$ system to allow each distinct module to be compiled separately, with the effect of inlining being realized by a subsequent linking process. Moreover, modular development installs boundaries in programs that can be important to the practical use of static analysis techniques and that are fundamental to the notion of separate compilation and testing.

6.1. Modules

A program module is a *CPRS-program* that defines an exported signature and imports some defined function symbols from other modules. Modules consist on *CPRS*-programs that can invoke some defined function symbols in Σ_d but have no rules defining those symbols, which can be defined in other modules. For simplicity, we represent modules as tuples containing the exported and imported signatures, and the program rules.

Definition 6.1 (Modules). *Let Σ be a signature. A CPRS-module over Σ is a tuple:*

$$\mathcal{M} = \langle\, \Sigma^{\mathcal{M}}, \Sigma_d^{\mathcal{M}}, \mathcal{R}_{\mathcal{M}} \,\rangle$$

where:

- *$\mathcal{R}_{\mathcal{M}}$ is a CPRS-program.*
- *$\Sigma^{\mathcal{M}} \subseteq \Sigma_d$ are the function symbols in Σ with no definition rule in $\mathcal{R}_{\mathcal{M}}$ that are invoked in $\mathcal{R}_{\mathcal{M}}$ (i.e., that appear in the rhs of the rules in $\mathcal{R}_{\mathcal{M}}$).*
- *$\Sigma_d^{\mathcal{M}} \subseteq \Sigma_d$ are the function symbols defined in $\mathcal{R}_{\mathcal{M}}$.*

The *CPRS*-program $\mathcal{R}_{\mathcal{M}}$ is called the *body* of the module and $\langle\, \Sigma^{\mathcal{M}}, \Sigma_d^{\mathcal{M}} \,\rangle$ its *interface*. More precisely, $\Sigma_d^{\mathcal{M}}$ is the *exported signature* of defined function symbols, that is the set of all function symbols defined in $\mathcal{R}_{\mathcal{M}}$, and $\Sigma^{\mathcal{M}}$ is the parameter (*imported*) signature, with all the function symbols invoked but not defined in $\mathcal{R}_{\mathcal{M}}$. With this definition, the interface of a module can be inferred from its body, since all the symbols that are defined in the program are exported and every defined function symbol that is invoked but not defined is imported.

Example 6.2. *Let $\mathcal{R}$ be the CPRS of Example 2.6. We can write a module $\mathcal{M}_1$ with the differentiation function 'diff' but not including the operations on natural numbers, that are relegated to another module $\mathcal{M}_2$. Doing this, it is possible to develop a different implementation of arithmetic operations and use it in the differentiation module $\mathcal{M}_1$ just by changing the module $\mathcal{M}_2$ that is imported:*

(1) Module $\mathcal{M}_1$ contains a single function 'diff' to evaluate the differentiation of mathematical functions at a certain point. The rules defining the differentiation invoke addition and multiplication functions that are not defined in the module.

$\mathcal{M}_1 = \langle\, \Sigma^{\mathcal{M}_1}, \Sigma_d^{\mathcal{M}_1}, \mathcal{R}_{\mathcal{M}_1} \,\rangle$ *where:*

- $\Sigma^{\mathcal{M}_1} = \{add, mult\}$
- $\Sigma_d^{\mathcal{M}_1} = \{diff\}$

- $\mathcal{R}_{\mathcal{M}_1} = \{$

$$
\begin{array}{lcl}
diff(\lambda u.F, x) & \rightarrow & zero \\
diff(\lambda u.u, x) & \rightarrow & s(zero) \\
diff(\lambda u.add(F(u), G(u)), x) & \rightarrow & add(diff(\lambda u.F(u), x), diff(\lambda u.G(u), x)) \\
diff(\lambda u.mult(F(u), G(u)), x) & \rightarrow & add(mult(diff(\lambda u.F(u), x), G(x)), \\
 & & \quad mult(diff(\lambda u.G(u), x), F(x)))
\end{array}
$$

$\}$

(2) Module $\mathcal{M}_2$ contains the definition of functions for the addition and multiplication of natural numbers.

$\mathcal{M}_2 = \langle\, \Sigma^{\mathcal{M}_2}, \Sigma_d^{\mathcal{M}_2}, \mathcal{R}_{\mathcal{M}_2} \,\rangle$ *where:*

- $\Sigma^{\mathcal{M}_2} = \emptyset$
- $\Sigma_d^{\mathcal{M}_2} = \{add, mult\}$
- $\mathcal{R}_{\mathcal{M}_2} = \{$

$$
\begin{array}{lcl}
add(x, zero) & \rightarrow & x \\
add(x, s(y)) & \rightarrow & s(add(x, y)) \\
 & & \\
mult(x, zero) & \rightarrow & zero \\
mult(x, s(y)) & \rightarrow & add(x, mult(x, y))
\end{array}
$$

$\}$

□

Our modular framework defined for higher-order functional logic programming consists of a small reduced number of operations over *CPRS*-modules. We present this set of basic operations to express typical features of modularization techniques in declarative programming [65]. We can define a modular design for *CPRS*-programs by means of successive applications of these operations. We focus on the most basic composition operations over declarative programs, the *union* of two modules and the *deletion* of a signature in a module.

- *Union of modules.* The union of modules reflects the characteristic of the majority of programming systems that allow adding program definitions stored in several files to the executable code. We define the union of two modules as the new module obtained as the simple union of signatures and rules. Formally, given two modules $\mathcal{M}_1 = \langle\, \Sigma^{\mathcal{M}_1}, \Sigma_d^{\mathcal{M}_1}, \mathcal{R}_{\mathcal{M}_1} \,\rangle$ and $\mathcal{M}_2 = \langle\, \Sigma^{\mathcal{M}_2}, \Sigma_d^{\mathcal{M}_2}, \mathcal{R}_{\mathcal{M}_2} \,\rangle$, their union $\mathcal{M}_1 \cup \mathcal{M}_2$ is defined as the new module:

$$\mathcal{M}_1 \cup \mathcal{M}_2 =_{def} \langle (\Sigma^{\mathcal{M}_1} \cup \Sigma^{\mathcal{M}_2}) \setminus (\Sigma_d^{\mathcal{M}_1} \cup \Sigma_d^{\mathcal{M}_2}), \Sigma_d^{\mathcal{M}_1} \cup \Sigma_d^{\mathcal{M}_2}, \mathcal{R}_{\mathcal{M}_1} \cup \mathcal{R}_{\mathcal{M}_2} \,\rangle$$

  We note that each argument in this operation is considered as an open *CPRS*-program that can be extended or completed with the other argument, possibly with additional rules for its exported signature.

- *Deletion of a signature.* The deletion of a signature Σ in a module $\mathcal{R}$ removes all rules defining function symbols in Σ in the signature $\Sigma_d^{\mathcal{M}}$, but maintains the occurrences of these

symbols in the rhs of the program rules. This operation can be used to abstract a signature Σ from a module $\mathcal{R}$, and is very useful for making generic modules from concrete ones. Formally, given a module $\mathcal{M} = \langle\, \Sigma^{\mathcal{M}}, \Sigma_d^{\mathcal{M}}, \mathcal{R}_{\mathcal{M}} \,\rangle$, the deletion in $\mathcal{M}$ of a signature Σ of function symbols produces the module:

$$\mathcal{M}\backslash\Sigma =_{def} \langle\, \Sigma'^{\mathcal{M}}, \Sigma_d^{\mathcal{M}} \setminus \Sigma, \mathcal{R}_{\mathcal{M}}\backslash\Sigma \,\rangle$$

The notation $\mathcal{R}_{\mathcal{M}}\backslash\Sigma$ denotes the set of those rules in $\mathcal{R}_{\mathcal{M}}$ defining function symbols not appearing in Σ, and $\Sigma'^{\mathcal{M}}$ denotes the corresponding parameter signature with all the symbols that occur in the rhs of some rules in $\mathcal{R}_{\mathcal{M}}\backslash\Sigma$ but are not defined in the resulting module.

The high expressiveness of these operations enable us to model typical constructs for program modularization like *export/import*, *instantiation*, and *inheritance with overriding* in a simple way.

- *Inheritance.* From the union and deletion operations we can model an inheritance relationship between modules. *Inheritance with overriding* may be captured by means of union and deletion as follows:

$$\mathcal{M}_1 \ll \mathcal{M}_2 =_{def} \mathcal{M}_1 \cup (\mathcal{M}_2 \setminus \Sigma_d^{\mathcal{M}_1})$$

This new module $\mathcal{M}_1 \ll \mathcal{M}_2$ inherits all functions in $\mathcal{M}_2$ not defined in $\mathcal{M}_1$ together with their rules, and uses the rules of $\mathcal{M}_1$ for all the functions defined there, overriding the definition rules in $\mathcal{M}_2$. In this case, overriding is carried out by deleting the common signature of the inherited module before adding it to the derived module.

- *Instantiation.* We can instantiate function symbols of a module $\mathcal{M}_1 = \langle\, \Sigma^{\mathcal{M}_1}, \Sigma_d^{\mathcal{M}_1}, \mathcal{R}_{\mathcal{M}_1} \,\rangle$ with function symbols exported by other module $\mathcal{M}_2$, simply by renaming suitably the functions of $\mathcal{M}_1$ to fit (a part of) the exported signature of $\mathcal{M}_2$. Thus, we obtain an *instantiation operation* that we denote $\mathcal{M}_1[\mathcal{M}_2, \theta]$ and formally define as:

$$\mathcal{M}_1[\mathcal{M}_2, \theta] =_{def} \mathcal{M}_2 \ll \theta(\mathcal{M}_1)$$

The renaming substitution of function symbols θ characterizes the instantiation, and is defined as $\theta(\mathcal{M}_1) =_{def} \langle\, \theta(\Sigma^{\mathcal{M}_1}) \setminus \theta(\Sigma_d^{\mathcal{M}_1}), \theta(\Sigma_d^{\mathcal{M}_1}), \mathcal{R}_{\mathcal{M}_1}\theta \,\rangle$. The renaming operation allows us to change function symbols with other ones in a given signature.

Example 6.3. *In this example we show how each of the four defined operations with modules work considering $\mathcal{M}_1$ and $\mathcal{M}_2$ from Example 6.2, $\Sigma_1 = \{add\}$, $\theta = \{add \mapsto addalt\}$ and two modules $\mathcal{M}_3$ and $\mathcal{M}_4$ that define the addition operation by recursion on its first argument with different but equivalent rules and they employ different names for the operation:*

1. $\mathcal{M}_3 = \langle\, \Sigma^{\mathcal{M}_3}, \Sigma_d^{\mathcal{M}_3}, \mathcal{R}_{\mathcal{M}_3} \,\rangle$
   - $\Sigma^{\mathcal{M}_3} = \emptyset$
   - $\Sigma_d^{\mathcal{M}_3} = \{add\}$
   - $\mathcal{R}_{\mathcal{M}_3} = \{$
     $add(zero, x) \quad \rightarrow \quad x$
     $add(s(x), y) \quad \rightarrow \quad s(add(x, y))$
     $\}$

2. $\mathcal{M}_4 = \langle\, \Sigma^{\mathcal{M}_4}, \Sigma_d^{\mathcal{M}_4}, \mathcal{R}_{\mathcal{M}_4} \,\rangle$

   - $\Sigma^{\mathcal{M}_4} = \emptyset$
   - $\Sigma_d^{\mathcal{M}_4} = \{addren\}$
   - $\mathcal{R}_{\mathcal{M}_4} = \{$

     $addalt(zero, x) \quad \rightarrow \quad x$
     $addalt(s(zero), x) \quad \rightarrow \quad s(x)$
     $addalt(s(s(x)), y) \quad \rightarrow \quad s(addalt(s(x), y))$

     $\}$

From these modules, we consider now an elaborated example illustrating the application of the four operations defined before:

1. $\mathcal{M}_1 \cup \mathcal{M}_2 = \langle\, \Sigma^{\mathcal{M}_1 \cup \mathcal{M}_2}, \Sigma_d^{\mathcal{M}_1 \cup \mathcal{M}_2}, \mathcal{R}_{\mathcal{M}_1 \cup \mathcal{M}_2} \,\rangle$

   - $\Sigma^{\mathcal{M}_1 \cup \mathcal{M}_2} = (\Sigma^{\mathcal{M}_1} \cup \Sigma^{\mathcal{M}_2}) \setminus (\Sigma_d^{\mathcal{M}_1} \cup \Sigma_d^{\mathcal{M}_2}) = (\{add, mult\} \cup \emptyset) \setminus (\{diff\} \cup \{add, mult\}) = \{add, mult\} \setminus \{diff, add, mult\} = \emptyset$
   - $\Sigma_d^{\mathcal{M}_1 \cup \mathcal{M}_2} = \{diff\} \cup \{add, mult\} = \{diff, add, mult\}$
   - $\mathcal{R}_{\mathcal{M}_1 \cup \mathcal{M}_2} = \{$

     $diff(\lambda u.F, x) \quad \rightarrow \quad zero$
     $diff(\lambda u.u, x) \quad \rightarrow \quad s(zero)$
     $diff(\lambda u.add(F(u), G(u)), x) \quad \rightarrow \quad add(diff(\lambda u.F(u), x), diff(\lambda u.G(u), x))$
     $diff(\lambda u.mult(F(u), G(u)), x) \quad \rightarrow \quad add(mult(diff(\lambda u.F(u), x), G(x)),$
     $\qquad mult(diff(\lambda u.G(u), x), F(x)))$

     $add(x, zero) \quad \rightarrow \quad x$
     $add(x, s(y)) \quad \rightarrow \quad s(add(x, y))$

     $mult(x, zero) \quad \rightarrow \quad zero$
     $mult(x, s(y)) \quad \rightarrow \quad add(x, mult(x, y))$

     $\}$

2. $\mathcal{M}_2 \backslash \Sigma_1 = \langle\, \Sigma^{\mathcal{M}_2 \backslash \Sigma_1}, \Sigma_d^{\mathcal{M}_2 \backslash \Sigma_1}, \mathcal{R}_{\mathcal{M}_2 \backslash \Sigma_1} \,\rangle$

   - $\Sigma^{\mathcal{M}_2 \backslash \Sigma_1} = \Sigma'^{\mathcal{M}_2} = \{add\}$
   - $\Sigma_d^{\mathcal{M}_2 \backslash \Sigma_1} = \Sigma_d^{\mathcal{M}_2} \setminus \Sigma_1 = \{add, mult\} \setminus \{add\} = \{mult\}$
   - $\mathcal{R}_{\mathcal{M}_2 \backslash \Sigma_1} = \{$

     $mult(x, zero) \quad \rightarrow \quad zero$
     $mult(x, s(y)) \quad \rightarrow \quad add(x, mult(x, y))$

     $\}$

3. $\mathcal{M}_3 \ll \mathcal{M}_2 = \mathcal{M}_3 \cup (\mathcal{M}_2 \setminus \Sigma_d^{\mathcal{M}_3}) = \langle\, \Sigma^{\mathcal{M}_3 \ll \mathcal{M}_2}, \Sigma_d^{\mathcal{M}_3 \ll \mathcal{M}_2}, \mathcal{R}_{\mathcal{M}_3 \ll \mathcal{M}_2} \,\rangle$

   - $\Sigma^{\mathcal{M}_3 \ll \mathcal{M}_2} = \emptyset$
   - $\Sigma_d^{\mathcal{M}_3 \ll \mathcal{M}_2} = \{add, mult\}$

- $\mathcal{R}_{\mathcal{M}_3 \ll \mathcal{M}_2} = \{$
  $add(zero, x) \rightarrow x$
  $add(s(x), y) \rightarrow s(add(x, y))$

  $mult(x, zero) \rightarrow zero$
  $mult(x, s(y)) \rightarrow add(x, mult(x, y))$
  $\}$

4. $\mathcal{M}_2[\mathcal{M}_4, \theta] = \mathcal{M}_4 \ll \theta(\mathcal{M}_2) = \mathcal{M}_4 \cup (\theta(\mathcal{M}_2) \setminus \Sigma_d^{\mathcal{M}_4}) = \langle\, \Sigma^{\mathcal{M}_2[\mathcal{M}_4,\theta]}, \Sigma_d^{\mathcal{M}_2[\mathcal{M}_4,\theta]}, \mathcal{R}_{\mathcal{M}_2[\mathcal{M}_4,\theta]} \,\rangle$
   - $\Sigma^{\mathcal{M}_2[\mathcal{M}_4,\theta]} = \emptyset$
   - $\Sigma_d^{\mathcal{M}_2[\mathcal{M}_4,\theta]} = \{addalt, mult\}$
   - $\mathcal{M}_2[\mathcal{M}_4, \theta] = \{$
     $addalt(zero, x) \rightarrow x$
     $addalt(s(zero), x) \rightarrow s(x)$
     $addalt(s(s(x)), y) \rightarrow s(addalt(s(x), y))$

     $mult(x, zero) \rightarrow zero$
     $mult(x, s(y)) \rightarrow add(x, mult(x, y))$
     $\}$

□

6.2. Compositional and Fully Abstract Semantics

An important aspect to be considered when a declarative language is extended for modular programming is the sound integration of the behavior of the modular operations into the semantics of the language. In this setting, the properties of *compositionality* and *full abstraction* have been recognized as two fundamental concepts in the studies on the semantics of declarative modular programming languages [66]. Simply stated, a semantics is *compositional* in a modular approach if semantically equivalent program modules are indistinguishable by means of module operations, that is, the meaning of a module can be deduced from the meaning of its components. Compositionality of the semantics ensures that program modules which are semantically equivalent can be replaced with other ones without affecting the intended semantics of the whole system. For instance, this property establishes a firm foundation for reasoning about programs and program transformations [67]: suppose a module $\mathcal{M}$ consisting of several components $\mathcal{M}_1, \ldots, \mathcal{M}_n$, suitably composed together by means of module operations; suppose also that $\mathcal{M}_i'$ is a transformed version of $\mathcal{M}_i$ (for example, a more efficient version), obtained by applying some program transformation technique to program $\mathcal{R}_i$ that corresponds to module $\mathcal{M}_i$. If $\mathcal{M}_i'$ is equivalent to $\mathcal{M}_i$ in the chosen semantics, then the property of compositionality ensures that the substitution of $\mathcal{M}_i'$ for $\mathcal{M}_i$ will not affect the meaning of the whole module $\mathcal{M}$. On the other hand, the property of *full abstraction* establishes that the equivalence relation induced by the semantics is the largest equivalence relation that can be used to substitute program modules without affecting the intended semantics of the whole system. In other words, a semantics is fully abstract if all indistinguishable program modules are semantically equivalent.

In this section we define a suitable modular semantics for *CPRS*-modules from the pattern algebra transformer defined in Section 5. This immediate consequence operator captures directly

the information concerning possible compositions obtained by the union of signatures and rules, and the corresponding semantics is directly *compositional* by construction with respect to the union operation of program modules. However, in order to obtain the complementary property for compositionality, that is the *full abstraction* property, the adequacy of this semantics must be established with respect to the deletion operation of a signature in a program module, used to delete whole sets of program rules defining functions. Therefore, we define a compositional and fully abstract semantics for the reduced set of operations on program modules defined in Subsection 6.1, union and deletion, that are enough to express, as we have seen, the most usual ways of composing modules and their relationships.

In this work we adopt an approach inspired in [66, 68], where compositionality and full abstraction are defined in terms of the equivalence relation induced by the *GHRC*-semantics. Without loss of generality, in the sequel we consider *CPRS*-programs instead of *CPRS*-modules since modules can be easily deduced from programs, and this allow us to present our results in a more natural way.

Definition 6.4 (Modular semantics). *Let us consider a GHRC-semantic $\mathcal{S}$ and its corresponding equivalence relation $\sim^{\mathcal{S}}$ (i.e., two CPRS-programs are $\sim^{\mathcal{S}}$-equivalent if and only if they have the same meaning (denotation) in the semantics $\mathcal{S}$). We define:*

1. *$\mathcal{S}$ is compositional if:*
   (a) *For all CPRS-programs $\mathcal{R}_1$ and $\mathcal{R}_2$:*

$$\mathcal{R}_1 \sim^{\mathcal{S}} \mathcal{R}_2 \;\Rightarrow\; \mathcal{M}_{\mathcal{R}_1} = \mathcal{M}_{\mathcal{R}_2}$$

   (b) *For all CPRS-programs $\mathcal{R}_{1i}$ and $\mathcal{R}_{2i}$ and signature Σ, for all $i \in \{1, \ldots, n\}$, and all $Op \in \{\cup, (\cdot)\backslash\Sigma\}$:*

$$\mathcal{R}_{1i} \sim^{\mathcal{S}} \mathcal{R}_{2i} \;\Rightarrow\; Op(\overline{\mathcal{R}_{1n}}) \sim^{\mathcal{S}} Op(\overline{\mathcal{R}_{2n}})$$

2. *$\mathcal{S}$ is fully abstract if and only if for all CPRS-programs $\mathcal{R}_1$ and $\mathcal{R}_2$:*

$$\mathcal{M}_{\mathtt{C}[\![\mathcal{R}_1]\!]} = \mathcal{M}_{\mathtt{C}[\![\mathcal{R}_2]\!]} \;\Rightarrow\; \mathcal{R}_1 \sim^{\mathcal{S}} \mathcal{R}_2$$

   for all context $\mathtt{C}$, where contexts $\mathtt{C}[\![\mathcal{X}]\!]$ are inductively defined as follows: the metavariable $\mathcal{X}$ and each CPRS-program is a context, and for each $Op \in \{\cup, (\cdot)\backslash\Sigma\}$ and $\mathtt{C}_1, \ldots, \mathtt{C}_n$ contexts, $Op(\mathtt{C}_1, \ldots, \mathtt{C}_n)$ is a context.

To find a compositional semantics in our higher-order programming framework, we can build *CPRS*-programs from other ones adding program rules for new functions or for already defined functions, and consider them as pattern algebra transformers as done in [66, 68]. However, to obtain full abstraction with respect to the deletion operation, we have to consider pattern models of *CPRS*-programs obtained by deleting any signature. This motivates the following definition:

Definition 6.5 (Transformer semantics).

- *We define the **pattern algebra transformer semantics** $\mathbb{T}$ of a CPRS-program $\mathcal{R}$ by denoting the meaning of $\mathcal{R}\backslash\Sigma$, for all signature Σ, by its pattern algebra transformer $\mathbb{T}_{\mathcal{R}}\backslash\Sigma$. By applying Theorem 5.10:*

$$[\![\mathcal{R}]\!]_{\mathbb{T}} = \{\mathcal{M} \mid \mathcal{M} \text{ is a pattern model of } \mathcal{R}\backslash\Sigma\}$$

for all signature Σ.

- *Two CPRS-programs $\mathcal{R}_1$ and $\mathcal{R}_2$ are $\sim^{\mathbb{T}}$-equivalents if and only if both define the same pattern algebra transformer by deleting any signature. By applying Theorem 5.10:*

$$\mathcal{R}_1 \sim^{\mathbb{T}} \mathcal{R}_2 \;\Leftrightarrow\; [\![\mathcal{R}_1]\!]_{\mathbb{T}} = [\![\mathcal{R}_2]\!]_{\mathbb{T}}$$

We are now ready to state and prove the main properties of compositionality and full abstraction for the pattern algebra transformer semantics $\mathbb{T}$, which justify the adoption of this semantics as the main point of our modular framework.

Theorem 6.6 (Modularity of $\mathbb{T}$). *The pattern algebra transformer semantics $\mathbb{T}$ is compositional and fully abstract with respect to the set of operations $\{\cup, (\cdot)\backslash\Sigma\}$.*

Proof We prove that the pattern algebra transformer semantics $\mathbb{T}$ is compositional with respect to the set of operations $\{\cup, (\cdot)\backslash\Sigma\}$. First, from $\mathcal{P} \sim^{\mathbb{T}} \mathcal{Q}$ we deduce that $\mathcal{P}\backslash\Sigma$ and $\mathcal{Q}\backslash\Sigma$ have the same pattern models for all signature Σ. In particular, for the empty signature, $\mathcal{P}$ and $\mathcal{Q}$ have the same least pattern models and the same least fixed-points. Thus, we conclude that $\mathcal{M}_{\mathcal{P}} = \mathcal{M}_{\mathcal{Q}}$. Now, we prove that the pattern algebra transformer semantics $\mathbb{T}$ is compositional with respect to the union of programs: $\mathcal{P}_i \sim^{\mathbb{T}} \mathcal{Q}_i$, for $i = 1, ..., n$, implies $\bigcup_{i=1}^n \mathcal{P}_i \sim^{\mathbb{T}} \bigcup_{i=1}^n \mathcal{Q}_i$. Since $\mathcal{P}_i \sim^{\mathbb{T}} \mathcal{Q}_i$, both define the same pattern algebra transformer $\mathbb{T}_{\mathcal{P}_i\backslash\Sigma} = \mathbb{T}_{\mathcal{Q}_i\backslash\Sigma}$ for all signature Σ. Let $\mathcal{M}$ be a pattern model of $\bigcup_{i=1}^n \mathcal{P}_i\backslash\Sigma$. From Theorem 5.10: $\bigsqcup_{i=1}^n \mathbb{T}_{\mathcal{P}_i\backslash\Sigma}(\mathcal{M}) = (\bigsqcup_{i=1}^n \mathbb{T}_{\mathcal{P}_i\backslash\Sigma})(\mathcal{M}) = \mathbb{T}_{\bigcup_{i=1}^n \mathcal{P}_i\backslash\Sigma}(\mathcal{M}) \sqsubseteq \mathcal{M}$. Therefore, $\mathbb{T}_{\mathcal{P}_i\backslash\Sigma}(\mathcal{M}) \sqsubseteq \mathcal{M}$ for $i = 1, \ldots, n$. Moreover, from $\mathcal{P}_i \sim^{\mathbb{T}} \mathcal{Q}_i$ we obtain $\mathbb{T}_{\mathcal{Q}_i\backslash\Sigma}(\mathcal{M}) \sqsubseteq \mathcal{M}$ for $i = 1, \ldots, n$, and then $\mathbb{T}_{\bigcup_{i=1}^n \mathcal{Q}_i\backslash\Sigma}(\mathcal{M}) = (\bigsqcup_{i=1}^n \mathbb{T}_{\mathcal{Q}_i\backslash\Sigma})(\mathcal{M}) = \bigsqcup_{i=1}^n \mathbb{T}_{\mathcal{Q}_i\backslash\Sigma}(\mathcal{M}) \sqsubseteq \mathcal{M}$. Therefore, $\mathcal{M}$ is a pattern model of $\bigcup_{i=1}^n \mathcal{Q}_i\backslash\Sigma$. By reasoning in a similar way, it can be obtained that all pattern models of $\bigcup_{i=1}^n \mathcal{Q}_i\backslash\Sigma$ are also pattern models of $\bigcup_{i=1}^n \mathcal{P}_i\backslash\Sigma$, and this proves that $\mathbb{T}_{\bigcup_{i=1}^n \mathcal{P}_i\backslash\Sigma} = \mathbb{T}_{\bigcup_{i=1}^n \mathcal{Q}_i\backslash\Sigma}$. Finally, since $(\bigcup_{i=1}^n \mathcal{P}_i)\backslash\Sigma = \bigcup_{i=1}^n (\mathcal{P}_i\backslash\Sigma)$, we deduce that $\mathbb{T}_{(\bigcup_{i=1}^n \mathcal{P}_i)\backslash\Sigma} = \mathbb{T}_{(\bigcup_{i=1}^n \mathcal{Q}_i)\backslash\Sigma}$ for all signature Σ, which means $\bigcup_{i=1}^n \mathcal{P}_i \sim^{\mathbb{T}} \bigcup_{i=1}^n \mathcal{Q}_i$.

Second, we prove that the pattern algebra transformer semantics $\mathbb{T}$ is compositional with respect to the deletion of a signature in a program: $P \sim^{\mathbb{T}} Q$ implies $\mathcal{P}\backslash\Sigma' \sim^{\mathbb{T}} \mathcal{Q}\backslash\Sigma'$, for every signature Σ'. Since $P \sim^{\mathbb{T}} Q$, both define the same pattern algebra transformer $\mathbb{T}_{\mathcal{P}\backslash(\Sigma\cup\Sigma')} = \mathbb{T}_{\mathcal{Q}\backslash(\Sigma\cup\Sigma')}$ for all signatures Σ and Σ'. Since $\mathcal{R}\backslash(\Sigma \cup \Sigma') = (\mathcal{R}\backslash\Sigma)\backslash\Sigma'$ for every *CPRS*-program $\mathcal{R}$, we deduce that $\mathbb{T}_{(\mathcal{P}\backslash\Sigma)\backslash\Sigma'} = \mathbb{T}_{(\mathcal{Q}\backslash\Sigma)\backslash\Sigma'}$ for every signatures Σ and Σ', which means that $\mathcal{P}\backslash\Sigma' \sim^{\mathbb{T}} \mathcal{Q}\backslash\Sigma'$, for every signature Σ'. Finally, to prove that the pattern algebra transformer semantics $\mathbb{T}$ is fully abstract, we only need to prove that $\mathcal{P} \nsim^{\mathbb{T}} \mathcal{Q}$ implies that there exists a context $\mathtt{C}$ where we can discriminate the observable behavior of both programs. From $\mathcal{P} \nsim^{\mathbb{T}} \mathcal{Q}$ we deduce that there exists a signature Σ such that the pattern algebra transformers $\mathbb{T}_{\mathcal{P}\backslash\Sigma}$ and $\mathbb{T}_{\mathcal{Q}\backslash\Sigma}$ are different. Then there exists a context $\mathtt{C}'$ such that $\mathcal{M}_{\mathtt{C}'[\![\mathcal{P}\backslash\Sigma]\!]} \neq \mathcal{M}_{\mathtt{C}'[\![\mathcal{Q}\backslash\Sigma]\!]}$. Thus by considering the new context $\mathtt{C}[\![\mathcal{R}]\!] = \mathtt{C}'[\![\mathcal{R}\backslash\Sigma]\!]$ we have that $\mathcal{M}_{\mathtt{C}[\![\mathcal{P}]\!]} \neq \mathcal{M}_{\mathtt{C}[\![\mathcal{Q}]\!]}$.

□

7. Related Work

In this section we compare our work to other existing approaches in the field of higher-order functional logic programming. We consider the frameworks referred in this section as the closest ones to our work in the context of higher-order functional logic programming.

In [69] was proposed a higher-order extension to the framework compiled in [31], without adding λ-abstractions to the language. This work claims that applicative expressions without λ-abstractions are expressive enough for many purposes; however, λ-abstractions are a powerful resource in some application fields such as symbolic computation or the design of logical circuits. This approach has been extended in [70, 71] and is currently implemented in the functional logic system $\mathcal{TOY}$ [63]. We consider this extension as a parallel line of research to ours, since both proposals have similar starting points and are based on similar principles, such as the *CRWL* first-order rewriting logic. However, our proposal is more devoted to provide semantic basis to develop verification and debugging techniques thanks to our modular semantics presented in Section 6.

A closely related framework for higher-order functional logic programming with λ-abstractions is the one developed in [30, 54], where is described a higher-order lazy narrowing calculus oriented to solve higher-order equations. This framework has foundations similar to ours, but with the key difference of its lack of an underlying rewriting logic, which leads to weaker semantics that also lack modularity.

In [72] it is developed a higher-order functional logic scheme with λ-abstraction that has been used for distributed constraint solving [73]; this scheme introduces a family of languages oriented to functional logic programming with constraints with higher-order features. Both works relies on higher-order lazy narrowing calculi and is implemented on top of the *Mathematica*^{TM} [20] system. However, its main limitation from our point of view is the lack of a suitable declarative semantics to support strong theoretical results about the soundness and completeness of the corresponding operational semantics in the line of [32].

To sum up, we consider this work as the first higher-order functional logic programming framework with λ-abstractions backed by a rewriting logic and declarative modular semantics where the main results guaranteing the correctness of the approach have been proved. This makes our framework an adequate theoretical foundations not only for the implementation of a higher-order functional logic programming system with λ-abstractions, but also to integrate advanced algorithmic debugging and formal verification techniques [33] that are easily supported thanks to the theoretical results proved in this work.

8. Conclusions and Future Work

In this work we have presented a self-contained exposition of a higher-order framework for functional logic programming with λ-abstractions. This work contains all the concepts and proofs of relevant results that guarantee the usefulness of the framework as the theoretical foundations of a higher-order functional logic system with λ-abstractions, and to be at the basis of operational semantics and implementations in multiparadigm declarative systems.

The language is developed syntactically in Section 2, as a higher-order extension of term rewriting systems based on simply-typed λ-calculus, that is a well-known and commonly adopted approach for integrating λ-expressions in declarative frameworks. In Section 3 we have presented the rewriting logic *GHRC* to provide a formal characterization of derivations by means of the *GHRC*-proof calculus in Figure 1, that is also a powerful tool to support algorithmic debugging techniques and formal verification methods, as is sketched in Example 3.3. Then we present the semantics of the framework; in Section 4 we provide classic model-theoretic semantics based on domain theory that characterizes univocally each program by a pattern model $\mathcal{M}_\mathcal{R}$ satisfying it, as defined in Definition 4.7; after that we establish soundness and completeness results of *GHRC*-provability with respect to semantic validity in models in Theorem 4.8. In Section 5 we have presented a complementary approach to the semantics in our framework, by characterizing the pattern model of a program as the least fixed-point of a continuous operator defined over pattern algebras, as stated in Theorem 5.10; this characterization of the semantics allows us to define an adequate modular semantics for our framework described in Section 6, and to prove the essential properties of compositionality and full abstraction of the modular semantics stated in Theorem 6.6.

With respect to extensions to the framework, it could be interesting to reformulate it in the context of more advanced type systems than the provided by the simply typed λ-calculus in this work; for example, it would be nice to extend with a polymorphic type system to increase its expressivity; even though higher-order patterns are problematic in the context of polymorphic types [74], recent results in this area [75] may be considered when extending the framework to deal with polymorphic types at its foundations. Another interesting extension could be considered for the module system presented in Section 6, where only union, deletion and two simple derived operations are defined; from the point of view of the implementation of a modular system, it would be useful to have additional operations to model usual features in this kind of systems, such as those ones for making visible or hiding some function definitions or signatures of modules and specific import and export operations, following the lines of [65, 66].

Also, program modules are of special interest in the context of constraint domains because they allow to represent simpler pure constraint domains $\mathcal{D}$ as modules and a suitable representation of *hybrid constraint domains* $\mathcal{C}$ [76] by means of operations between modules. This modular design supports the cooperation and coordination of predefined $\mathcal{D}$-modules, so more declarative and efficient solutions for practical problems can be promoted. So that, an extension of this framework to parameterize it by a constraint domain could be at the basis of the development of new techniques for the sound and efficient integration of several constraint solvers for simple and hybrid domains.

There are also many lines of future work that can start from this work related to the implementation of systems and tools that can benefit from the strong theoretical foundation that we provide. First of all, it would be interesting to have an efficient system for higher-order functional logic programming with λ-abstractions, where the main issues would be the definition of an adequate operational semantics, that could be obtained following the ideas of [32] and the efficient implementation of the λ-calculus, where a good starting point can be similar to other ones in the context of logic programming with λ-abstractions such as [28].

Other field where this framework can be applied is in the verification and debugging of pro-

grams. For example, following the ideas from [77], it is possible to represent the instantiation of our proof calculus *GHRC* (see Section 3) to a particular *CPRS*-program as an equivalent *higher-order logic theory* suitable to the *Isabelle theorem prover* [34]. With this theory, it is possible to certify the validity of *GHRC*-proofs within this theorem prover, but also to prove more complex properties about programs represented in *Higher-Order Logic* (*HOL*). The *HOL*-theory obtained by this procedure would allow not only to generate certificates for reduction and equality statements, but also for more expressive and interesting properties on declarative programs. It is a matter of undoubtable interest the formalization of a translation from *CPRS*-programs to equivalent higher-order logic theories, and the efficient implementation of such transformation; also, it would be very interesting to have access to automated methods to generate certificates in an automatic or semiautomatic way of properties about particular *CPRS*-programs.

References

[1] P. Hudak, Conception, evolution and application of functional programming languages, ACM Computing Surveys 21 (1989) 359–411.

[2] C. Reade, Elements of functional programming, Addison-Wesley, 1989.

[3] K. R. Apt, Handbook of theoretical computer science, Elsevier Science Publishers, 1990, Ch. 10, pp. 495–574.

[4] J. W. Lloyd, Foundations of logic programming, 2nd Edition, Springer-Verlag, 1987.

[5] M. H. A. Newman, On theories with a combinatorial definition of "equivalence", Annals of Mathematics 43 (2) (1942) 223–243.

[6] J. A. Robinson, A machine-oriented logic based on the resolution principle, Journal of the ACM 12 (1965) 23–41.

[7] M. Hanus, The integration of functions into logic programming: From theory to practice, Journal of Logic Programming 19&20 (1994) 583–628.

[8] M. Rodríguez Artalejo, Functional and constraint logic programming, in: Constraints in Computational Logics, Springer-Verlag, 2001, pp. 202–270.

[9] S. Antoy, M. Hanus, Functional logic programming, in: Communications of the ACM, Vol. 53, 2010, pp. 74–85.

[10] M. Hanus, Multi-paradigm declarative languages, in: Proceeding of the 23rd International Conference on Logic Programming (ICLP 2007), Vol. 4670 of LNCS, 2007, pp. 45–75.

[11] M. Hanus, A unified computation model for functional and logic programming, in: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'97), 1997.

[12] R. del Vado Vírseda, A demand-driven narrowing calculus with overlapping definitional trees, in: Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 2003, pp. 253–263.

[13] S. Antoy, R. Echahed, M. Hanus, A needed narrowing strategy, Journal of the ACM (1994) 268–279.

[14] S. Antoy, Programming with narrowing: a tutorial, Journal of Symbolic Computation 5 (2010) 501–522.

[15] D. S. Lankford, Canonical inference, Tech. Rep. ATP-32, Department of Mathematics and Computer Science, University of Texas at Austin (1975).

[16] M. J. Fay, First-order unification in an equational theory, in: Proceedings of the Workshop on Automated Deduction (CADE'79), 1979, pp. 161–177.

[17] J. M. Hullot, Canonical forms and unification, in: Proceeedings of the 5th Conference on Automated Deduction (CADE'80), 1980, pp. 318–334.

[18] J. R. Slagle, Automated theorem-proving for theories with simplifiers commutativity, and associativity, Journal of the ACM 21 (4) (1974) 622–642.

[19] U. S. Reddy, Narrowing as the operational semantics of functional languages., in: Proceedings of the IEEE International Symposium on Logic in Computer Science (SDP'85), 1985, pp. 138–151.

[20] S. Wolfram, Mathematica: a system for doing mathematics by computer, Addison-Wesley Longman Publishing Co., 1988.

[21] C. Hankin, An introduction to Lambda Calculi for Computer Scientists, King's College Publications, 2004.

[22] J. R. Hindley, J. P. Seldin, Lambda-Calculus and Combinators, an introduction., Cambridge University Press, 2008.

[23] D. H. Warren, Higher-order extensions to prolog: are they needed?, Machine Intelligence 10 (1982) 441–454.

[24] L. Naish, Higher-order logic programming in Prolog, Tech. Rep. 96/2, University of Melbourne (1995).

[25] D. Miller, A logic programming language with lambda-abstraction, function variables, and simple unification, in: Proceedings of the International Workshop on Extensions of Logic Programming, New York, NY, USA, 1991, pp.

253–281.
[26] D. Miller, G. Nadathur, Higher-order logic programming, in: Third International Conference on Logic Programming (ICLP'86), 1986, pp. 448–462.
[27] J. Lipton, S. Nieva, Higher-order logic programming languages with constraints: A semantics., in: Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications (TLCA 2007), 2007, pp. 272–289.
[28] X. Qi, An implementation of the language lambda prolog organized around higher-order pattern unification, Ph.D. thesis, University of Minnesota (2009).
[29] A. Casas, D. Cabeza, M. V. Hermenegildo, A syntactic approach to combining functional notation, lazy evaluation, and higher-order in lp systems., in: Proceedings of the 8th International Symposium on Functional and Logic Programming (FLOPS 2006), 2006, pp. 146–162.
[30] C. Prehofer, Solving higher-order equations. From logic to programming, Birkhäuser, 1999.
[31] J. C. González Moreno, M. T. Hortalá González, F. J. López Fraguas, M. Rodríguez Artalejo, An approach to declarative programming based on a rewriting logic, The journal of logic programming 40 (1999) 47–87.
[32] R. del Vado Vírseda, A higher-order demand-driven narrowing calculus with definitional trees, in: Theoretical Aspects of Computing - ICTAC 2007, 4th International Colloquium, September 26-28, 2007, Proceedings, 2007, pp. 169–184.
[33] R. del Vado Vírseda, A higher order logical framework for the algorithmic debugging and verification of declarative programs, in: Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, pp. 49–60.
[34] L. C. Paulson, Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow), Vol. 828 of Lecture Notes in Computer Science, Springer, 1994.
[35] G. Huet, G. Kahn, C. Paulin-Mohring, The Coq proof assistant. A tutorial (April 2011).
[36] R. del Vado Vírseda, Cooperation of algebraic constraint domains, in: M. Johnoson, D. Pavlovic (Eds.), Proceedings of the Thirteenth International Conference on Algebraic Methodology And Software Technology (AMAST 2010), Vol. 6486 of LNCS, 2010, pp. 180–200.
[37] R. del Vado Vírseda, F. Pérez Morente, A modular semantics for higher-order declarative programming with constraints, in: Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'11), July 20-22, Odense, Denmark, 2011, pp. 41–52.
[38] D. Kranzlmüller, C. Schaubschläger, M. Scarpa, J. Volkert, A modular debugging infraestructure for parallel programs, in: Proceedings of the International Conference on Parallel Computing: Software Technology, Algorithms, Architectures & Applications ParCo 2003, Vol. 13, 2004, pp. 143–150.
[39] P. Mancarella, D. Pedreschi, An algebra of logic programs, in: Proceedings of the Fifth International Conference and Symposium on Logic Programming (SLP'88), 1988, pp. 1006–1023.
[40] F. Logozzo, Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of java classes, in: Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2007), Vol. LNCS, 2007, pp. 283–298.
[41] T. R. Chuang, J. L. Lin, On modular transformation of structural content, in: Proceedings of the 2004 ACM symposium on Document engineering, ACM, 2004, pp. 201–210.
[42] M. Bezem, J. Willem-Klop, R. de Vrijer, Term Rewriting Systems, Cambridge University Press, 2003.
[43] H. P. Barengredt, The Lambda Calculus. Its syntax and semantics., revised Edition, Elsevier, 1984.
[44] G. Dowek, Handbook of automated reasoning, volume 2, Elsevier Science Publishers, 2001, Ch. 16, pp. 1009–1062.
[45] B. C. Pierce, Types and programming languages, The MIT Press, 2002.
[46] F. Baader, T. Nipkow, Term rewriting and all that, Cambridge University Press, New York, NY, USA, 1998.
[47] J. W. Klop, Term rewriting systems - Handbook of logic in computer science, Vol. 2, Oxford University Press, 1992, Ch. 1, pp. 1–116.
[48] W. Goldfarb, The undecidability of the second-order unification problem, Theoretical Computer Science 13 (1981) 225–230.
[49] R. del Vado Vírseda, Estrategias de estrechamiento perezoso, Master's thesis, Universidad Complutense de Madrid, Editorial Académica Española (1999).
[50] R. del Vado Vírseda, I. Castiñeiras Pérez, A theoretical framework for the declarative debugging of functional logic programs with lambda abstractions, in: Proceedings of the 18th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP2009), 2009, pp. 162–178.
[51] R. Caballero, F. J. López Fraguas, M. Rodríguez Artalejo, A logical framework for the algorithmic debugging of lazy functional-logic programs, in: Proceedings of the 9th International Workshop on Functional and Logic Programming (WFLP'00), 2000, pp. 8–22.
[52] R. Caballero, M. Rodríguez Artalejo, A declarative debugging system for lazy functional logic programs, in: Proceedings of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001), Vol. 64 of Electronic Notes in Theoretical Computer Science, 2001.

[53] R. Caballero, M. Rodríguez Artalejo, DDT: a declarative debugging tool for functional-logic languages., in: Proceedings of the Functional and Logic Programming, 7th International Symposium (FLOPS'04), 2004, pp. 70–84.
[54] M. Hanus, C. Prehofer, Higher-order narrowing with definitional trees, Journal of Functional Programming 9 (1) (1999) 33–75.
[55] A. J. Gill, Cheap deforestation for non-strict functional languages, Master's thesis, University of Glasgow (1996).
[56] M. Falaschi, G. Levi, M. Martelli, C. Palamidessi, Declarative modeling of the operational behaviour of logic languages, Theoretical Computer Science 69 (1989) 289–318.
[57] M. Falaschi, G. Levi, M. Martelli, C. Palamidessi, A model-theoretic reconstruction of the operational semantics of logic programs, Information and Computation 102 (1) (1993) 86–113.
[58] D. S. Scott, Domains for denotational semantics, in: Proceedings of the International Conference on Automata, Languages and Programming (ICALP'82), 1982, pp. 577–613.
[59] D. S. Scott, Continuous lattices, in: F. Lawvere (Ed.), Toposes, Algebraic Geometry and Logic, no. 274 in Lecture Notes in Mathematics, Berlin, 1972, pp. 97–136.
[60] B. Möller, On the algebraic specification of infinite objects - ordered and continuous models of algebraic types, Acta Informatica 22 (1985) 537–578.
[61] S. Abramsky, A. Jung, Handbook of Logic in Computer Science, Vol. 3, Clarendon Press, Oxford, 1994, Ch. Domain Theory, pp. 1–168.
[62] M. Fitting, Fixpoint semanticss for logic programming. a survey, Theoretical Computer Science 278 (2002) 25–51.
[63] F. J. López Fraguas, J. Sánchez Hernández, TOY: A multiparadigm declarative system, in: Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA'99), 1999, pp. 244–247.
[64] A. J. Fernandez Leiva, T. Hortalá González, F. Sáenz Pérez, R. del Vado Vírseda, Constraint functional logic programming over finite domains, Journal of Theory and Practice of Logic Programming (TPLP) 7 (2007) 537–582.
[65] A. Brogi, P. Mancarella, D. Pedreschi, F. Turini, Modular logic programming, ACM Transactions on Programming Languages and Systems (TOPLAS'94) 16 (1994) 1361–1398.
[66] A. Brogi, F. Turini, Fully abstract composition semantics for an algebra of logic programs, Theoretical Computer Science 149 (1995) 201–229.
[67] P. Pietrzak, J. Correas, M. Hermenegildo, G. Puebla, A practical type analysis for verification of modular prolog programs, in: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM'08), 2008, pp. 61–70.
[68] J. M. Molina Bravo, E. Pimentel, Composing programs in a rewritting logic for declarative programming, Theory and Practice of Logic Programming (TPLP) 3 (2003) 189–221.
[69] J. C. González Moreno, M. T. Hortalá González, M. Rodríguez Artalejo, A higher order rewriting logic for functional logic programming., in: Proceedings of the Fourteenth International Conference on Logic Programming (ICLP'97), 1997, pp. 153–167.
[70] F. J. López Fraguas, J. Rodríguez Hortalá, J. Sánchez Hernández, Rewriting and call-time choice: the HO case., in: Proceedings of the 17th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP'08), 2008, pp. 147–162.
[71] F. J. López Fraguas, J. Rodríguez Hortalá, The full abstraction problem for higher order functional-logic programs., in: Proceedings of The 19th Workshop on Logic-based methods in Programming Environments (WLPE'09), 2009.
[72] M. Hamada, Strong completeness of a narrowing calculus for conditional rewrite systems with extra variables., in: Proceedings of Computing: the Australasian Theory Symposium (CATS 2000), 2000, pp. 89–103.
[73] M. Marin, Functional logic programming with distributed constraint solving, Ph.D. thesis, Universität Linz (2000).
[74] J. C. González Moreno, M. T. Hortalá González, M. Rodríguez Artalejo, Polymorphic types in functional logic programming, Journal of Functional and Logic Programming 1 (2001) 1–71.
[75] F. López Fraguas, E. Martin Martin, J. Rodriguez Hortala, Lyberal typing for functional logic programs, in: Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS'10), Vol. 6461 of Lecture Notes in Computer Science, 2010, pp. 80–96.
[76] S. Estévez, T. Hortalá, M. Rodríguez, R. del Vado Vírseda, F. Sáenz, A. Fernández, On the cooperation of constraint domains H, R and FD in CFLP, Theory and Practice of Logic Programming (TPLP) 9 (4) (2009) 415–527.
[77] J. M. Cleva, J. Leach, F. J. López Fraguas, A logic programming approach to the verification of functional-logic programs, in: Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'04), 2004, pp. 9–19.

Appendix (Proofs omitted from the paper).

Theorem 4.8 (Adequateness of $\mathcal{M}_\mathcal{R}$). *Let $\mathcal{R}$ be a CPRS. The pattern model $\mathcal{M}_\mathcal{R}$ is a model of $\mathcal{R}$. Moreover, for any reduction or equality statement φ, the following conditions are equivalent:*

1) $\mathcal{R} \vdash \varphi$.
2) $\mathcal{A} \vDash_\eta \varphi$, for every $\mathcal{A} \vDash \mathcal{R}$ and every totally defined valuation η.
3) $\mathcal{M}_\mathcal{R} \vDash_\varepsilon \varphi$, where ε is the identity valuation.

Proof To prove $\mathcal{M}_\mathcal{R} \vDash \mathcal{R}$, we consider a pattern rewrite rule $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$ and a value substitution $\theta \in \mathit{VSubst}(\Sigma_\perp, \mathcal{V})$ over $\mathcal{M}_\mathcal{R}$.
Assume that $\mathcal{M}_\mathcal{R} \vDash_\theta C$. Then, $\mathcal{R} \vdash C\theta$ by applying Definition 4.7 (see the proof of the implication 3) $\Rightarrow$ 1) below for more details).
It follows that $\mathcal{R} \vdash (f(\overline{l_n}) \twoheadrightarrow r)\theta$, or equivalently, $\mathcal{R} \vdash f(\overline{l_n})\theta \twoheadrightarrow r\theta$, by applying the rule **OR** of the *GHRC* calculus (i.e.,

$$\frac{\lambda\overline{x_k}.\, s_1 \twoheadrightarrow l_1^{\uparrow\overline{x_k}}\theta \cdots \lambda\overline{x_k}.\, s_n \twoheadrightarrow l_n^{\uparrow\overline{x_k}}\theta \quad C^{\uparrow\overline{x_k}}\theta \quad r^{\uparrow\overline{x_k}}\theta \twoheadrightarrow u}{\lambda\overline{x_k}.\, f(\overline{s_n}) \twoheadrightarrow u}\ (\mathbf{OR})$$

because $\mathcal{R} \vdash r\theta \twoheadrightarrow r\theta$ trivially holds by **RF**).
We conclude that $\{t \in \mathit{Val}(\Sigma_\perp, \mathcal{V}) \mid \mathcal{R} \vdash r\theta \twoheadrightarrow t\} \subseteq \{t \in \mathit{Val}(\Sigma_\perp, \mathcal{V}) \mid \mathcal{R} \vdash f(\overline{l_n})\theta \twoheadrightarrow t\}$. This means:

$$\begin{array}{lcl}
[\![\, r \,]\!]_\theta^{\mathcal{M}_\mathcal{R}} & = & [\![\, r\theta \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}} \\
& = & \{t \in \mathit{Val}(\Sigma_\perp, \mathcal{V}) \mid t \in [\![\, r\theta \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}}\} \\
& = & \{t \in \mathit{Val}(\Sigma_\perp, \mathcal{V}) \mid \langle\, t \,\rangle = [\![\, t \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}} \subseteq [\![\, r\theta \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}}\} \\
& = & \{t \in \mathit{Val}(\Sigma_\perp, \mathcal{V}) \mid \mathcal{M}_\mathcal{R} \vDash_\varepsilon (r\theta \twoheadrightarrow t)\} \\
& \subseteq & \{t \in \mathit{Val}(\Sigma_\perp, \mathcal{V}) \mid \mathcal{R} \vdash r\theta \twoheadrightarrow t\} \\
& \subseteq & \{t \in \mathit{Val}(\Sigma_\perp, \mathcal{V}) \mid \mathcal{R} \vdash f(\overline{l_n})\theta \twoheadrightarrow t\} \\
& = & [\![\, f(\overline{l_n}) \,]\!]_\theta^{\mathcal{M}_\mathcal{R}}
\end{array}$$

by applying again Definition 4.7 (and again the proof of the implication 3) $\Rightarrow$ 1)).
Therefore, $\mathcal{M}_\mathcal{R} \vDash_\theta (f(\overline{l_n}) \twoheadrightarrow r)$, and we can conclude that $\mathcal{M}_\mathcal{R} \vDash (f(\overline{l_n}) \to r \Leftarrow C)$. By Definition 4.6, $\mathcal{M}_\mathcal{R}$ is a *GHRC*-model of $\mathcal{R}$ (i.e., $\mathcal{M}_\mathcal{R} \vDash \mathcal{R}$).

Now we prove the equivalence of 1), 2) and 3) by proving that 1) $\Rightarrow$ 2), 2) $\Rightarrow$ 3) and 3) $\Rightarrow$ 1):

1) $\Rightarrow$ 2) Assume that $\mathcal{R} \vdash \varphi$ for any reduction or equality statement φ. Let $\mathcal{A} \vDash \mathcal{R}$ be an arbitrarily fixed totally defined valuation η. Now, we prove that $\mathcal{A} \vDash_\eta \varphi$ by induction on the length of the *GHRC*-proof:

B If $\mathcal{R} \vdash \lambda\overline{x_k}.\pi \twoheadrightarrow \lambda\overline{x_k}.\, \perp$ then $[\![\, \lambda\overline{x_k}.\, \perp \,]\!]_\eta^\mathcal{A} = \langle\, \perp_\mathcal{A} \,\rangle = \{d \in D_\mathcal{A} \mid d \sqsubseteq_{D_\mathcal{A}} \perp_\mathcal{A}\} = \{\perp_\mathcal{A}\}$.
Since $[\![\, \lambda\overline{x_k}.\pi \,]\!]_\eta^\mathcal{A}$ is a downward closed subset of $D_\mathcal{A}$, $\perp_\mathcal{A} \in [\![\, \lambda\overline{x_k}.\pi \,]\!]_\eta^\mathcal{A}$ and $[\![\, \lambda\overline{x_k}.\pi \,]\!]_\eta^\mathcal{A} \supseteq [\![\, \lambda\overline{x_k}.\, \perp \,]\!]_\eta^\mathcal{A}$.
By Definition 4.6, we conclude that $\mathcal{A} \vDash_\eta (\lambda\overline{x_k}.\pi \twoheadrightarrow \lambda\overline{x_k}.\perp)$.

MN If $\mathcal{R} \vdash \lambda\overline{x_k}.a(\overline{s_n}) \twoheadrightarrow \lambda\overline{x_k}.a(\overline{t_n})$, we can assume that $\mathcal{A} \vDash_\eta (\lambda\overline{x_k}.s_i \twoheadrightarrow \lambda\overline{x_k}.t_i)$, for $i = 1, \ldots, n$, by induction hypothesis, where $\{\overline{x_k \mapsto d_k}\} \in \eta$ with $d_i \in D_\mathcal{A}$ arbitrarily fixed.

By Definition 4.6, $[\![s_i^{\updownarrow\overline{x_k}}]\!]_\eta^\mathcal{A} \supseteq [\![t_i^{\updownarrow\overline{x_k}}]\!]_\eta^\mathcal{A}$. We prove that $[\![\lambda\overline{x_k}.a(\overline{s_n})]\!]_\eta^\mathcal{A} \supseteq [\![\lambda\overline{x_k}.a(\overline{t_n})]\!]_\eta^\mathcal{A}$.

Since:

$$\begin{array}{lcl} [\![\lambda\overline{x_k}.a(\overline{s_n})]\!]_\eta^\mathcal{A} & = & (\cdots([\![a]\!]_\eta^\mathcal{A} \circ^\mathcal{A} [\![s_1^{\updownarrow\overline{x_k}}]\!]_\eta^\mathcal{A}) \circ^\mathcal{A} \cdots \circ^\mathcal{A} [\![s_n^{\updownarrow\overline{x_k}}]\!]_\eta^\mathcal{A}) \\ & \supseteq & (\cdots([\![a]\!]_\eta^\mathcal{A} \circ^\mathcal{A} [\![t_1^{\updownarrow\overline{x_k}}]\!]_\eta^\mathcal{A}) \circ^\mathcal{A} \cdots \circ^\mathcal{A} [\![t_n^{\updownarrow\overline{x_k}}]\!]_\eta^\mathcal{A}) \\ & = & [\![\lambda\overline{x_k}.a(\overline{t_n})]\!]_\eta^\mathcal{A} \end{array}$$

we conclude that $\mathcal{A} \vDash_\eta (\lambda\overline{x_k}.a(\overline{s_n}) \twoheadrightarrow \lambda\overline{x_k}.a(\overline{t_n}))$.

RF If $\mathcal{R} \vdash s \twoheadrightarrow s$, trivially $[\![s]\!]_\eta^\mathcal{A} \supseteq [\![s]\!]_\eta^\mathcal{A}$, and $\mathcal{A} \vDash_\eta (s \twoheadrightarrow s)$ by Definition 4.6.

OR If $\mathcal{R} \vdash \lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow u$, where $f \in \Sigma_d$ and $u \neq \lambda\overline{x_k}.\bot$, we can consider $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$, $\theta \in VSubst(\Sigma_\bot, \mathcal{V})$, and $\{\overline{x_k \mapsto d_k}\} \in \eta$ with $d_i \in D_\mathcal{A}$ arbitrarily fixed.

Assume that $\mathcal{A} \vDash_\eta C^{\updownarrow\overline{x_k}}\theta$. Then, with $\rho = \eta\theta$, $\mathcal{A} \vDash_\rho C^{\updownarrow\overline{x_k}}$.

Since $\mathcal{A} \vDash \mathcal{R}$ by hypothesis, we conclude that $\mathcal{A} \vDash_\rho (f(\overline{l_n^{\updownarrow\overline{x_k}}}) \twoheadrightarrow r^{\updownarrow\overline{x_k}})$ by Definition 4.6, we come to $\mathcal{A} \vDash_\eta (f(\overline{l_n^{\updownarrow\overline{x_k}}}) \twoheadrightarrow r)\theta$, or equivalently, $[\![f(\overline{l_n^{\updownarrow\overline{x_k}}\theta})]\!]_\eta^\mathcal{A} \supseteq [\![r^{\updownarrow\overline{x_k}}\theta]\!]_\eta^\mathcal{A}$.

By induction hypothesis, $[\![s_i^{\updownarrow\overline{x_k}}]\!]_\eta^\mathcal{A} \supseteq [\![l_i^{\updownarrow\overline{x_k}}\theta]\!]_\eta^\mathcal{A}$, for $i = 1, \ldots, n$, and $[\![r^{\updownarrow\overline{x_k}}\theta]\!]_\eta^\mathcal{A} \supseteq [\![u]\!]_\eta^\mathcal{A}$. We prove that $[\![f(\overline{s_n})^{\updownarrow\overline{x_k}}]\!]_\eta^\mathcal{A} \supseteq [\![u]\!]_\eta^\mathcal{A}$.

Since:

$$\begin{array}{lcl} [\![f(\overline{s_n})^{\updownarrow\overline{x_k}}]\!]_\eta^\mathcal{A} & = & (\cdots([\![f]\!]_\eta^\mathcal{A} \circ^\mathcal{A} [\![s_1^{\updownarrow\overline{x_k}}]\!]_\eta^\mathcal{A}) \circ^\mathcal{A} \cdots \circ^\mathcal{A} [\![s_n^{\updownarrow\overline{x_k}}]\!]_\eta^\mathcal{A}) \\ & \supseteq & (\cdots([\![f]\!]_\eta^\mathcal{A} \circ^\mathcal{A} [\![l_1^{\updownarrow\overline{x_k}}\theta]\!]_\eta^\mathcal{A}) \circ^\mathcal{A} \cdots \circ^\mathcal{A} [\![l_n^{\updownarrow\overline{x_k}}\theta]\!]_\eta^\mathcal{A}) \\ & = & [\![f(\overline{l_n^{\updownarrow\overline{x_k}}\theta})]\!]_\eta^\mathcal{A} \\ & \supseteq & [\![r^{\updownarrow\overline{x_k}}\theta]\!]_\eta^\mathcal{A} \\ & \supseteq & [\![u]\!]_\eta^\mathcal{A} \end{array}$$

we conclude $\mathcal{A} \vDash_\eta (\lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow u)$.

J If $\mathcal{R} \vdash s == t$, we can assume that $\mathcal{A} \vDash_\eta (s \twoheadrightarrow u)$ and $\mathcal{A} \vDash_\eta (t \twoheadrightarrow u)$ for some $u \in Val(\Sigma, \mathcal{V})$, by induction hypothesis. By Definition 4.6, this means $[\![s]\!]_\eta^\mathcal{A} \supseteq [\![u]\!]_\eta^\mathcal{A}$ and $[\![t]\!]_\eta^\mathcal{A} \supseteq [\![u]\!]_\eta^\mathcal{A}$.

We know that $[\![u]\!]_\eta^\mathcal{A} = \langle d \rangle$ for some maximal element $d \in D_\mathcal{A}$. Again, by Definition 4.6, we can conclude that $[\![s]\!]_\eta^\mathcal{A} \cap [\![s]\!]_\eta^\mathcal{A}$ contains a maximal element $d \in D_\mathcal{A}$, and $\mathcal{A} \vDash_\eta (s == t)$.

2) $\Rightarrow$ 3) Trivially, because $\mathcal{M}_\mathcal{R}$ is a *GHRC*-model of $\mathcal{R}$ and ε is a totally defined valuation.

3) $\Rightarrow$ 1) Let ε be the identity valuation over $\mathcal{M}_\mathcal{R}$. For any approximation or equality statement φ, we prove that if $\mathcal{M}_\mathcal{R} \vDash_\varepsilon \varphi$ then $\mathcal{R} \vdash \varphi$. We reason by induction on the size of φ, defined as the number of symbols occurring in φ.
Since $\mathcal{M}_\mathcal{R} \vDash_\varepsilon \varphi$, there are only five cases to consider:

1. If $\varphi \equiv \lambda\overline{x_k}.\pi \twoheadrightarrow \lambda\overline{x_k}.\ \bot$ then $\mathcal{R} \vdash \lambda\overline{x_k}.\pi \twoheadrightarrow \lambda\overline{x_k}.\ \bot$ holds because of rule **B**.

2. If $\varphi \equiv s \twoheadrightarrow s$ then $\mathcal{R} \vdash s \twoheadrightarrow s$ holds because of rule **RF**.

3. If $\varphi \equiv \lambda\overline{x_k}.\, a(\overline{s_n}) \twoheadrightarrow \lambda\overline{x_k}.\, a(\overline{t_n})$, by construction of $\mathcal{M}_\mathcal{R}$ we have that $\mathcal{M}_\mathcal{R} \models_\varepsilon \varphi$ entails $\mathcal{M}_\mathcal{R} \models_\varepsilon (\lambda\overline{x_k}.s_i \twoheadrightarrow \lambda\overline{x_k}.t_i)$, for $i = 1, \ldots, n$.
Then, by induction hypothesis, $\mathcal{R} \vdash \lambda\overline{x_k}.s_i \twoheadrightarrow \lambda\overline{x_k}.t_i$ and $\frac{\lambda\overline{x_k}.s_1 \twoheadrightarrow \lambda\overline{x_k}.t_1 \cdots \lambda\overline{x_k}.s_n \twoheadrightarrow \lambda\overline{x_k}.t_n}{\lambda\overline{x_k}.a(\overline{s_n}) \twoheadrightarrow \lambda\overline{x_k}.a(\overline{t_n})}$(**MN**).

4. If $\varphi \equiv \lambda\overline{x_k}.\, f(\overline{s_n}) \twoheadrightarrow t$ with $t \neq \lambda\overline{x_k}.\bot$, then $[\![\, t \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}} = \langle\, t\varepsilon \,\rangle = \langle\, t \,\rangle$ and $t \in (\cdots (f^{\mathcal{M}_\mathcal{R}} \circ^{\mathcal{M}_\mathcal{R}} [\![\, s_1^{\uparrow\overline{x_k}} \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}}) \circ^{\mathcal{M}_\mathcal{R}} \cdots \circ^{\mathcal{M}_\mathcal{R}} [\![\, s_n^{\uparrow\overline{x_k}} \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}})$.
Hence, there are some $t_i \in [\![\, s_i^{\uparrow\overline{x_k}} \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}}$, for $i = 1, \ldots, n$, such that $t \in f^{\mathcal{M}_\mathcal{R}}(\overline{t_n})$. Therefore, we have $\mathcal{M}_\mathcal{R} \models_\varepsilon (\lambda\overline{x_k}.s_i \twoheadrightarrow t_i)$ and $\mathcal{R} \vdash f(\overline{t_n}) \twoheadrightarrow t$ by construction of $\mathcal{M}_\mathcal{R}$ (see Definition 4.7).
By induction hypothesis, we can assume $\mathcal{R} \vdash \lambda\overline{x_k}.s_i \twoheadrightarrow t_i$, for $i = 1, \ldots, n$. Then, $\frac{\lambda\overline{x_k}.s_1 \twoheadrightarrow t_1 \cdots \lambda\overline{x_k}.s_n \twoheadrightarrow t_n}{\lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow f(\overline{t_n})}$(**MN**) and $\mathcal{R} \vdash \lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow f(\overline{t_n})$. Since $\mathcal{R} \vdash f(\overline{t_n}) \twoheadrightarrow t$, we conclude that $\mathcal{R} \vdash \lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow t$.

5. If $\varphi \equiv s == t$, we get that $\mathcal{M}_\mathcal{R} \models_\varepsilon (s == t)$ entails the existence of a maximal element $u \in \mathit{Val}(\Sigma_\bot, \mathcal{V})$ such that $u \in [\![\, s \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}} \cap [\![\, t \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}}$.
Since maximal elements in $\mathcal{M}_\mathcal{R}$ are totally defined elements in $\mathit{Val}(\Sigma_\bot, \mathcal{V})$, we have that u is a total value (i.e., $u \in \mathit{Val}(\Sigma_\bot, \mathcal{V})$). Then, $u \in [\![\, s \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}}$ and $u \in [\![\, t \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}}$.
Due to the fact that $[\![\, u \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}} = \langle\, u\varepsilon \,\rangle = \langle\, u \,\rangle$, we can deduce that $\mathcal{M}_\mathcal{R} \models_\varepsilon (s \twoheadrightarrow u)$ and $\mathcal{M}_\mathcal{R} \models_\varepsilon (t \twoheadrightarrow u)$.
By induction hypothesis, $\mathcal{R} \vdash s \twoheadrightarrow u$ and $\mathcal{R} \vdash t \twoheadrightarrow u$ for $u \in \mathit{Val}(\Sigma_\bot, \mathcal{V})$, and then $\frac{s \twoheadrightarrow u \quad t \twoheadrightarrow u}{s == t}$(**J**). We can conclude that $\mathcal{R} \vdash \varphi$.

□

Lemma 5.4 (Continuity of valuations in Alg_Σ). *For each partial term $t \in T(\Sigma_\bot, \mathcal{V})$ and each value substitution $\theta \in \mathit{VSubst}(\Sigma_\bot, \mathcal{V})$:*

1. *If $\mathcal{A} \sqsubseteq \mathcal{B}$ then $[\![\, t \,]\!]_\theta^\mathcal{A} \subseteq [\![\, t \,]\!]_\theta^\mathcal{B}$, for $\mathcal{A}, \mathcal{B} \in Alg_\Sigma$.*
2. *$[\![\, t \,]\!]_\theta^{\sqcup D} = \bigcup_{\mathcal{A} \in D} [\![\, t \,]\!]_\theta^\mathcal{A}$, for all directed subsets $D \subseteq Alg_\Sigma$.*

Proof

1. The first statement is proved by induction on the structure of t:
   - If $t \in \{\bot\} \cup \mathcal{V}$ or $t \in \Sigma_c$ with $\mathit{arity}(t) = 0$ then $[\![\, t \,]\!]_\theta^\mathcal{A}$ does not depend on the particular pattern algebra $\mathcal{A}$ and $[\![\, t \,]\!]_\theta^\mathcal{A} = [\![\, t \,]\!]_\theta^\mathcal{B}$.
   - If $t \in \Sigma_d$ with $\mathit{arity}(t) = 0$, $\mathcal{A} \sqsubseteq \mathcal{B}$ implies $t^\mathcal{A} \subseteq t^\mathcal{B}$ and then $[\![\, t \,]\!]_\theta^\mathcal{A} \subseteq [\![\, t \,]\!]_\theta^\mathcal{B}$.
   - If $t = f(\overline{t_n})$ with $f \in \Sigma_d$ and $\mathit{arity}(f) = n > 0$, assuming $[\![\, t_i \,]\!]_\theta^\mathcal{A} \subseteq [\![\, t_i \,]\!]_\theta^\mathcal{B}$, for $i = 1, \ldots, n$, as the induction hypothesis, for every $s \in [\![\, t \,]\!]_\theta^\mathcal{A}$ we have $s \in f^\mathcal{A}(\overline{s_n})$ for some $s_i \in [\![\, t_i \,]\!]_\theta^\mathcal{A}$, which implies $s \in f^\mathcal{B}(\overline{s_n})$ with $s_i \in [\![\, t_i \,]\!]_\theta^\mathcal{B}$ as a consequence of $\mathcal{A} \sqsubseteq \mathcal{B}$ and the induction hypothesis. Thus, we get $s \in [\![\, t \,]\!]_\theta^\mathcal{B}$, and consequently $[\![\, t \,]\!]_\theta^\mathcal{A} \subseteq [\![\, t \,]\!]_\theta^\mathcal{B}$.
   - If $t = (t_1\ t_2)$, assuming $[\![\, t_i \,]\!]_\theta^\mathcal{A} \subseteq [\![\, t_i \,]\!]_\theta^\mathcal{B}$ for $i = 1, 2$, as the induction hypothesis, for every $s \in [\![\, t \,]\!]_\theta^\mathcal{A} = [\![\, (t_1\ t_2) \,]\!]_\theta^\mathcal{A} = [\![\, t_1 \,]\!]_\theta^\mathcal{A} \circ^\mathcal{A} [\![\, t_2 \,]\!]_\theta^\mathcal{A} \subseteq [\![\, t_1 \,]\!]_\theta^\mathcal{B} \circ^\mathcal{B} [\![\, t_2 \,]\!]_\theta^\mathcal{B} = [\![\, t \,]\!]_\theta^\mathcal{B}$ as a consequence of $\mathcal{A} \sqsubseteq \mathcal{B}$ and the induction hypothesis. Thus, we get $s \in [\![\, t \,]\!]_\theta^\mathcal{B}$, and consequently $[\![\, t \,]\!]_\theta^\mathcal{A} \subseteq [\![\, t \,]\!]_\theta^\mathcal{B}$.

- If $t = \lambda\overline{x_k}.a(\overline{t_n})$, assuming $[\![a]\!]^{\mathcal{A}}_{\theta} \subseteq [\![a]\!]^{\mathcal{B}}_{\theta}$ and $[\![t_i]\!]^{\mathcal{A}}_{\theta} \subseteq [\![t_i]\!]^{\mathcal{B}}_{\theta}$ for $i = 1, \ldots, n$, as the induction hypothesis, where $\{\overline{x_k \mapsto s_k}\} \in \theta$ with s_i arbitrarily fixed partial values of $D_{\mathcal{A}}$, for every

$$\begin{array}{rcl} s \in [\![t]\!]^{\mathcal{A}}_{\theta} & = & [\![\lambda\overline{x_k}.a(\overline{t_n})]\!]^{\mathcal{A}}_{\theta} \\ & = & [\![(\cdots(a\ t_1)\cdots t_n)]\!]^{\mathcal{A}}_{\theta} \\ & = & (\cdots([\![a]\!]^{\mathcal{A}}_{\theta} \circ^{\mathcal{A}} [\![t_1]\!]^{\mathcal{A}}_{\theta}) \circ^{\mathcal{A}} \cdots \circ^{\mathcal{A}} [\![t_n]\!]^{\mathcal{A}}_{\theta}) \\ & \subseteq & (\cdots([\![a]\!]^{\mathcal{B}}_{\theta} \circ^{\mathcal{B}} [\![t_1]\!]^{\mathcal{B}}_{\theta}) \circ^{\mathcal{B}} \cdots \circ^{\mathcal{B}} [\![t_n]\!]^{\mathcal{B}}_{\theta}) \\ & = & [\![(\cdots(a\ t_1)\cdots t_n)]\!]^{\mathcal{B}}_{\theta} \\ & = & [\![\lambda\overline{x_k}.a(\overline{t_n})]\!]^{\mathcal{A}}_{\theta} \\ & = & [\![t]\!]^{\mathcal{A}}_{\theta} \end{array}$$

  as a consequence of $\mathcal{A} \sqsubseteq \mathcal{B}$ and the induction hypothesis. Thus, we get $s \in [\![t]\!]^{\mathcal{B}}_{\theta}$, and consequently $[\![t]\!]^{\mathcal{A}}_{\theta} \subseteq [\![t]\!]^{\mathcal{B}}_{\theta}$.

2. To prove the second statement we only need to prove the following inclusion: $[\![t]\!]^{\sqcup D}_{\theta} \subseteq \bigcup_{\mathcal{A}\in D}[\![t]\!]^{\mathcal{A}}_{\theta}$, because the inclusion in the other way is trivially derived from the first statement. We also proceed by induction on t:

   - If $t \in \{\bot\} \cup \mathcal{V}$ or $t \in \Sigma_c$ with $arity(t) = 0$ then, as $[\![t]\!]^{\mathcal{A}}_{\theta}$ does not depend on $\mathcal{A}$, $[\![t]\!]^{\sqcup D}_{\theta} = [\![t]\!]^{\mathcal{A}}_{\theta}$ for all $\mathcal{A} \in D$.
   - If $t \in \Sigma_d$ with $arity(t) = 0$ then $[\![t]\!]^{\sqcup D}_{\theta} = t^{\sqcup D}$ and, by definition, $t^{\sqcup D} = \bigcup_{\mathcal{A}\in D} t^{\mathcal{A}}$. So, in all these cases, $[\![t]\!]^{\sqcup D}_{\theta} = \bigcup_{\mathcal{A}\in D}[\![t]\!]^{\mathcal{A}}_{\theta}$.
   - If $t = f(\overline{t_n})$ with $f \in \Sigma_d$ and $arity(f) = n > 0$, assuming $[\![t_i]\!]^{\sqcup D}_{\theta} \subseteq \bigcup_{\mathcal{A}\in D}[\![t_i]\!]^{\mathcal{A}}_{\theta}$, $i = 1, \ldots, n$, as the induction hypothesis, for every $s \in [\![t]\!]^{\sqcup D}_{\theta}$ we have $s \in f^{\sqcup D}(\overline{s_n})$ for some $s_i \in [\![t_i]\!]^{\sqcup D}_{\theta}$, $i = 1, \ldots, n$.

     By definition, $f^{\sqcup D}(\overline{s_n}) = \bigcup_{\mathcal{A}\in D} f^{\mathcal{A}}(\overline{s_n})$, and from this and the induction hypothesis we can deduce $s \in f^{\mathcal{A}_0}(\overline{s_n})$ with $s_i \in [\![t_i]\!]^{\mathcal{A}_i}_{\theta}$, for some $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_n \in D$.

     Since D is directed, there exists $\mathcal{A} \in D$, such that $\mathcal{A}_i \sqsubseteq \mathcal{A}$, $i = 0, 1, \ldots, n$, and so $s \in f^{\mathcal{A}}(\overline{s_n})$ with $s_i \in [\![t_i]\!]^{\mathcal{A}}_{\theta}$, which implies $s \in [\![t]\!]^{\mathcal{A}}_{\theta}$ and $[\![t]\!]^{\sqcup D}_{\theta} \subseteq \bigcup_{\mathcal{A}\in D}[\![t]\!]^{\mathcal{A}}_{\theta}$.
   - If $t = (t_1\ t_2)$, we know by definition that $\circ^{\sqcup D} = \bigcup_{\mathcal{A}\in D} \circ^{\mathcal{A}}$, and D is directed by initial hypothesis.

     Assuming $[\![t_i]\!]^{\sqcup D}_{\theta} \subseteq \bigcup_{\mathcal{A}\in D}[\![t_i]\!]^{\mathcal{A}}_{\theta}$, $i = 1, 2$, as the induction hypothesis, for every

$$\begin{array}{rcl} s \in [\![t]\!]^{\sqcup D}_{\theta} & = & [\![(t_1\ t_2)]\!]^{\sqcup D}_{\theta} \\ & = & [\![t_1]\!]^{\sqcup D}_{\theta} \circ^{\sqcup D} [\![t_2]\!]^{\sqcup D}_{\theta} \\ & \subseteq & (\bigcup_{\mathcal{A}\in D}[\![t_1]\!]^{\mathcal{A}}_{\theta}) \circ^{\sqcup D} (\bigcup_{\mathcal{A}\in D}[\![t_2]\!]^{\mathcal{A}}_{\theta}) \\ & = & \bigcup_{\mathcal{A}\in D}([\![t_1]\!]^{\mathcal{A}}_{\theta} \circ^{\mathcal{A}} [\![t_2]\!]^{\mathcal{A}}_{\theta}) \\ & = & \bigcup_{\mathcal{A}\in D}[\![(t_1\ t_2)]\!]^{\mathcal{A}}_{\theta} \\ & = & \bigcup_{\mathcal{A}\in D}[\![t]\!]^{\mathcal{A}}_{\theta} \end{array}$$

     which implies $s \in \bigcup_{\mathcal{A}\in D}[\![t]\!]^{\mathcal{A}}_{\theta}$ and $[\![t]\!]^{\sqcup D}_{\theta} \subseteq \bigcup_{\mathcal{A}\in D}[\![t]\!]^{\mathcal{A}}_{\theta}$.
   - If $t = \lambda\overline{x_k}.a(\overline{t_n})$, assuming $[\![a]\!]^{\sqcup D}_{\theta} \subseteq \bigcup_{\mathcal{A}\in D}[\![a]\!]^{\mathcal{A}}_{\theta}$ and $[\![t_i]\!]^{\sqcup D}_{\theta} \subseteq \bigcup_{\mathcal{A}\in D}[\![t_i]\!]^{\mathcal{A}}_{\theta}$, for $i = 1, \ldots, n$, as the induction hypothesis, where $\{\overline{x_k \mapsto s_k}\} \in \theta$ with s_i arbitrarily fixed partial values of $\mathcal{A} \in D$ and D directed, for every

$$
\begin{array}{lcl}
s \in [\![\, t \,]\!]_\theta^{\sqcup D} & = & [\![\, \lambda\overline{x_k}.a(\overline{t_n}) \,]\!]_\theta^{\sqcup D} \\
& = & [\![\, (\cdots (a\ t_1) \cdots t_n) \,]\!]_\theta^{\sqcup D} \\
& = & (\cdots ([\![\, a \,]\!]_\theta^{\sqcup D} \circ^{\sqcup D} [\![\, t_1 \,]\!]_\theta^{\sqcup D}) \circ^{\sqcup D} \cdots \circ^{\sqcup D} [\![\, t_n \,]\!]_\theta^{\sqcup D}) \\
& \subseteq & (\cdots ((\bigcup_{\mathcal{A} \in D} [\![\, a \,]\!]_\theta^{\mathcal{A}}) \circ^{\sqcup D} (\bigcup_{\mathcal{A} \in D} [\![\, t_1 \,]\!]_\theta^{\mathcal{A}})) \circ^{\sqcup D} \cdots \circ^{\sqcup D} (\bigcup_{\mathcal{A} \in D} [\![\, t_n \,]\!]_\theta^{\mathcal{A}})) \\
& = & (\cdots (\bigcup_{\mathcal{A} \in D} ([\![\, a \,]\!]_\theta^{\mathcal{A}} \circ^{\mathcal{A}} [\![\, t_1 \,]\!]_\theta^{\mathcal{A}})) \circ^{\sqcup D} \cdots \circ^{\sqcup D} (\bigcup_{\mathcal{A} \in D} [\![\, t_n \,]\!]_\theta^{\mathcal{A}})) \\
& = & \bigcup_{\mathcal{A} \in D} (\cdots ([\![\, a \,]\!]_\theta^{\mathcal{A}} \circ^{\mathcal{A}} [\![\, t_1 \,]\!]_\theta^{\mathcal{A}}) \circ^{\mathcal{A}} \cdots \circ^{\mathcal{A}} [\![\, t_n \,]\!]_\theta^{\mathcal{A}}) \\
& = & \bigcup_{\mathcal{A} \in D} [\![\, \lambda\overline{x_k}.a(\overline{t_n}) \,]\!]_\theta^{\mathcal{A}} \\
& = & \bigcup_{\mathcal{A} \in D} [\![\, t \,]\!]_\theta^{\mathcal{A}}
\end{array}
$$

which implies $s \in \bigcup_{\mathcal{A} \in D} [\![\, t \,]\!]_\theta^{\mathcal{A}}$ and $[\![\, t \,]\!]_\theta^{\sqcup D} \subseteq \bigcup_{\mathcal{A} \in D} [\![\, t \,]\!]_\theta^{\mathcal{A}}$.

□