Università degli Studi di Udine
Dipartimento di Matematica e Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

An Abstract Interpretation Framework for Semantics and Diagnosis of Lazy Functional-Logic Languages

Candidate:
Giovanni Bacci

Supervisor:
Marco Comini

Academic Year 2010-11

Author's e-mail: giovanni.bacci@uniud.it

Author's address:

Dipartimento di Matematica e Informatica
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine
Italia

Contents

Introduction

Programming languages are divided into different paradigms. The most known and used is the imperative paradigm. Declarative paradigms have born with the aim to ease the task of programming: in contrast to imperative programming, one does not prescribes a sequence of steps to be performed to obtain a solution to a problem, but what are the properties of the problem and the expected solutions. Declarative programming languages provide a higher and more abstract level of programming that leads to reliable and maintainable programs. This paradigm has become of particular interest recently, as it may greatly simplify writing parallel programs.

Historically declarative languages have been separated in two main paradigms: the functional paradigm and the logic paradigm, both of them supporting different features that have been shown to be useful in application programming. Since these worlds of programming are based on common grounds, it is a natural idea to combine them into a single multi-paradigm declarative language. However, the interactions between the different features are complex in detail so that the concrete design of a multi-paradigm declarative language is non-trivial. This is demonstrated by many different proposals and a lot of research work on the semantics, operational principles, and implementation of multi-paradigm declarative languages since more than two decades. The languages Curry [51] and $\mathcal{TOY}$ [16] come through the problem with a single mechanism, called *narrowing*, which amalgamates the functional concept of reduction with unification and nondeterministic search from logic programming. Moreover, if unification on terms is generalized to constraint solving, features of constraint programming are also covered. Based on the narrowing principle, one can define declarative languages integrating the good features of the individual paradigms, in particular, with a sound and complete operational semantics that is optimal for a large class of programs [6].

The operational narrowing principle has a very elegant logical counterpart, the Constru ctor-based ReWriting Logic (CRWL) [49, 50, 64], which provides a logical semantics that helped to better understand the meaning of programs and also to detect issues and flaws in the language definitions (like the call-time versus need-time choice issue).

A (declarative) semantics for a programming language provides meanings for programs or, more generally, program components. Clearly one purpose of a semantics is to help understand the meaning of programs, however this is not the only one. Some successful applications are the semantic-based techniques (such as program analysis, debugging and transformation) which have been used to develop useful programming tools.

When a semantics is to be used for the development of efficacious semantic-based program manipulation tools, one of the most useful features it can have is to be "condensed". Many declarative semantics contain many "semantically useless" elements that can be retrieved from a smaller set of "essential" elements. A semantics is "condensed" if it retains in denotations *only* the "essential" elements. The operational [1, 6] or the rewriting logic semantics [49] that are most commonly considered in functional logic programming are not condensed. Indeed, a "condensed" (goal-independent) semantics for functional logic languages, modeling the actual (implemented) behavior, does not exist.

The main motivation of this thesis is to develop a framework for defining "condensed" semantics which are adequate to model any abstraction (either precise or approximate) of "the computed narrowing trees", to be used as base semantics for the construction of semantic-based program development tools (like program analyzers, debuggers, correctors and verifiers).

In particular this framework should allow us to address problems such as

- the relation between the operational and the denotational semantics
- the existence of a goal-independent denotation for a set of program equations
- the properties of the denotations (e.g. compositionality, correctness and full abstraction)

We will construct such a framework starting from a semantics modeling the "small-step" behavior (by means of what we called small-step trees) and then we will apply abstract interpretation techniques to define observable properties (i.e., behavioral properties of the computation).

We will show then some applications of this framework. Namely,

- groundness dependency properties of computed results in lazy functional logic programs;
- an extension to the functional logic paradigm of the semantics-based debugging approach of [26, 27], for logic programs, which extends declarative debugging to cope with the analysis of operational properties such as computed and correct answers, and abstract properties, e.g. $depth(k)$ answers and groundness dependencies;
- automatic synthesis of (property-oriented) specifications in terms of equations relating (nested) operation calls that have the same behavior.

I.1 The functional logic paradigm

The evolution of programming languages is the stepwise introduction of abstractions hiding the underlying computer hardware and the details of program execution. Along these lines, declarative languages – the most prominent representatives of which are functional and logic languages – hide the order of evaluation by removing assignment and other control statements.

In contrast to imperative programming, in a declarative setting one does not have to provide a description on *how* to obtain a solution to a problem by performing a sequence of steps but on *what* are the properties of the problem and the expected solutions. In fact, a declarative program is a set of logical statements describing properties of the application domain. The execution of a declarative program is the computation of the value(s) of an expression with respect to these properties. Thus, the programming effort shifts from encoding the steps for computing a result to structuring the application data and the relationships between the application components. Declarative languages could be seen as formal specification languages, but with a significant difference: *they are executable*. Different formalisms lead to different classes of declarative languages. This kinds of languages have similar motivations but provide different features. For example, functional

languages provide efficient, demand-driven (or lazy) evaluation strategies that support infinite structures, whereas logic languages provide nondeterminism and predicates with multiple input/output modes that offer code reuse.

Functional logic languages aim to combine the features of both paradigms in a conservative manner. Programs that do not use the features of one paradigm behave as programs of the other paradigm.

I.2 Computation properties and semantics

Semantics can help to develop practical tools such as those which performs program analysis, program debugging and program transformations. For this purpose we need notions of models which really capture the actual operational properties of functional logic programs and are, therefore, useful both for defining program equivalences and for semantic-based analysis.

A program admits a number of different semantics depending on which properties of the computation (observed behavior) we are interested in. A given choice of the property to observe, let us call it α, induces an *observational equivalence* on programs. Namely $P_1 \approx_\alpha P_2$ if and only if P_1 and P_2 can't be distinguished on α, that is they have the same behavior w.r.t. the observation made on α. In functional logic programs we can be interested in different behavioral properties: if we are only concerned with the input-output behavior of programs, we should just observe computed results. However, there are tasks, such as program analysis and optimization, which force us to observe and take into account other features of the derivations. In principle one could be interested in complete informations about the small-step behavior, namely the possible sequences of (narrowing) steps starting from any expression.

Defining an equivalence on programs $\approx_\alpha$ and a formal semantics $\mathcal{S}^\alpha[\![P]\!]$ are two strongly related tasks. If the semantics is *fully abstract*, that is $P_1 \approx_\alpha P_2$ if and only if $\mathcal{S}^\alpha[\![P_1]\!] = \mathcal{S}^\alpha[\![P_2]\!]$, then it identifies all and only the programs which can't be distinguished by α. Such an abstract model can be considered as the semantics of a language w.r.t. a given property α: all the other semantics can be reduced to it by abstracting from the redundant information. A non-fully abstract semantics makes the intended meaning of a program to include non relevant aspects, which do not depend on the behavior of the program but on a particular "implementation". Moreover this property is important, for instance, for deciding correctness of program transformation techniques. *Compositionality* is considered one of the most desirable characteristics of a formal semantics, since it provides a foundation for *program verification* and *modular design*. Compositionality has to do with a (syntactic) program construction operator $\circ$, and holds when the semantics of the compound construct $C_1 \circ C_2$ can be computed by composing the semantics of the constituents C_1 and C_2, i.e., if for a suitable homomorphism f, $\mathcal{S}^\alpha[\![P_1 \circ P_2]\!] = \mathcal{S}^\alpha[\![P_1]\!]\ f(\circ)\ \mathcal{S}^\alpha[\![P_2]\!]$.

Note that the typical operations that characterize a modular language are operations on the parts which compose the program rather than on the program itself. Separate compilation or analysis are examples of these operations. In fact, one of the most critical aspects in modular systems is the possibility of making a separate compilation of modules, and this can only be made in the presence of this property.

I.3 Abstract interpretation and program analysis

Abstract interpretation [29, 31], a technique for constructing verified analyses of program execution behavior, has been extensively applied to logic programming [57, 67, 44, 23]. The relevant feature of abstract interpretation is that, once the property intended to be observed has been modeled by an abstract domain, we have a methodology to systematically derive an abstract semantics, which in turn allows us to effectively compute a (correct) approximation of the property. By using this approach, most of the theorem-proving, in the logical theory involved in program verification, boils down to computing on the abstract domain. This is obtained in general at the expense of precision.

A program analysis is viewed as a non-standard, abstract semantics defined over a domain of data description. An abstract semantics is constructed by replacing operations in a suitable concrete semantics with the corresponding abstract operations defined on data descriptions, namely, *abstract domains*. Such domains are called abstract because they abstract, from the concrete computation domain, the properties of interest.

The definition of an appropriate concrete semantics, capable of modeling those program properties of interest, is a key point in abstract interpretation [29] and semantic-based data-flow analysis. Program analyses are then defined by providing finitely computable abstract interpretations which preserve interesting aspects of program behavior. Formal justification of program analyses is reduced to proving conditions on the relation between data and data descriptions and on the elementary operations defined on the data description.

In program analysis, abstract interpretation theory is often used to establish the correctness of specific analysis algorithms and abstract domains. We are more concerned instead in its application to the systematic derivation of the (optimal) abstract semantics from the abstract domain.

Abstract interpretation is inherently semantic sensitive and different semantic definition styles lead to different approaches to program analysis, the main are the *top-down* and the *bottom-up* approaches. In the case of functional logic programs, the most popular approach is the top-down, which propagates the information as narrowing does. In this class there are ad hoc algorithms, frameworks based on an operational semantics and frameworks based on a denotational semantics. The bottom-up approach propagates the information as in the computation of the least fixpoint of the immediate consequence operator. The main difference between the top-down and the bottom-up approach is usually related to *goal dependency*. In particular, a top-down analysis starts with a specific goal, while the bottom-up approach determines an approximation of properties of programs which is goal independent. Note, however, that goal-independent (concrete and abstract) semantics can also be defined in a top-down way.

I.4 Program debugging

The time and effort spent on validation of computer programs is known to take well over half of the total time for software development. Debugging is an essential ingredient in software development. Indeed the testing phase may also arise after the installation of the software and throughout its lifetime. An advantage of declarative languages is that they facilitate declarative programming. They make it possible, at least to a certain extent,

to separate the declarative, logical semantics (what is computed) from the operational semantics (how it is computed). Even though declarative languages aim at easing the task of programming, they can at the same time make debugging more difficult since the computation flow is often hard to predict. Thus the emergence of automatic tools for debugging closely follows the development of new programming paradigms and languages.

Debugging of functional logic programs is a special case of the general problem of debugging. The role of debugging in general is to identify and eliminate differences between the intended semantics of a program and its actual semantics. We will assume that the user has a clear idea about the results that should be computed by the program. An error occurs when the program computes something that the programmer did not intend (*incorrectness symptom*), or when it fails to compute something he was expecting (*incompleteness* or *insuffciency symptom*). In other words, incorrectness symptoms are answers which are in the actual program semantics but are not in the intended semantics, while incompleteness symptoms are answers which are in the intended semantics but are not in the actual program semantics.

In principle, all the reasoning concerning the correctness of a program can be done on the level of the declarative semantics, thus abstracting from any details of the computation. This advantage is however lost when debugging has to be done on the level of the concrete computations: this happens in the case of debugging tools which are different various versions of tracers. They force the programmer to think in terms of the sequences of computation steps. Thus it is important to develop debugging methods based on more abstract semantics which take into account *abstract* properties of the computation.

The debugging problem can formally be defined as follows. Let P be a program, $\mathcal{S}^\alpha[\![P]\!]$ be the behavior of P w.r.t. the property α, and $\mathcal{I}^\alpha$ be the specification of the *intended* behavior of P w.r.t. α. Debugging consists in comparing $\mathcal{S}^\alpha[\![P]\!]$ and $\mathcal{I}^\alpha$ and determining the wrong program components which are sources of errors, when $\mathcal{S}^\alpha[\![P]\!] \neq \mathcal{I}^\alpha$. The formulation is parametric w.r.t. the property α considered in the specification $\mathcal{I}^\alpha$ and in the actual behavior $\mathcal{S}^\alpha[\![P]\!]$.

Declarative debugging – firstly proposed for logic programs [88, 62, 40] then extended both for functional [72, 75] and functional logic programs [19, 17, 18, 14] – is concerned with model-theoretic properties. The specification is the intended declarative semantics (e.g. the least *Herbrand model* in [88], the set of atomic logical consequences in [40] and the *free term-model* in [19, 17, 18]). The idea behind declarative debugging is to collect information about what the program is intended to do and compare this with what it actually does. By reasoning from this, a diagnoser can find errors. The information needed can be found by asking the user a formal specification (which can be an extensive description of the intended program behavior or an older correct version of the program). The entities that provide the diagnoser with information are referred to as the *oracle*.

The declarative debugging method consists in two main techniques: incorrectness error diagnosis and insufficiency error diagnosis. The principal idea for find incorrectness errors is to inspect the *proof tree* constructed for an incorrectness symptom. To find the erroneous declaration (which corresponds to a clause in a logic program or an equation in a functional logic program) the diagnoser traverses the proof tree. At each node it asks the oracle about the validity of the corresponding atom. With the aid of the answers the diagnoser can identify the erroneous declaration. Dually, insufficiency error diagnosis concerns the case when a program fails to compute some expected results. The objective for insufficiency diagnosis is to scrutinize the attempt to construct a proof for a result which incorrectly

fails (or suspends). The reason for the error is located to one of the procedures (all clauses defining a predicate or all equations defining an operation) in the computation.

The idea in declarative debugging to restrict attention to model-theoretic properties has some limitations.

- The most natural observable for the diagnosis is that of *computed results*. It leads to a more precise diagnosis technique than the declarative debugging in [88, 40, 19, 17, 18], which can be reconstructed in terms of the more abstract observables *instances of computed results* and *correct answers*.
- Debugging w.r.t. $depth(k)$-answers or groundness dependencies makes diagnosis effective, since both $\mathcal{I}^\alpha$ and $\mathcal{S}^\alpha[\![P]\!]$ are finite.
- Debugging w.r.t. types allows us to detect bugs as the *inadmissible calls* in [81]. If $\mathcal{I}^\alpha$ specifies the intended program behavior w.r.t. types, abstract diagnosis boils down to type checking.
- Debugging w.r.t. modes and ground dependencies allows us to verify other partial program properties.

1

Preliminaries

Most of the notations used in this thesis are introduced in this chapter.

Essentially, this chapter presents the informal, logical and set-theoretic notations and concepts we will use to write down and reason about our ideas. The chapter is simply presented by using an informal extension of our everyday language to talk about mathematical objects like sets; it is not to be confused with the formal definitions about them that we will encounter later.

Some more specific notions will be introduced in the chapters where they are needed. For the terminology not explicitly shown and for a more motivated introduction about fixpoint theory and algebraic notation, the reader can consult [66, 13, 12].

1.1 Basic Set Theory

To define the basic notions we will use the standard (meta) logical notation to denote conjunction, disjunction, quantification and so on (*and*, *or*, *for each*, ...). We will use some informal logical notation in order to stop our mathematical statements getting out of hand. For statements (or assertions) A and B, we will commonly use abbreviations like:

$\boldsymbol{A, B}$ for (A and B), the conjunction of A and B,

$\boldsymbol{A \Longrightarrow B}$ for (A implies B), which means (if A then B),

$\boldsymbol{A \Longleftrightarrow B}$ for (A if and only if B), which expresses the logical equivalence of A and B.

We will also make statements by forming disjunctions (A or B), with the self-evident meaning, and negations (not A), sometimes written $\neg A$, which is true if and only if A is false. It is a tradition to write $x \not\leq y$ instead of $\neg(x \leq y)$.

A statement like $P(x, y)$, which involves variables x, y, is called a predicate (or property, or relation, or condition) and it only becomes true or false when the pair x, y stands for particular things. We use logical quantifiers $\exists$ (read "there exists") and $\forall$ (read "for all") to write assertions like $\exists x.\, P(x)$ as abbreviating "for some x, $P(x)$" or "there exists x such that $P(x)$", and $\forall x.\, P(x)$ as abbreviating "for all x, $P(x)$" or "for any x, $P(x)$". The statement $\exists x, y, \ldots, z.\, P(x, y, \ldots, z)$ abbreviates $\exists x. \exists y. \cdots \exists z.\, P(x, y, \ldots, z)$, and $\forall x, y, \ldots, z.\, P(x, y, \ldots, z)$ abbreviates $\forall x. \forall y. \cdots \forall z.\, P(x, y, \ldots, z)$. In order to specify a set S over which a quantifier ranges, we write $\forall x \in S.\, P(x)$ instead of $\forall x.\, x \in S \Longrightarrow P(x)$, and $\exists x \in S.\, P(x)$ instead of $\exists x.\, x \in S, P(x)$.

1.1.1 Sets

Intuitively, a set is an (unordered) collection of objects, which are elements (or members) of it. We write $a \in S$ when a is an element of the set S. Moreover, we write $\{a, b, c, \ldots\}$ for the set of elements $a, b, c, \ldots$.

A set S is said to be a subset of a set S', written $S \subseteq S'$, if and only if every element of S is an element of S', i.e., $S \subseteq S' \iff \forall z \in S.\, z \in S'$. A set is determined solely by its elements in the sense that two sets are equal if and only if they have the same elements. So, sets S and S' are equal, written $S = S'$, if and only if every element of S is an element of S' and vice versa.

Sets and Properties

A set can be determined by a property P. We write $S := \{x \,|\, P(x)\}$, meaning that the set S has as elements precisely all those x for which $P(x)$ is true. We will not be formal about it, but we will avoid trouble like Russell's paradox (see [85]) and will have at the same time a world of sets rich enough to support most mathematics. This will be achieved by assuming that certain given sets exist right from the start and by using safe methods for constructing new sets.

We write $\emptyset$ for the null or empty set and $\mathbb{N}$ for the set of natural numbers $0, 1, 2, \ldots$.

The cardinality of a set S is denoted by $|S|$. A set S is called *denumerable* if $|S| = |\mathbb{N}|$ and *countable* if $|S| \leq |\mathbb{N}|$.

Constructions on Sets

Let S be a set and $P(x)$ be a property. By $\{x \in S \,|\, P(x)\}$ we denote the set $\{x \,|\, x \in S, P(x)\}$. Sometimes, we will use a further abbreviation. Suppose $E(x_1, \ldots, x_n)$ is some expression which for particular elements $x_1 \in S_1, \ldots, x_n \in S_n$ yields a particular element and $P(x_1, \ldots, x_n)$ is a property of such $x_1, \ldots, x_n$. We use

$$\{E(x_1, \ldots, x_n) \,|\, x_1 \in S_1, \ldots, x_n \in S_n,\, P(x_1, \ldots, x_n)\}$$

to abbreviate $\{y \,|\, \exists x_1 \in S_1, \ldots, x_n \in S_n.\, y = E(x_1, \ldots, x_n),\, P(x_1, \ldots, x_n)\}$.

The *powerset* of a set S, $\{S' \mid S' \subseteq S\}$, is denoted by $\wp(S)$.

Let I be a set. By $\{x_i\}_{i \in I}$ (or $\{x_i \,|\, i \in I\}$) we denote the set of (unique) objects x_i, for any $i \in I$. The elements x_i are said to be *indexed* by the elements $i \in I$.

The *union* of two sets is $S \cup S' := \{a \,|\, a \in S \text{ or } a \in S'\}$. Let $\mathcal{S}$ be a set of sets, $\bigcup \mathcal{S} = \{a \,|\, \exists S \in \mathcal{S}.\, a \in S\}$. When $\mathcal{S} = \{S_i\}_{i \in I}$, for some indexing set I, we write $\bigcup \mathcal{S}$ as $\bigcup_{i \in I} S_i$. The *intersection* of two sets is $S \cap S' := \{a \,|\, a \in S, a \in S'\}$. Let $\mathcal{S}$ be a nonempty set of sets. Then $\bigcap \mathcal{S} := \{a \,|\, \forall S \in \mathcal{S}.\, a \in S\}$. When $\mathcal{S} = \{S_i\}_{i \in I}$ we write $\bigcap \mathcal{S}$ as $\bigcap_{i \in I} S_i$.

The *cartesian product* of S and S' is the set $S \times S' := \{(a, b) \,|\, a \in S, b \in S'\}$, the set of ordered pairs of elements with the first from S and the second from S'. More generally $S_1 \times S_2 \times \cdots \times S_n$ consists of the set of n-tuples $(x_1, \ldots, x_n)$ with $x_i \in S_i$ and S^n denotes the set of n-tuples of elements in S.

$S \setminus S'$ denotes the set where all the elements from S, which are also in S', have been removed, i.e., $S \setminus S' := \{x \,|\, x \in S, x \notin S'\}$.

1.1.2 Relations and Functions

A *binary relation* between S and S' ($R : S \times S'$) is an element of $\wp(S \times S')$. We write $x\ R\ y$ for $(x, y) \in R$.

A *partial function* from S to S' is a relation $f \subseteq S \times S'$ for which $\forall x, y, y'.\, (x, y) \in f, (x, y') \in f \implies y = y'$. By $f \colon S \rightharpoonup S'$ we denote a partial function of the set S (the *domain*) into the set S' (the *range*). The set of all partial functions from S to S' is denoted by $[S \rightharpoonup S']$. Moreover, we use the notation $f(x) = y$ when there is a y such that $(x, y) \in f$ and we say $f(x)$ is defined, otherwise $f(x)$ is undefined. Sometimes, when $f(x)$ is undefined, we write $f(x) \in \aleph$, where $\aleph$ denotes the undefined element. For each set S we assume that $\aleph \subseteq S$, $\aleph \cup S = S$ and $\emptyset \not\subseteq \aleph$. This will be formally motivated in Section 1.2.1.

Given a partial function $f \colon S \rightharpoonup S'$, the sets $\mathit{supp}(f) := \{x \in S \mid f(x) \text{ is defined}\}$ and $\mathit{img}(f) := \{f(x) \in S' \mid \exists x \in S.\, f(x) \text{ is defined}\}$ are, respectively, the *support* and the *image* of f. A partial function is said to be *finite-support* if $\mathit{supp}(f)$ is finite. Moreover, it is said to be *finite* if both $\mathit{supp}(f)$ and $\mathit{img}(f)$ are finite. In the following, we will often use finite-support partial functions. Hence, to simplify the notation, by

$$f := \begin{cases} v_1 \mapsto r_1 \\ \quad \vdots \\ v_n \mapsto r_n \end{cases}$$

we will denote (by cases) any function f which assumes on input values $v_1, \ldots, v_n$ output values $r_1, \ldots, r_n$ and is otherwise undefined. Furthermore, if the support of f is just the singleton $\{v\}$, we will denote it by $f := v \mapsto r$.

A (total) function f from S to S' is a partial function from S to S' such that, for all $x \in S$, there is some $y \in S'$ such that $f(x) = y$. That is equivalent as saying that f is total if $\mathit{supp}(f) = S$. Although total functions are a special kind of partial function, it is a tradition to understand something described as simply a function to be a total function. So we will always say explicitly when a function is partial. To indicate that a function f from S to S' is total, we write $f \colon S \to S'$. Moreover, the set of all (total) functions from S to S' is denoted by $[S \to S']$.

A function $f \colon S \to S'$ is *injective* if and only if for each $x, y \in S$ if $f(x) = f(y)$ then $x = y$. f is *surjective* if and only if for each $x' \in S'$ there exists $x \in S$ such that $f(x) = x'$.

We denote by $f = g$ the extensional equality, i.e., for each $x \in S$, $f(x) = g(x)$.

Lambda Notation

It is sometimes useful to use the lambda notation to describe functions. It provides a way of referring to functions without having to name them. Suppose $f \colon S \to S'$ is a function which, for any element $x \in S$, gives a value $f(x)$ which is exactly described by expression E, probably involving x. Then we can write $\lambda x \in S.\, E$ for the function f. Thus, $(\lambda x \in S.\, E) := \{(x, E[x]) \,|\, x \in S\}$ and so $\lambda x \in S.\, E$ is just an abbreviation for the set of input-output values determined by the expression $E[x]$. We use the lambda notation also to denote partial functions by allowing expressions in lambda-terms that are not always defined. Hence, a lambda expression $\lambda x \in S.\, E$ denotes a partial function $S \rightharpoonup S'$ which, on input $x \in S$, assumes the value $E[x] \in S'$, if the expression $E[x]$ is defined, and otherwise it is undefined.

Composing Relations and Functions

We compose relations, and so partial and total functions, $R : S \times S'$ and $Q : S' \times S''$ by defining their *composition* (a relation between S and S'') by $Q \circ R := \{(x, z) \in S \times S'' \mid y \in S', (x, y) \in R, (y, z) \in Q\}$. R^n is the relation

$$\underbrace{R \circ \cdots \circ R}_{n},$$

i.e., $R^1 := R$ and (assuming R^n is defined) $R^{n+1} := R \circ R^n$. Each set S is associated with an identity function $Id_S := \{(x, x) \mid x \in S\}$, which is the neutral element of $\circ$. Thus we define $R^0 := Id_S$.

The *transitive and reflexive closure* R^* of a relation R on S is $R^* := \bigcup_{i \in \mathbb{N}} R^i$.

The *function composition* of $g \colon S \rightharpoonup S'$ and $f : S' \rightharpoonup S''$ is the partial function $f \circ g \colon S \rightharpoonup S''$, where $(f \circ g)(x) := f(g(x))$, if $g(x)$ (first) and $f(g(x))$ (then) are defined, and it is otherwise undefined. When it is clear from the context $\circ$ will be omitted.

A function $f : S \to S'$ is *bijective* if it has an *inverse* $g : S' \to S$, i.e., if and only if there exists a function g such that $g \circ f = Id_S$ and $f \circ g = Id_{S'}$. Then the sets S and S' are said to be in 1-1 correspondence. Any set in 1-1 correspondence with a subset of natural numbers $\mathbb{N}$ is said to be countable. Note that a function f is bijective if and only if it is injective and surjective.

Direct and Inverse Image of a Relation

We extend relations, and thus partial and total functions, $R \colon S \times S'$ to functions on subsets by taking $R(X) := \{y \in S' \mid \exists x \in X.\, (x, y) \in R\}$ for $X \subseteq S$. The set $R(X)$ is called the *direct image* of X under R. We define $R^{-1}(Y) := \{x \in S \mid \exists y \in Y.\, (x, y) \in R\}$ for $Y \subseteq S'$. The set $R^{-1}(Y)$ is called the *inverse image* of Y under R. Thus, if $f \colon S \rightharpoonup S'$ is a partial function, $X \subseteq S$ and $X' \subseteq S'$, we denote by $f(X)$ the *image* of X under f, i.e., $f(X) := \{f(x) \mid x \in X\}$ and by $f^{-1}(X')$ the *inverse image* of X' under f, i.e., $f^{-1}(X') := \{x \mid f(x) \in X'\}$.

Equivalence Relations and Congruences

An *equivalence relation* $\approx$ on a set S is a binary relation on S ($\approx \colon S \times S$) such that, for each $x, y, z \in S$,

$x \; R \; x$	(reflexivity)
$x \; R \; y \Longrightarrow y \; R \; x$	(symmetricity)
$x \; R \; y, y \; R \; z \Longrightarrow x \; R \; z$	(transitivity)

The *equivalence class* of an element $x \in S$, with respect to $\approx$, is the subset $[x]_\approx := \{y \mid x \approx y\}$. When clear from the context we abbreviate $[x]_\approx$ by $[x]$ and often abuse notation by letting the elements of a set denote their correspondent equivalence classes. The *quotient set* $S/_\approx$ of S modulo $\approx$ is the set of equivalence classes of elements in S (w.r.t. $\approx$).

An equivalence relation $\approx$ on S is a *congruence* w.r.t. a partial function $f : S^n \rightharpoonup S$ if and only if, for each pair of elements a_i, $b_i \in S$ such that $a_i \approx b_i$, (if $f(a_1, \ldots, a_n)$ is defined then also $f(b_1, \ldots, b_n)$ is defined and)

$$f(a_1, \ldots, a_n) \approx f(b_1, \ldots, b_n).$$

Then, we can define the partial function $f_{\approx}\colon (S/_{\approx})^n \rightharpoonup S/_{\approx}$ as

$$f_{\approx}([a_1]_{\approx}, \ldots, [a_n]_{\approx}) := [f(a_1, \ldots, a_n)]_{\approx},$$

since, given $[a_1]_{\approx}, \ldots, [a_n]_{\approx}$, the class $[f(a_1, \ldots, a_n)]_{\approx}$ is uniquely determined independently of the choice of the representatives $a_1, \ldots, a_n$.

1.2 Domain Theory

We will present here the (abstract) concepts of complete lattices, continuous functions and fixpoint theory, which are the standard tools of denotational semantics.

1.2.1 Complete Lattices and Continuous Functions

A binary relation $\leq$ on S ($\leq\colon S \times S$) is a *partial order* if, for each $x, y \in S$,

$x \leq x$	(reflexivity)
$x \leq y, y \leq x \Longrightarrow x = y$	(antisymmetry)
$x \leq y, y \leq z \Longrightarrow x \leq z$	(transitivity)

A *partially ordered set* (poset) $(S, \leq)$ is a set S equipped with a partial order $\leq$. A set S is *totally ordered* if it is partially ordered and, for each $x, y \in S$, $x \leq y$ or $y \leq x$. A *chain* is a (possibly empty) totally ordered subset of S.

A *preorder* is a binary relation which is reflexive and transitive. A preorder $\leq$ on a set S induces on S an equivalence relation $\approx$ defined as follows: for each $x, y \in S$,

$$x \approx y \Longleftrightarrow x \leq y, y \leq x.$$

Moreover, $\leq$ induces on $S/_{\approx}$ the partial order $\leq_{\approx}$ such that, for each $[x]_{\approx}, [y]_{\approx} \in S/_{\approx}$,

$$[x]_{\approx} \leq_{\approx} [y]_{\approx} \Longleftrightarrow x \leq y.$$

A binary relation $<$ is *strict* if and only if it is anti-reflexive (i.e., not $x < x$) and transitive.

Given a poset $(S, \leq)$ and $X \subseteq S$, $y \in S$ is an *upper bound* for X if and only if, for each $x \in X$, $x \leq y$. Moreover, $y \in S$ is the *least upper bound* (called also join) of X, if y is an upper bound of X and, for every upper bound y' of X, $y \leq y'$. A least upper bound of X is often denoted by $lub_S\, X$ or by $\bigsqcup_S X$. We also write $\bigsqcup_S \{d_1, \ldots, d_n\}$ as $d_1 \sqcup_S \cdots \sqcup_S d_n$. Dually an element $y \in S$ is a *lower bound* for X if and only if, for each $x \in X$, $y \leq x$. Moreover, $y \in S$ is the *greatest lower bound* (called also meet) of X, if y is a lower bound of X and for every lower bound y' of X, $y' \leq y$. A greatest lower bound of X is often denoted by $glb_S\, X$ or by $\bigsqcap_S X$. We also write $\bigsqcap_S \{d_1, \ldots, d_n\}$ as $d_1 \sqcap_S \cdots \sqcap_S d_n$. When it is clear from the context, the subscript S will be omitted. Moreover $\bigsqcup\{D_i\}_{i \in I}$ and $\bigsqcap\{D_i\}_{i \in I}$ can be denoted by $\bigsqcup_{i \in I} D_i$ and $\bigsqcap_{i \in I} D_i$. It is easy to check that if *lub* and *glb* exist, then they are unique.

Complete Partial Orders and Lattices

A *direct set* is a poset in which any subset of two elements (and hence any finite subset) has an upper bound in the set. A *complete partial order* (*CPO*) S is a poset such that every chain D has the least upper bound (i.e., there exists $\bigsqcup D$). Notice that any set ordered by the identity relation forms a *CPO*, of course without a bottom element. Such *CPO*s are called discrete. We can add a bottom element to any poset $(S, \leq)$ which does not have one (even to a poset which already has one). The new poset $S_\perp$ is obtained by adding a new element $\perp$ to S and by extending the ordering $\leq$ as $\forall x \in S. \perp \leq x$. If S is a discrete *CPO*, then $S_\perp$ is a *CPO* with bottom element, which is called flat.

A *complete lattice* is a poset $(S, \leq)$ such that for every subset X of S there exists $\bigsqcup X$ and $\bigsqcap X$. Let $\top$ denote the *top element* $\bigsqcup S = \bigsqcap \emptyset$ and $\perp$ denote the *bottom element* $\bigsqcap S = \bigsqcup \emptyset$ of S. The elements of a complete lattice are thought of as points of information and the ordering as an approximation relation between them. Thus, $x \leq y$ means x approximates y (or, x has less or the same information as y) and so $\perp$ is the point of least information. It is easy to check that, for any set S, $\wp(S)$ under the subset ordering $\subseteq$ is a complete lattice, where $\sqcup$ is union, $\sqcap$ is intersection, the top element is S and the bottom element is $\emptyset$. Also $(\wp(S))_\perp$ is a complete lattice.

Given a complete lattice $(L, \leq)$, the set of all partial functions $F = [S \rightharpoonup L]$ inherits the complete lattice structure of L. Let simply define $f \preceq g := \forall x \in S. f(x) \leq g(x)$, $(f \sqcup g)(x) := f(x) \sqcup g(x)$, $(f \sqcap g)(x) := f(x) \sqcap g(x)$, $\perp_F := \lambda x \in S. \perp_L$ and $\top_F := \lambda x \in S. \top_L$.

Continuous and Additive Functions

Let $(L, \leq)$ and $(M, \sqsubseteq)$ be (complete) lattices. A function $f : L \to M$ is *monotonic* if and only if

$$\forall x, y \in L. x \leq y \Longrightarrow f(x) \sqsubseteq f(y).$$

Moreover, f is *continuous* if and only if, for each non-empty chain $D \subseteq L$,

$$f(\bigsqcup_L D) = \bigsqcup_M f(D).$$

Every continuous function is also monotonic, since $x \leq y$ implies $f(\bigsqcup_L \{x, y\}) = f(y)$, by continuity $\bigsqcup_M \{f(x), f(y)\} = f(\bigsqcup_L \{x, y\})$, which implies that $f(x) \sqsubseteq f(y)$, since $f(x) \sqsubseteq \bigsqcup_M \{f(x), f(y)\}$ and we already seen that $f(\bigsqcup_L \{x, y\}) = f(y)$.

Complete partial orders correspond to types of data (data that can be used as input or output to a computation) and computable functions are modeled as continuous functions between them.

A partial function $f : S \rightharpoonup S'$ is *additive* if and only if the previous continuity condition is satisfied for each non-empty set. Hence, every additive function is also continuous. Dually we define *co-continuity* and *co-additivity*, by using $\sqcap$ instead of $\sqcup$.

It can be proved that the composition of monotonic, continuous or additive functions is, respectively, monotonic, continuous or additive.

The mathematical way of expressing that structures are "essentially the same" is through the concept of isomorphism which establishes when structures are isomorphic. A continuous function $f : D \to E$ between *CPO*s D and E is said to be an *isomorphism*

if there is a continuous function $g\colon E \to D$ such that $g \circ f = Id_D$ and $f \circ g = Id_E$. Thus f and g are mutual inverses. This is actually an instance of a general definition which applies to a class of objects and functions between them (*CPO*s and continuous functions in this case). It follows from the definition that isomorphic *CPO*s are essentially the same but for a renaming of elements. It can be proved that a function $f\colon D \to E$ is an isomorphism if and only if f is bijective and, for all $x, y \in D$, $x \leq_D y \iff f(x) \leq_E f(y)$.

Function Space

Let D, E be *CPO*s. It is a very important fact that the set of all continuous functions from D to E can be made into a complete partial order. The function space $[D \to E]$ consists of continuous functions $f\colon D \to E$ ordered pointwise by $f \sqsubseteq g \iff \forall d \in D.\, f(d) \sqsubseteq g(d)$. This makes the function space a complete partial order. Note that, provided E has a bottom element $\bot_E$, such a function space of *CPO*s has a bottom element, the constantly $\bot_E$ function $\bot_{[D\to E]} := \lambda d \in D.\, \bot_E$. Least upper bounds of chains of functions are given pointwise, i.e., a chain of functions $f_0 \sqsubseteq f_1 \sqsubseteq \ldots \sqsubseteq f_n \sqsubseteq \ldots$ has *lub* $\bigsqcup_{[D\to E]} f_n := \lambda d \in D.\, \bigsqcup_E \{f_n(d)\}_{n\in\mathbb{N}}$.

It is not hard to see that the partial functions $L \rightharpoonup D$ are in 1-1 correspondence with the (total) functions $L \to D_\bot$, and that, in this case, any total function is continuous; the inclusion order between partial functions corresponds to the "pointwise order" $f \sqsubseteq g \iff \forall \sigma \in L.\, f(\sigma) \sqsubseteq g(\sigma)$ between functions $L \to D_\bot$. Because partial functions from a *CPO* so does the set of functions $[L \to D_\bot]$ ordered pointwise. This is the reason why we assumed that, for each set S, $\aleph \subseteq S$, $\aleph \cup S = S$ and $\emptyset \not\subseteq \aleph$.

1.2.2 Fixpoint Theory

Given a poset $(S, \leq)$ and a function $f\colon S \to S$, a *fixpoint* of f is an element $x \in S$ such that $f(x) = x$. A *pre-fixpoint* of f is an element $x \in S$ such that $f(x) \leq x$ and dually a *post-fixpoint* of f is an element $x \in S$ such that $x \leq f(x)$. Moreover, we say that $x \in S$ is the *least fixpoint of* f (denoted by *lfp* f) if and only if x is a fixpoint of f and for all fixpoints y of f, $x \leq y$. Dually, we define the *greatest fixpoint* (denoted by *gfp* f).

The fundamental theorem of Knaster-Tarski states that the set of fixpoints of a monotonic function f is a complete lattice.

Theorem 1.2.1 (Fixpoint theorem) *[92] A monotonic function f on a complete lattice $(L, \leq)$ has the least fixpoint and the greatest fixpoint. Moreover,*

$$lfp(f) = \bigsqcap \{x \mid f(x) \leq x\} = \bigsqcap \{x \mid x = f(x)\}$$
$$gfp(f) = \bigsqcup \{x \mid x \leq f(x)\} = \bigsqcup \{x \mid x = f(x)\}.$$

The Knaster-Tarski Theorem is important because it applies to any monotone function on a complete lattice. However, most of the time we will be concerned with least fixpoints of continuous functions which we will construct by the techniques of the previous section, as least upper bounds of chains in a *CPO*. Therefore, it's useful to state some more notations and results on fixpoints of continuous functions defined on (complete) lattices.

First of all we have to introduce the notion of *ordinal*. We assume that an ordinal is a set, where every element of an ordinal is still an ordinal and the class of ordinals is ordered by membership relation ($\alpha < \beta$ means $\alpha \in \beta$). Consequently, every ordinal coincides with

the set of all smaller ordinals. The least ordinals are 0, $1 := \{0\}$, $2 := \{0, \{0\}\}$, *etc.*. Intuitively, the class of ordinals is the transfinite sequence $0 < 1 < 2 < \ldots < \omega < \omega + 1 < \ldots < \omega + \omega < \ldots < \omega^{\omega}$, *etc.*. Ordinals will be often denoted by Greek letters. An ordinal γ is a *limit ordinal* if it is neither 0 nor the successor of an ordinal; so, if $\beta < \gamma$, then there exists σ such that $\beta < \sigma < \gamma$. The first limit ordinal, which is equipotent with the set of natural numbers, is denoted (by an abuse of notation) by ω. Often, in the definitions of *CPO* and of continuity, directed sets are used instead of chains. It is possible to show that if the set S is denumerable, then the definitions are equivalent.

The ordinal powers of a monotonic function $T\colon S \to S$ on a *CPO* S are defined as

$$T{\uparrow}\alpha(x) := \begin{cases} x & \text{if } \alpha = 0 \\ T(T{\uparrow}(\alpha - 1)(x)) & \text{if } \alpha \text{ is a successor ordinal} \\ \bigsqcup\{T{\uparrow}\beta(x) \mid \beta < \alpha\} & \text{if } \alpha \text{ is a limit ordinal.} \end{cases}$$

In the following, we will use the standard notation $T{\uparrow}\alpha := T{\uparrow}\alpha(\bot)$, where $\bot$ is the least element of S. In particular, $T{\uparrow}\omega := \bigsqcup_{n<\omega} T{\uparrow}n$, $T{\uparrow}n + 1 := T(T{\uparrow}n)$, for $n < \omega$, and $T{\uparrow}0 := \bot$, where $\bigsqcup$ is the *lub* operation of S. Sometimes, $T{\uparrow}\alpha(x)$ may be denoted simply by $T^{\alpha}(x)$.

The next important result is usually attributed to Kleene and gives an explicit construction of the least fixpoint of a continuous function f on a *CPO* D.

Theorem 1.2.2 (Fixpoint Theorem) *Let $f\colon D \to D$ be a continuous function on a CPO D and $d \in D$ be a pre-fixpoint of f. Then $\bigsqcup\{f{\uparrow}n(d) \mid n \leq \omega\}$ is the least fixpoint of f greater than d. In particular $f{\uparrow}\omega$ is the least pre-fixpoint and least fixpoint of f.*

Each *CPO* D with bottom $\bot$ is associated with a fixpoint operator $\mathit{fix}\colon [D \to D] \to D$, $\mathit{fix} := \bigsqcup_{n<\omega}(\lambda f.\, f^n(\bot))$, i.e., *fix* is the least upper bound of the chain of the functions $\lambda f.\, \bot \sqsubseteq \lambda f.\, f(\bot) \sqsubseteq \lambda f.\, f(f(\bot)) \sqsubseteq \ldots$, where each of these is continuous and so an element of the *CPO* $[[D \to D] \to D]$.

1.3 Haskell and Curry Languages

In this section we describe those features of declarative programming that are preliminary for the developments in this thesis. We present such features by means of the two languages we'll take into consideration in this thesis, namely Haskell (for the functional paradigm) and Curry (for the functional logic paradigm). We chose these particular languages both for they importance and for they syntactic similarities[1].

For a detailed description about the syntax of Haskell we refer to [55, 82], and for Curry to [51].

We start in Subsection 1.3.1 with important concepts found in functional programming languages, namely, higher-order functions and demand-driven evaluation. Subsection 1.3.2 describes essential features of logic programming, namely, nondeterminism, unknown values and built-in search

[1] Actually Curry has born as a semantic extension of Haskell with the features of the logic paradigm.

1.3.1 The Haskell Language

While running an imperative program means to execute commands, running a functional program means to evaluate expressions. Functions in a functional program are functions in a mathematical sense: the result of a function call depends only on the values of the arguments. Functions in imperative programming languages may have access to variables other than their arguments and the result of such a "function" may also depend on those variables. Moreover, the values of such variables may be changed after the function call, thus, the meaning of a function call is not solely determined by the result it returns. Because of such side effects, the meaning of an imperative program may be different depending on the order in which function calls are executed. An important aspect of functional programs is that they do not have side effects and, hence, the result of evaluating an expression is determined only by the parts of the expression – not by evaluation order. As a consequence, functional programs can be evaluated with different evaluation strategies, like demand-driven evaluation.

Higher-order features

Functions in a functional program can not only map data to data but may also take functions as arguments or return them as result. In type signatures of higher-order functions, parentheses are used to group functional arguments. Probably the simplest example of a higher-order function is the infix operator $ for function application:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Another useful operator is function composition

```
(.) :: (a -> b) -> (b -> c) -> (a -> c)
f . g = \x -> f (g x)
```

This definition uses a *lambda abstraction* that denotes an *anonymous function*. The operator for function composition is a function that takes two functions as arguments and yields a function as result. Lambda abstractions have the form `\x -> e` where x is a variable and e is an arbitrary expression. The variable x is the argument and the expression e is the body of the anonymous function. The body may itself be a function and the notation `\x y z -> e` is short hand for `\x -> \y -> \z -> e`. While the first of these lambda abstractions looks like a function with three arguments, the second looks like a function that yields a function that yields a function. In Haskell, there is no difference between the two. A function that takes many arguments is a function that takes one argument and yields a function that takes the remaining arguments. Representing functions like this is called *currying*.

There are a number of predefined higher-order functions for list processing. In order to get a feeling for the abstraction facilities they provide, we discuss one of them here. The *map* function applies a given function to every element of a given list:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

If the given list is empty, then the result is also the empty list. If it contains at least the element x in front of an arbitrary list xs of remaining elements, then the result of calling map is a non-empty list where the first element is computed using the given function f and the remaining elements are processed recursively. The type signature of map specifies that (i) the argument type of the given function f and the element type of the given list and, (ii) the result type of f and the element type of the result list must be equal. For instance, `map length ["Haskell", "Curry"]` is a valid application of `map` because the a in the type signature of map can be instantiated with `String` which is defined as `[Char]` and matches the argument type `[a]` of `length`. The type `b` is instantiated with `Int` and, therefore, the returned list has the type `[Int]`.

The type signature is a partial documentation for the function map because we get an idea of what map does without looking at its implementation. If we do not provide the type signature, then *type inference* deduces it automatically from the implementation.

Lazy evaluation

With lazy evaluation arguments of functions are only computed as much as necessary to compute the result of a function call. Parts of the arguments that are not needed to compute a result are not demanded and may contain divergent and/or expensive computations. For example, we can compute the length of a list without demanding the list elements. In a programming language with lazy evaluation, like Haskell, we can compute the result of `length [loop, fibonacci 100]`. Neither the diverging computation `loop` nor the possibly expensive computation `fibonacci 100` are evaluated to compute the result 2. This example demonstrates that lazy evaluation can be faster than eager evaluation because unnecessary computations are skipped.

With lazy evaluation we can not only handle large data efficiently, we can even handle unbounded, potentially infinite data. Consider the function `takeWhile`

```
takeWhile                :: (a -> Bool) -> [a] -> [a]
takeWhile _ []           =  []
takeWhile p (x:xs)
            | p x        =  x : takeWhile p xs
            | otherwise =  []
```

that returns the longest prefix (possibly empty) of `xs` of elements that satisfy the predicate `p`, and the function `iterate`

```
iterate :: (a -> a) -> a -> [a]
iterate f x =  x : iterate f (f x)
```

which, returns an infinite list of repeated applications of `f` to `x`. With lazy evaluation we can generate an unbounded list number of increasing numbers, namely `iterate (\n -> 2*n) 1`, selecting those which are less then 100:

```
takeWhile (< 100) (iterate (\n -> 2*n) 1)
```

Sharing

Conceptually, the call `iterate f x` yields the infinite list $[x, fx, f(fx), f(f(fx)), \ldots]$. The elements of this list are only computed if they are demanded by the surrounding

computation because lazy evaluation is *non-strict*. Although it is duplicated in the right-hand side of iterate, the argument `x` is evaluated at most once because lazy evaluation is sharing the values that are bound to variables once they are computed. If we call `iterate sqrt (fibonacci 100)`, then the call `fibonacci 100` is only evaluated once, although it is duplicated by the definition of iterate. Sharing of sub computations ensures that lazy evaluation does not perform more steps than a corresponding eager evaluation because computations bound to duplicated variables are performed only once even if they are demanded after they are duplicated.

1.3.2 The Curry Language

Functional programming, discussed in the previous section, is one important branch in the field of declarative programming. Logic programming is another. Despite conceptual differences, research on combining these paradigms has shown that their conceptual divide is not as big as one might expect. The programming language Curry unifies lazy functional programming as in Haskell with essential features of logic programming. The main extensions of Curry compared to the pure functional language Haskell are: logic variables, nondeterministic operations, and built-in search.

Logic variables

The most important syntactic extension of Curry compared to Haskell are declarations of logic variables. Instead of binding variables to expressions, Curry programmers can state that the value of a variable is unknown by declaring it `free`. A logic variable will be bound during execution according to demand: just like patterns in the left-hand side of functions cause unevaluated expressions to be evaluated, they cause unbound logic variables to be bound.

Consider a simple deductive database with family relationships[2].

```
data Person = Mike | Hanna | George | John

parent :: Person -> Person -> Success
parent Mike Hanna = success
parent Hanna George = success
parent Hanna John = success

ancestor :: Person -> Person -> Success
ancestor x y = parent x y
ancestor x y = parent x z &> ancestor z x where z free
```

`parent` represents the basic parental relationship, while `ancestor` is a deduced relationship which yields success if and only if there exists a parental path in the defined database between two given persons. For instance the call `ancestor Mike John` yields success since `Mike` is parent of `Hanna` which is parent of `John`.

We can also use logical variables to query the database. For instance the goal

```
> ancestor Mike p where p free
```

[2] We use a relational programming style, i.e., all relationships are represented as constraints (i.e., functions with result type `Success`)

asks every persons `p` who has `Mike` as ancestor. Thus the response is:

```
{p=Hanna} | {p=George} | {p=John}
```

Another example is the function `last` which returns the last element of a given list `xs`. As in the logic paradigm, instead of having to write a recursive definition explicitly, we can use the property that `last ls` equals `x` if and only if there is a list `xs` such that `xs ++ [x]` equals `ls`.

```
last :: [a] -> a
last ls | ls =:= xs++[x] = x where x,xs free
```

The possibility to use predicates that involve previously defined operations to define new ones improves the possibility of code reuse in functional logic programs. Logic variables are considered existentially quantified and the evaluation mechanism of Curry includes a search for possible instantiations.

Nondeterministic operations

The built-in search for instantiations of logic variables can lead to different possible instantiations and, hence, nondeterministic results of computations. Consider, for example, the following definition of `insert`:

```
insert :: a -> [a] -> [a]
insert x ls |  ls =:= xs++ys = xs ++ x:ys
        where xs,ys free
```

If the argument `ls` of insert is non-empty then there are different possible bindings for `xs` and `ys` such that `xs ++ ys =:= l`. Consequently, the result of insert may contain `x` at different positions and, thus, there is more than one possible result when applying insert to a non-empty list. Mathematically, insert does not denote a function that maps arguments to deterministic results but a relation that specifies a correspondence of arguments to possibly nondeterministic results. To avoid the contradictory term nondeterministic function we call `insert` (and other defined operations that may have more than one result) nondeterministic operation.

Variable instantiations are not the only source of nondeterminism in Curry programs. As the run-time system needs to handle nondeterminism anyway, Curry also provides a direct way to define nondeterministic operations. Unlike in Haskell, the meaning of defined Curry operations does not depend on the order of their defining rules. While in Haskell the rules of a function are tried from top to bottom committing to the first matching rule[3], in Curry the rules of an operation are tried nondeterministically. As a consequence, overlapping rules lead to possibly nondeterministic results. We can use overlapping rules to give an alternative implementation of the insert operation.

```
insert x ls = x: ls
insert x (y:ys) = y: insert x ys
```

This definition either inserts the given element `x` in front of the given list `ls` or – if `ls` is non-empty – inserts `x` in the tail `ys` of `ls`, leaving the head `y` in its original position.

[3] We will see that this rule selection strategy, actually implements a sort of orthogonalization of the constructor based TRS. It is well known that orthogonal TRSs are confluent. This property is necessary to define (partial) functions.

This version of insert is more lazy than the former version: while the equality constraint `=:=` (which is strict) in the guard forces the evaluation of both arguments of insert, the version with overlapping rules can yield a result without evaluating any of the arguments. In the following we use the second definition of insert to benefit from its laziness. The advantage of implicit nondeterminism (as opposed to explicitly using, for example, lists to represent multiple results) is that the source code does not contain additional combinators to handle nondeterminism which eases the composition of more complex nondeterministic operations from simpler ones. For example, we can compute permutations of a given list nondeterministically by recursively inserting all its elements into an empty list.

```
perm :: [a] -> [a]
perm [] = []
perm (x:xs) = insert x (perm xs)
```

In addition to the convenience of using the features of both paradigms within a single language, the combination has additional advantages. For instance, the demand-driven evaluation of functional programming applied to nondeterministic operations of logic programming leads to more efficient search strategies.

For instance, one can use the non-deterministic operation `perm` to define a sorting function `psort` based on a "partial identity" function `sorted` that returns its input list if it is sorted:

```
sorted [] = []
sorted [x] = [x]
sorted (x:y:xs) | x <= y = x : sorted (y:xs)
psort xs = sorted (perm xs)
```

Thus, `psort xs` returns only those permutations of `xs` that are sorted. The advantage of this definition of psort in comparison to traditional "generate-and-test" solutions becomes apparent when one considers the demand-driven evaluation strategy. Since in an expression like `sorted (perm xs)` the argument `perm xs` is only evaluated as demanded by sorted, the permutations are not fully computed at once. If a permutation starts with a non-ordered prefix, like `1:0:perm xs`, the application of the third rule of sorted fails and, thus, the computation of the remaining part of the permutation (which can result in $n!$ different permutations if n is the length of the list `xs`) is discarded. The overall effect is a reduction in complexity in comparison to the traditional generate-and-test solution.

Thus nondeterministic operations have advantages w.r.t. demand-driven evaluation strategies so that they became a standard feature of recent functional logic languages (whereas older languages put confluence requirements on their programs).

Call-time choice sematics

A subtle aspect of nondeterministic operations is their treatment if they are passed as arguments. For instance, consider the call `double coin` and the program

```
coin = 0
coin = 1
double x = x + x
```

If the argument `coin` is evaluated (to `0` or `1`) before it is passed to `double`, we obtain the possible results `0` and `2`. However, if the argument `coin` is passed unevaluated to

`double` (as it should be done with a pure lazy strategy), we obtain after one reduction step the expression `coin+coin` which has four possible results `0`, `1`, `1`, and `2`. The former behavior is referred to as *call-time choice* semantics [56] since the choice for the desired value of a nondeterministic operation is made at call time, whereas the latter is referred to as *need-time choice* semantics.

Although call-time choice suggests an eager or call-by-value strategy, it fits well into the framework of demand-driven evaluation where arguments are shared to avoid multiple evaluations of the same subexpression. For instance, the actual subexpression (e.g. `coin`) associated to argument `x` in the rule for `double` is not duplicated in the right-hand side but a reference to it is passed so that, if it is evaluated by one subcomputation, the same result will be taken in the other subcomputation. In contrast to Haskell, sharing is not only a technique essential to obtain efficient (and optimal) evaluation strategies, but in presence of nondeterministic operations, if used, it implements the call-time choice semantics without any further machinery.

Furthermore, in many situations call-time choice is the semantics with the "least astonishment". For instance, consider the reformulation of the operation `psort` to

```
psort xs = idOnSorted (perm xs)
idOnSorted xs | sorted xs =:= xs = xs
```

then, for the call `psort xs`, the call-time choice semantics delivers only sorted permutations of `xs`, as expected, whereas the need-time choice semantics delivers all permutations of `xs` since the different occurrences of `xs` in the rule of `idOnSorted` are not shared. Due to these reasons, Curry and current functional logic languages like $\mathcal{TOY}$ adopt the call-time choice semantics.

1.4 First Order Functional Logic (and Functional) Programming

1.4.1 Terms, Equations and Substitutions

For detailed preliminaries about Term Rewriting the reader can consult [8, 59, 93]. For simplicity, definitions are given in the one-sorted case. The extension to many–sorted signatures is straightforward, see [78].

We consider a typed signature Σ partitioned into a set $\mathcal{C}$ of constructor (also called data constructors) and a set $\mathcal{D}$ of defined symbols (also called operations). We write $c_{/n} \in \mathcal{C}$ and $f_{/n} \in \mathcal{D}$ for any n-ary constructor and defined symbols, respectively. The set of well-typed *terms* and well-typed *constructor terms* with *variables* from $\mathcal{V}$ are denoted by $\mathcal{T}(\Sigma, \mathcal{V})$ and $\mathcal{T}(\mathcal{C}, \mathcal{V})$, respectively. The set of variables occurring in a term t is denoted by $var(t)$. A term is *linear* if it does not contain multiple occurrences of any variable. We write $\overrightarrow{o_n}$ for the *list* of syntactic objects $o_1, \ldots, o_n$. A *pattern* is a term of the form $f(\overrightarrow{t_n})$ where $f_{/n} \in \mathcal{D}$ and $t_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ for every $i \in \{1, \ldots, n\}$. A term t is *operation-rooted* (respectively constructor-rooted) if it has a defined (respectively constructor) symbol at the root. A *position* p in term t is represented by a sequence of natural numbers (Λ denotes the empty sequence, i.e., the root position). $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s. $\equiv$ denotes syntactic equality.

We denote by $\{x_1/t_1, \ldots, x_n/t_n\}$ the substitution σ with $\sigma(x_i) = t_i$ for $i \in \{1, \ldots, n\}$, and $\sigma(x) = x$ for all other variables x; $dom(\sigma)$, $img(\sigma)$ and $range(\sigma)$ indicate the domain set $\{x_1, \ldots, x_n\}$, the image set $\{t_1, \ldots, t_n\}$ and the range set $\bigcup_{i=1}^{n} var(t_i)$ respectively. A substitution σ is *(ground) constructor*, if $\sigma(x)$ is (ground) constructor for all $x \in dom(\sigma)$. *Substs* and $\mathcal{C}$*Substs* indicate the set of substitutions and the set of constructor substitutions respectively. A substitution $\sigma := \{x_1/t_1, \ldots, x_n/t_n\}$ is *linear* if and only if for every $i, j \in \{1, \ldots, n\}$ t_i is a linear term and $i \neq j \Rightarrow var(t_i) \cap var(t_j) = \emptyset$. Furthermore, if σ belongs to $\mathcal{C}$*Substs* then σ is said to be $\mathcal{C}$-linear (constructor linear).

The identity substitution is denoted by ε. Given a term t and a substitution σ, $\sigma(t)$ (or $t\sigma$) indicates the term obtained from t by replacing all the occurrences of a variable x in it by $\sigma(x)$. Given a substitution σ and a set of variables $V \subseteq \mathcal{V}$ we denote by $\sigma{\restriction_V}$ the substitution obtained from σ by restricting its domain to V. Moreover, given a term t, we abuse notation by denoting $\sigma{\restriction_{var(t)}}$ simply as $\sigma{\restriction_t}$. The composition $\theta \circ \sigma$ of θ and σ is the substitution s.t. $(\theta \circ \sigma)(x) = \theta(\sigma(x))$ for any $x \in \mathcal{V}$. We use the notation $\sigma\theta$ to indicate $\theta \circ \sigma$. Given two substitutions ϑ_1 and ϑ_2 and two terms t and s, we say that ϑ_1 (respectively t) is more general then ϑ_2 (respectively s), namely $\vartheta_1 \preceq \vartheta_2$ (respectively $t \preceq s$) if and only if there exists a substitution σ s.t. $\vartheta_1\sigma = \vartheta_2$ (respectively $t\sigma =$ s). We denote by $\simeq$ the induced equivalence, i.e., $\theta \simeq \vartheta$ if and only if there exists a renaming ρ s.t. $\theta\rho = \sigma$ (and $\sigma\rho^{-1} = \theta$).

An equation is an expression of the form $t = s$ where t and s are terms. A set of equations E is *unifiable* if and only if there exists a substitution θ s.t. for all $t = s$ in E, $t\theta \equiv s\theta$ holds, θ is said an *unifier* for E. A unifier σ for E is the *most general* (m.g.u.) if and only if $\sigma \preceq \theta$ for any unifier θ of E. Two terms t and s unify if and only if $\{t = s\}$ unify and $mgu(t, s)$ indicates its most general unifier.

With $s \ll X$ we denote a *fresh* variant s of a set of syntactic objects X, i.e., a renaming of an $x \in X$ that contains no variable previously met during computation (standardized apart).

We will use the notation $\sigma \uparrow \sigma'$ of [79] to indicate the l.u.b. (w.r.t. $\preceq$) of σ and σ' ($\sigma \uparrow \sigma' = \sigma\, mgu(\sigma, \sigma') = \sigma'\, mgu(\sigma, \sigma')$). We assume that the reader has familiarity with the algebraic properties of idempotent substitutions described in [79].

Term graphs and homomorphisms

Since we want to devise a semantics compliant with *call-time choice* in presence of sharing we need indeed to express and work explicitly with sharing. Thus we cannot work on terms that are simply trees, but we need DAGs. Now we introduce a formalism to work on *term graphs* [37] so we can give, in Subsection 2.1.1, an explicit formal definition of the small-step operational semantics of Curry in presence of sharing.

Many different notations are considered in the literature about graph rewriting [38, 90], we are consistent with [37].

We consider a graph as a set of nodes and edges between the nodes. Each node is labelled with a symbol in Σ or a variable in $\mathcal{V}$. We indicate with $\mathcal{N}$ the countable set of *nodes*. We assume that $\mathcal{N}$ and $\mathcal{V}$ are fixed throughout the rest of the thesis.

A graph g over $\langle \Sigma, \mathcal{N}, \mathcal{V} \rangle$ is a tuple $g = \langle \mathcal{N}_g, \mathcal{L}_g, \mathcal{S}_g, \mathcal{R}oot_g \rangle$ such that

- $\mathcal{N}_g$ is a set of nodes;

- $\mathcal{L}_g \colon \mathcal{N}_g \to \Sigma \cup \mathcal{V}$ is a *labeling function*, which maps to every node of g a symbol or a variable;
- $\mathcal{S}_g$ is a successor function, which maps to every node of g a (possibly empty) string of nodes, and
- $\mathcal{R}oot_g$ is a subset of $\mathcal{N}_g$ called *root nodes* of g.

We also assume two conditions of well definedness.

1. Graphs are connected, i.e., for all nodes $p \in \mathcal{N}_g$, there exists a root $r \in \mathcal{R}oot_g$ and a path from r to p.
2. Let $var(g)$ be the set of variables of g, for all $x \in var(g)$, there are no other nodes in $\mathcal{N}_g$ labelled with the same x.

In [37] the authors consider possibly cyclic term graphs, namely a subclass of term graphs they called *admissible term graphs*. In contrast in this thesis a *term graph* (or just term when no confusion can arise) is just a (possibly infinite) acyclic graph with one root denoted $\mathcal{R}oot_g$.

This is not a restriction since we will consider term graphs up to bisimilarity and an admissible term graph with cycles is bisimilar to an infinite acyclic graph. Two term graphs g_1 and g_2 are *bisimilar* if and only if they represent the same (infinite) tree when one unravels them. With $\wr t \int$ we denote the unraveling of a term graph t.

Given a graph g, and two nodes in $p, q \in \mathcal{N}_g$ we say that, q is reachable from p in g, written $p \twoheadrightarrow_g q$ if the successor function $\mathcal{S}_g$ induces a path from p to q, formally there exists a sequence $n_1, \ldots, n_k$ of nodes in $\mathcal{N}_g$ such that, for every $1 \leq i < k$, n_{i+1} occurs in $\mathcal{S}_g(n_i)$, where $n_0 = p$ and $n_k = q$. In contrast to the classical term case there could be more than one path between two nodes, while in contrast to [37] the relation $\twoheadrightarrow_g$ is a partial order, since we are considering only the class of acyclic graphs. A path $p_1, \ldots, p_n$ in a graph g is said a *constructor (respectively operator) path* if for every $1 \leq i \leq n$, p_i is labeled with a constructor (respectively operator) symbol.

A *subgraph* of a graph g rooted by a node p, denoted $g|_p$, is built by considering p as root and deleting all the nodes which are not reachable from p in g. An *outermost op-rooted subgraph* (respectively a *critical variable*) of a graph g is an op-rooted (respectively variable) subgraph which can be reached by a constructor path that starts from $\mathcal{R}oot_g$.

Two term graphs g_1 and g_2 are said *compatible* if every node shared by g_1 and g_2 has the same label and the same successors, and every variable occurring both in g_1 and g_2 labels the same node. The *sum* of two compatible graphs, denoted $g_1 \oplus g_2$, is the graph whose nodes and roots are those of g_1 and g_2. The *replacement* by a term graph s of the subgraph rooted by a node p in a term graph g, denoted $g[s]_p$, is built in three steps:

1. compute $g \oplus s$,
2. redirect all the edges pointing on p to point on $\mathcal{R}oot_s$ in $g \oplus s$ and
3. delete all the "uninteresting" nodes that are those which are no more reachable from $\mathcal{R}oot_g$.

A (rooted) *homomorphism* h from a graph g_1 to a graph g_2, denoted $h\colon g_1 \to g_2$, is a mapping from $\mathcal{N}_{g_1}$ to $\mathcal{N}_{g_2}$ such that $\mathcal{R}oot_{g_2} = h(\mathcal{R}oot_{g_1})$ and for all nodes $p \in \mathcal{N}_{g_1}$, if $\mathcal{L}_{g_1}(p) \notin \mathcal{V}$ then $\mathcal{L}_{g_2}(h(p)) = \mathcal{L}_{g_1}(p)$ and $\mathcal{S}_{g_2}(h(p)) = h(\mathcal{S}_{g_1}(p))$; if $\mathcal{L}_{g_1}(p) \in \mathcal{V}$ then $h(p) \in \mathcal{N}_{g_2}$. If $h\colon g_1 \to g_2$ is a homomorphism and g is a subgraph of g_1 rooted in p, then we write $h[g]$ for the subgraph $g_2|_{h(p)}$. If $h\colon g_1 \to g_2$ is a homomorphism and g is a graph, $h[g]$ is the graph built from g by replacing all the subgraphs shared between g and g_1 by their corresponding subgraphs in g_2. [36] proves that there exists a homomorphism $h'\colon g \to h[g]$, h' is called the *extension* of h to g.

Given a set of nodes N and a homomorphism $h\colon g_1 \to g_2$, then h is said to be N-preserving if for every $p \in N$, $h(p) = p$; analogously, given a set of labels L, h is said L-preserving if for every $p \in \mathcal{N}_{g_1}$, if $\mathcal{L}_{g_1}(p) \in L$ then $h(p) = p$.

In the rest of the thesis, with an abuse of notation we use the same name to denote a homomorphism $h\colon g_1 \to g_2$ and its possible extensions. Moreover, if not explicitly stated otherwise, we will implicitly refer to the extension of h which is N-preserving for the set of nodes N that doesn't belong to $\mathcal{N}_{g_1}$.

A term graph l matches a graph g at node p if there exists a homomorphism $h\colon l \to g|_p$. h is called a *matcher* of l on g at node p. Two term graphs g_1 and g_2 are *unifiable* if there exist two graphs G and H and a homomorphism $h\colon G \to H$ such that

1. g_1 and g_2 are both subgraphs of G and
2. $h[g_1] = h[g_2]$.

h is called *unifier* of g_1 and g_2. If g_1 and g_2 are unifiable, it has been shown in [37] that there exists a *most general unifier* $h\colon g_1 \oplus g_2 \to g$ such that $h[g_1] = h[g_2] = g$ and for all unifiers $h'\colon G \to H$, there exists a homomorphism $h''\colon g \to h'[g_1 \oplus g_2]$.

Independently of homomorphisms, we need substitutions in order to define solutions computed by narrowing. In the term graph case a term *substitution* is a partial function from variables to the set of graph terms. Given a homomorphism $h\colon g_1 \to g_2$, we indicate with σ_h the substitution such that for every $x \in var(g_1)$ which labels some node $p \in \mathcal{N}_{g_1}$, $x\sigma_h = g_2|_{h(p)}$.

All concepts related to substitutions of standard terms (domain, range, restriction, ...) have a straightforward generalization to the term graph case. We will abuse notations about standard terms for term graphs as well.

1.4.2 Rewriting

A set of rewrite rules $l \to r$ s.t. $l \notin \mathcal{V}$ is called a *term rewriting system* [4] (TRS). The terms l and r are called the *left-hand side (LHS)* and the *right-hand side (RHS)* of the rule, respectively. A TRS is *left-linear* if every LHS of its rules is linear. Two, possibly renamed, rules $l \to r$ and $l' \to r'$ *overlap* if there is a non-variable position p in l such that $l|_p$ and l' unify.

A TRS is *constructor based* (CB) if each LHS is a pattern. In the remainder of this thesis, a *(first order) functional logic program* P is a left-linear constructor based TRS where each rule is well-typed. By $\mathbb{P}_\Sigma$ we denote the set of functional logic programs over signature Σ.

[4] Note that in literature, $l \to r$ is a rewrite rule if $l \notin \mathcal{V}$ and $var(r) \subseteq var(l)$. In this thesis we allow *extra (or logical) variables* occur in the RHS of a rule.

A rewrite step $t \xrightarrow[l \to r]{p,\eta} t'$ w.r.t. a given TRS P is defined if there are a position p in t, $l \to r \ll P$ and a substitution η with $dom(\eta) \subseteq var(l)$ and $t|_p \equiv l\eta$ such that $t' \equiv t[r\eta]_p$. $t|_p$ is called redex. A redex is outermost if it is not a subterm of another redex. As usual we use $\to^*$ to denote the transitive and reflexive closure of the rewriting relation $\to$. A term t is called a *normal form* if there is no term s such that $t \to s$, while it is an *head normal form* (HNF) if it has not a defined symbol in its root (i.e., has a constructor or a variable).

1.4.3 Needed Narrowing (without sharing)

The combination of variable instantiation and rewriting is called *narrowing*. Formally, $t \overset{p,\sigma}{\underset{l \to r}{\leadsto}} t'$ is a narrowing step if $t\sigma \xrightarrow[l \to r]{p,\eta} t'$ were σ is a substitution, $t|_p \notin \mathcal{V}$ and $dom(\eta) \subseteq var(l)$. The requirement that $dom(\eta) \subseteq var(l)$ ensures that no extra variable in a rule is instantiated during a narrowing step.

Needed narrowing [6] aims at performing only those narrowing steps which are really needed for reaching a value (i.e., a constructor term). Its definition is based on the concept of *Definitional Trees*, a hierarchical structure containing all the rules of a defined function.

Definition 1.4.1 (Definitional Tree [51, App. D.1]) *T is a* definitional tree with call pattern π *if and only if T is finite and one of the following cases hold:*

- $T = rule(l \to r)$, *where* $l \to r \ll P$ *such that* $l = \pi$.
- $T = branch(\pi, p, T_1, \ldots, T_k)$, *where* $k > 0$, $\pi|_p \in \mathcal{V}$, $c_1, \ldots, c_k$ *are different constructor symbols of the sort of* $\pi|_p$ *and, for all* $i = 1, \ldots, k$, T_i *is a definitional tree with call pattern* $\pi[c_i(x_1, \ldots, x_{n_i})]_p$ *where* n_i *is the arity of* c_i *and* $x_1, \ldots, x_{n_i}$ *are fresh variables.*
- $T = or(T_1, T_2)$, *where* T_1 *and* T_2 *are definitional trees with pattern* π.

A definitional tree of an n-ary function f *is a definitional tree T with call pattern* $f(\overrightarrow{x_n})$, *where* $\overrightarrow{x_n}$ *are distinct variables, such that for each rule* $l \to r$ *with* $l = f(\overrightarrow{t_n})$ *there is a node* $rule(l' \to r')$ *in T with* $l' \to r'$ *variant of* $l \to r$.

In the rest of this thesis we indicate by $pat(T)$ *the call pattern of* T.

It is worth to mention that for every $P \in \mathbb{P}_\Sigma$ there exists at least a definitional tree of each specific operator symbol $f \in \mathcal{D}$ and, in general, there could be more than one [5].

Needed narrowing chooses the actual positions to perform a narrowing step, starting from some outermost op-rooted term, in accord to definitional trees. Moreover substitutions are not just MGUs between redex and rule head but depend also upon the initial outermost op-rooted term. The formal definition is not relevant for our thesis, for details consult [6].

[5] Curry considers only definitional trees generated by the algorithm shown in [51, App. D.5]

1.5 Abstract Interpretation

Abstract interpretation [29, 31] is a general theory for approximating the semantics of discrete dynamic systems, originally developed by Patrick and Radhia Cousot, in the late 70's, as a unifying framework for specifying and validating static program analyses. The *abstract semantics* is an approximation of the concrete one, where exact (concrete) values are replaced by approximated (abstract) values, modelled by an abstract domain that explicitly exhibits a structure (e.g. ordering) which is somehow present in the richer concrete structure associated to program execution.

1.5.1 Closures on Complete Lattices

Closures play a fundamental role in semantics and approximation theory [31]. In the following we recall some basic notions on closure theory that will be useful throughout the thesis. For a more complete treatment of the subject see [12, 30]. A *closure operator* on a complete lattice $(L, \leq)$ is an operator $\rho\colon L \to L$ such that, for each $x, y \in L$,

$$x \leq \rho(x) \qquad \text{(extensivity)}$$
$$x \leq y \Longrightarrow \rho(x) \leq \rho(y) \qquad \text{(monotonicity)}$$
$$\rho(\rho(x)) = \rho(x) \qquad \text{(idempotence)}$$

Let $(L, \leq)$ be a complete lattice. In the following we enumerate some basic properties of closure operators on L. Let ρ be an upper closure operator on $(L, \leq)$.

- For all $x \in L$, the set $\{y \in \rho(L) \,|\, x \leq y\}$ is not empty and $\rho(x)$ is the least element.
- The image $R := \rho(L)$ of L by ρ is a complete lattice $(R, \leq)$, such that $\bigsqcup_R(X) = \rho(\bigsqcup_L(X))$ and $\bigsqcap_R(X) = \bigsqcap_L(X)$.
- ρ is a *quasi-complete-join-morphism*. Namely, for each $X \subseteq L$, $\rho(\bigsqcup(X)) = \rho(\bigsqcup(\rho(X)))$.
- Let $R \subseteq L$ and $\rho\colon L \to R$ such that, for any $x \in L$, $\rho(x)$ is the least element in $\{y \in R \,|\, x \leq y\}$. Then ρ is an upper closure operator on $(L, \leq)$ and $R := \rho(L)$.
- Let $uco(L)$ be the set of all upper closure operators on L. Then $(uco(L), \preceq)$ is a complete lattice, where $\preceq$ is defined as follows. For each $\rho, \rho' \in uco(L)$,

  $$\rho \preceq \rho' \iff \forall x \in L.\, \rho(x) \leq \rho'(x).$$

1.5.2 Galois Connections

The theory requires the two semantics to be defined on domains which are partially ordered sets. $(\mathbb{C}, \sqsubseteq)$ (the concrete domain) is the domain of the concrete semantics, while $(\mathbb{A}, \leq)$ (the abstract domain) is the domain of the abstract semantics. The partial order relations reflect an approximation relation.

The guiding idea is to relate the concrete and the abstract interpretation of the calculus by a pair of functions, *abstraction* α and *concretization* γ, which form a Galois connection. Galois connections are used to formalize this relation between abstract and concrete meaning of a computation. This notion has been introduced in [77] to discuss a general

type of correspondence between structures occurring in a great variety of mathematical theories.

Galois connections can be defined on partially ordered sets. However in this thesis we restrict our attention to complete lattices, where they enjoy stronger properties. Since in approximation theory a partial order specifies the precision degree of any element in a poset, it is obvious to assume that if α is a mapping associating an abstract object in $(\mathbb{A}, \leq)$ for any concrete element in $(\mathbb{C}, \sqsubseteq)$ then the following holds: if $\alpha(x) \leq y$, then y is also a correct, although less precise, abstract approximation of x. The same argument holds if $x \sqsubseteq \gamma(y)$. Then y is also a correct approximation of x, although x provides more accurate information than $\gamma(y)$. This gives rise to the following formal definition.

Definition 1.5.1 (Galois Insertion) *Let* $(\mathbb{C}, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ *and* $(\mathbb{A}, \leq, \bigvee, \bigwedge, \bot, \top)$ *be two complete lattices. A* Galois Connection $(\mathbb{C}, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathbb{A}, \leq)$ *is a pair of maps* $\alpha : \mathbb{C} \to \mathbb{A}$ *and* $\gamma : \mathbb{A} \to \mathbb{C}$ *such that, for each* $x \in \mathbb{C}$ *and* $y \in \mathbb{A}$,

$$\alpha(x) \leq y \iff x \sqsubseteq \gamma(y) \tag{1.5.1}$$

Moreover, a Galois Insertion *(of* $\mathbb{A}$ *in* $\mathbb{C}$*)* $(\mathbb{C}, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows\!\!\!\!\!\rightarrow}} (\mathbb{A}, \leq)$ *is a Galois connection where* α *is surjective.*

An equivalent definition of Galois Connection is: a pair of maps $\alpha : \mathbb{C} \to \mathbb{A}$ and $\gamma : \mathbb{A} \to \mathbb{C}$ such that

1. α and γ are monotonic,
2. for each $x \in \mathbb{C}$, $x \sqsubseteq (\gamma \circ \alpha)(x)$ and
3. for each $y \in \mathbb{A}$, $(\alpha \circ \gamma)(y) \leq y$.

Property 2 is called *extensivity* of $\gamma\alpha$ while Property 3 is called *reductivity* of $\alpha\gamma$.

When, in a Galois connection $\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}$, γ is not injective, several distinct elements of the abstract domain $(\mathbb{A}, \leq)$ have the same meaning (by γ). This is usually considered useless [31]; thus a Galois insertion can always be forced by considering a more concise abstract domain $(\mathbb{A}/_{\overset{\gamma}{\approx}}, \leq/_{\overset{\gamma}{\approx}})$, such that for each $x, y \in A$. $x \overset{\gamma}{\approx} y \iff \gamma(x) = \gamma(y)$.

The following basic properties are satisfied by any Galois connection.

1. γ is injective if and only if α is surjective if and only if $\alpha \circ \gamma = id_{\mathbb{A}}$.
2. α is injective if and only if γ is surjective if and only if $\gamma \circ \alpha = id_{\mathbb{C}}$.
3. α is additive $(\alpha(\bigsqcup X) = \bigvee \alpha(X))$ and γ is co-additive $(\gamma(\bigwedge Y) = \bigsqcap \gamma(Y))$.
4. α and γ uniquely determine each other. Namely,

$$\gamma(y) = \bigsqcup \{x \in \mathbb{C} \mid \alpha(x) \leq y\}, \qquad \alpha(x) = \bigwedge \{y \in \mathbb{A} \mid x \sqsubseteq \gamma(y)\}.$$

5. $\gamma \circ \alpha$ is an upper closure operator in $(\mathbb{C}, \sqsubseteq)$.
6. α is an isomorphism from $(\gamma\alpha)(\mathbb{C})$ to $\mathbb{A}$, having γ as its inverse.

Because of Property 4 the map α (γ) is called the *lower* (*upper*) *adjoint*.

Properties 6 and 5 characterize the ability of Galois insertions to formalize the notion of "machine-representable" abstractions. An abstract domain is isomorphic (up to representation) to an upper closure operator of the concrete domain of the computation $\mathbb{C}$. Thus, in principle, we can handle abstract computations as concrete computations on the complete lattice which is the image of the upper closure operator $\gamma \circ \alpha$. However, machine representable abstractions often result to be more intuitive and provides better experimental results in efficient implementations.

A straightforward consequence of the latter observation is that abstract interpretations can be formalized in a hierarchical framework. Abstract domains can be partially ordered using the ordering on the corresponding closure operators on $\mathbb{C}$. The lattice of abstract interpretations of $\mathbb{C}$ is then the lattice of upper closure operators over $\mathbb{C}$. As observed in [77] the composition of upper closure operators is not (in general) an upper closure operator. However, an abstract domain can be designed by successive approximations. Let ρ be an upper closure operator on $(\mathbb{C}, \sqsubseteq)$ and η be an upper closure operator on $\rho(\mathbb{C})$. Then $\eta \circ \rho$ is an upper closure operator on $(\mathbb{C}, \sqsubseteq)$. In view of the compositional design of abstract interpretations we have that the composition of Galois insertions is a Galois insertion. Several techniques can be used to systematically derive new abstract interpretations from a given set of abstract domains [31, 32]. We do not address these techniques because they are outside the scope of this thesis.

1.5.3 Abstract semantics, correctness and precision

In abstract interpretation based static program analysis we compute an abstract (fixpoint semantics). Given a concrete semantics and a Galois insertion between the concrete and the abstract domain, we want to define an abstract semantics. The theory requires to have a semantic evaluation function $\mathcal{P}[\![P]\!] : \mathbb{C} \to \mathbb{C}$ on a complete lattice $(\mathbb{C}, \sqsubseteq)$, the concrete domain, whose least fixpoint $lfp_{\mathbb{C}}(\mathcal{P}[\![P]\!])$ is the (concrete) semantics of the program P. The class of program properties we want to consider is formalized as a complete lattice $(\mathbb{A}, \leq)$, the abstract domain, related to $(\mathbb{C}, \sqsubseteq)$ by a Galois insertion $(\mathbb{C}, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows\!\!\!\twoheadrightarrow}} (\mathbb{A}, \leq)$.

An abstract semantic function $\mathcal{P}^a[\![P]\!] : \mathbb{A} \to \mathbb{A}$ is *correct* if $\forall x \in \mathbb{C}.\ \mathcal{P}[\![P]\!](x) \sqsubseteq \gamma(\mathcal{P}^a[\![P]\!](\alpha(x)))$. The resulting abstract semantics $lfp_{\mathbb{A}}(\mathcal{P}^a[\![P]\!])$ is a correct approximation of the concrete semantics by construction, i.e., $\alpha(lfp_{\mathbb{C}}(\mathcal{P}[\![P]\!])) \leq lfp_{\mathbb{A}}(\mathcal{P}^a[\![P]\!])$, and no additional "correctness" theorems need to be proved.

Moreover, we can systematically derive from $\mathcal{P}[\![P]\!]$, α and γ a correct abstract semantic evaluation function as $\mathcal{P}^\alpha[\![P]\!] := \alpha \circ \mathcal{P}[\![P]\!] \circ \gamma$. $\mathcal{P}^\alpha[\![P]\!]$ is indeed the *most precise* abstract counterpart of $\mathcal{P}[\![P]\!]$, as for any correct $\mathcal{P}^a[\![P]\!]$, $\mathcal{P}^\alpha[\![P]\!] \leq \mathcal{P}^a[\![P]\!]$. Thus $\mathcal{P}^\alpha[\![P]\!]$ is called the *optimal abstract version* of $\mathcal{P}[\![P]\!]$.

The abstract semantics $lfp_{\mathbb{A}}(\mathcal{P}^\alpha[\![P]\!])$ models a safe approximation of the property of interest: if the property is verified in $lfp_{\mathbb{A}}(\mathcal{P}^\alpha[\![P]\!])$ it will also be verified in $lfp_{\mathbb{C}}(\mathcal{P}[\![P]\!])$. An analysis method based on the computation of the abstract semantics $lfp_{\mathbb{A}}(\mathcal{P}^\alpha[\![P]\!])$ is effective only if the least fixpoint is reached in finitely many iterations, i.e., if the abstract domain is Nötherian. If this is not the case, *widening operators* can be used to ensure the termination. Widening operators [33] give an upper approximation of the least fixpoint and guarantee termination by introducing further approximation.

The framework of abstract interpretation can be useful to study hierarchies of semantics and to reconstruct data-flow analysis methods and type systems. It can be used to prove

the safety of an analysis algorithm. However, it can also be used to systematically derive "optimal" abstract semantics from the abstract domain.

The systematic design aspect can be pushed forward, by using suitable abstract domain design methodologies (e.g. domain refinements) [41, 45, 47], which allow us to systematically improve the precision of the domain.

1.5.4 Correctness and precision of compositional semantics

It is often the case that $\mathcal{P}^{\alpha}[\![P]\!]$ is in turn defined as composition of "primitive" operators. Let $f : \mathbb{C}^n \to \mathbb{C}$ be one such an operator and assume that $\tilde{f}$ is its abstract counterpart. Then $\tilde{f}$ is *(locally) correct* w.r.t. f if $\forall x_1, \ldots, x_n \in \mathbb{C}. f(x_1, \ldots, x_n) \sqsubseteq \gamma(\tilde{f}(\alpha(x_1), \ldots, \alpha(x_n)))$. By replacing all concrete f with the abstract $\tilde{f}$ in the formal definition of $\mathcal{P}^{\alpha}[\![P]\!]$ we obtain the definition of an abstract operator $\mathcal{P}^{a}[\![P]\!]$. The local correctness of all the primitive operators implies the global correctness (the correctness of $\mathcal{P}^{a}[\![P]\!]$). Hence, we can define an abstract semantics by defining locally correct abstract primitive semantic functions. According to the theory, for each operator f, there exists an optimal (most precise) locally correct abstract operator $\tilde{f}$ defined as $\tilde{f}(y_1, \ldots, y_n) = \alpha(f(\gamma(y_1), \ldots, \gamma(y_n)))$. However, the composition of optimal operators is not necessarily optimal.

The abstract operator $\tilde{f}$ is *precise* if $\forall x_1, \ldots, x_n \in \mathbb{C}$

$$\alpha(f(x_1, \ldots, x_n)) = \tilde{f}(\alpha(x_1), \ldots, \alpha(x_n))$$

which is equivalent to

$$\alpha(f(x_1, \ldots, x_n)) = \alpha(f((\gamma \circ \alpha)(x_1), \ldots, (\gamma \circ \alpha)(x_n))).$$

Hence the precision of an optimal abstract operator can be reformulated in terms of properties of α, γ and the corresponding concrete operator. The above definitions are naturally extended to "primitive" semantic operators from $\wp(\mathbb{C})$ to $\mathbb{C}$.

There is not presently an agreement on a name for what we call precision. For instance: [46] calls it full-completeness; [34, 69, 84, 87] use the term *completeness*; while [35] use the term *optimality* for the same notion. We prefer to use the term precision, since completeness may be confused with the completeness of a semantics.

Note that if $\bigsqcup$ is the *lub* operation over $(\mathbb{C}, \sqsubseteq)$ and $\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}$ is a Galois insertion then $\widetilde{\bigsqcup} = \alpha \circ \bigsqcup \circ \gamma$ is the *lub* of $(\mathbb{A}, \leq)$ and is precise, i.e., $\widetilde{\bigsqcup} \circ \alpha = \alpha \circ \bigsqcup$ (which is equivalent to $\alpha \circ \bigsqcup = \alpha \circ \bigsqcup \circ \gamma \circ \alpha$).

2

Small-step semantics

2.1 Small-Step Operational Semantics of Curry and Haskell

2.1.1 Small-Step Operational Semantics of Curry Expressions

A first-order Curry program is a set of constructor based rules. To deal with sharing, similarly to [37][1], we have to put some conditions regarding the graph structure of a single program rule $l \to r$:

1. l and r are term graphs, where l, in particular, is a pattern and
2. l and r share only variable nodes, that is, if $p \in \mathcal{N}_l \cap \mathcal{N}_r$ than $\mathcal{L}_r(p) = \mathcal{L}_l(p) \in \mathcal{V}$ [2].

Needed narrowing [6] implements the, so called, *need-time choice* semantics. However Curry implements what is known as *call-time choice* semantics, where all descendants of a subterm are reduced to the same value in a derivation, according with the semantic models of [50]. In [61, 37] has been shown that the call-time choice semantics is consistent with a lazy evaluation strategy where all descendants of a subterm are shared. According to [51, App. D.4] we need that in an expression *several occurrences of the same variable are always shared*. Thus if, during a call, an argument of a function is instantiated to an expression and this expression is evaluated to some expression s, then all other expressions resulting from instantiating occurrences of the same argument are replaced by the same s. This is necessary for the soundness of the operational semantics in presence of non-deterministic functions. However, unlike [1] we intend to describe the small-step operational semantics in a way which is more close to the official version described in [51, App. D.4].

With the term graph notation and constructions defined in the previous subsections we can give a formal precise definition of the operational semantics of Curry along the lines of that defined in [51, App. D.4]. Actually (as explicitly said in [51]), for the sake of simplicity, *that* description of the operational semantics is based on term rewriting and does not take into account that common subterms are shared. However, since we need to work explicitly with sharing of variables, we use the more complicated framework of graph rewriting and here we write a (fully detailed) formal description of Curry's small-step operational semantics based on term graphs. We also explicitly record outermost op-rooted positions since we will need them in the definition of the abstraction for big-step semantics.

[1] In [37] the authors take in consideration a particular class of programs, called *admissible graph rewriting systems*, which is bigger than the Curry programs class.

[2] Point 2 implies that l and r are compatible term graphs.

Actually, given a Curry program as a set of rules, the (small-step operational) semantics of an expression in is not given just in terms of the rules of the program, but in terms of definitional trees for any $f \in \mathcal{D}$, which are not necessarily unique. In Section 2.2, by an abuse of notation, whenever we will refer to "a program P" we will actually intend *a* set of definitional trees for each $f \in \mathcal{D}$. In examples, we will implicitly use the definitional trees obtained by the algorithm of [51, App. D.5]. In Section 3.1, where results will not depend upon definitional trees we will simply use sets of rules.

The *computation step* relation $Eval[\![t]\!] \overset{p,\sigma}{\Longrightarrow} s$ for a term (graph) t that evaluates the subterm at node p, computing substitution σ, and delivering the term (graph) s, is defined by the following rules.

Computation step for an expression:

$$\frac{Eval[\![t_i]\!] \overset{p,\sigma}{\Longrightarrow} s}{Eval[\![t]\!] \overset{p,\sigma}{\Longrightarrow} t[s]_{p_i}} \text{ if } t \text{ is of the form } c(t_1,\ldots,t_n),\ \mathcal{R}oot_{t_i} = p_i,\ i \in \{1,\ldots,n\} \qquad \text{(Cntx)}$$

$$\frac{Eval[\![t;\ T]\!] \overset{\sigma}{\Rightarrow} s}{Eval[\![t]\!] \overset{\mathcal{R}oot_t,\sigma}{\Longrightarrow} s} \text{ if } t \text{ is of the form } f(t_1,\ldots,t_n),\ T \text{ is a fresh definitional tree for } f \qquad \text{(OuterOp)}$$

Note that we are keeping track of the position of the outermost op-rooted position (even if the actual step is occurring in a subterm).

Computation step for an operation-rooted expression:

$$\frac{}{Eval[\![t;\ rule(l \to r)]\!] \overset{\varepsilon}{\Rightarrow} h[r]} \text{ if } h\colon l \to t \qquad \text{(Rw)}$$

$$\frac{Eval[\![t;\ T_i]\!] \overset{\sigma}{\Rightarrow} s}{Eval[\![t;\ or(T_1,T_2)]\!] \overset{\sigma}{\Rightarrow} s}\ i \in \{1,2\}$$

$$\frac{Eval[\![t;\ T_i]\!] \overset{\sigma}{\Rightarrow} s}{Eval[\![t;\ branch(\pi,q,T_1,\ldots,T_k)]\!] \overset{\sigma}{\Rightarrow} s} \text{ if } h\colon\ pat(T_i) \to t$$

$$\frac{\sigma = \{x/\ pat(T_i)|_q\}}{Eval[\![t;\ branch(\pi,q,T_1,\ldots,T_k)]\!] \overset{\sigma}{\Rightarrow} \sigma(t)} \text{ if } h\colon \pi \to t,\ t|_{h(q)} = x \in \mathcal{V},\ \forall i \in \{1,\ldots,k\} \qquad \text{(Ins)}$$

$$\frac{Eval[\![t|_{h(q)}]\!] \overset{h(q),\sigma}{\Longrightarrow} s}{Eval[\![t;\ branch(\pi,q,T_1,\ldots,T_k)]\!] \overset{\sigma}{\Rightarrow} t[s]_{h(q)}} \text{ if } h\colon \pi \to t \text{ and } \mathcal{L}_t(h(q)) \in \mathcal{D} \qquad \text{(Dem)}$$

We call *pure rewriting steps* those corresponding to (Rw) while we call *instantiation steps* those corresponding to (Ins). During an admissible evaluation sequence, according to the Definitional Tree, several instantiation steps take place until a (sub)term becomes a redex and then a pure rewriting step takes place. The "composition" of all these instantiation steps with the rewriting step correspond exactly to a needed narrowing step.

2.1.2 Small-Step Operational Semantics of Haskell Expressions

Now we give a formal precise definition of the operational semantics of Haskell along the lines of that defined in [48]. As done for Curry, we work explicitly with sharing of variables, using the framework of graph rewriting and here we write a (fully detailed) formal description of Haskell's small-step operational semantics based on term graphs. Accordingly to

Curry's small-step operational semantics, we also explicitly record outermost op-rooted positions which will be useful for the definition of the abstraction for big-step semantics.

Since Haskell considers the order of the program's equations[3], a program P has to be formalized as a list of rules. Thus, in the following, by abuse of notation, whenever we refer to a Haskell program P we will implicitly intend it as an ordered list of rules. We will use Haskell syntax for lists to denote a list of rules.

Before presenting the Haskell's *computation step* relation we need the notion of *feasible rule*.

Definition 2.1.1 (feasible rule) *For an op-rooted ground term $e = f(e_1, \ldots, e_n)$ and a rule $R\colon l \to r$, we say that R is* feasible *for e if R defines f and either*

1. *l matches e, or*
2. *there exists a position p where $l|_p$ is constructor rooted whereas $e|_p$ is not* [4].

The computation step relation $HEval[\![e]\!] \overset{p}{\Rightarrow} e'$ for a term (graph) e that evaluates the subterm at node p delivering the term (graph) e', is defined as the smallest relation satisfying the following rules:

Computation step for an expression:

$$\frac{HEval[\![e; Q]\!] \Rightarrow s}{HEval[\![e]\!] \overset{\mathcal{R}oot_e}{\Longrightarrow} s} \begin{array}{l}\text{if } e \text{ is of the form } f(t_1, \ldots, t_n)\\ Q \text{ rules defining } f \text{ in } P\end{array} \quad \text{(H-OuterOp)}$$

$$\frac{HEval[\![e_i]\!] \overset{p}{\Rightarrow} s}{HEval[\![e]\!] \overset{p}{\Rightarrow} e[s]_{p_i}} \begin{array}{l}\text{if } e \text{ is of the form } c(e_1, \ldots, e_n)\\ \mathcal{R}oot_{e_i} = p_i,\ i \in \{1, \ldots, n\}\end{array} \quad \text{(H-Cntx)}$$

Note that we are keeping track of the position of the outermost op-rooted position as done in Subsection 2.1.1 for Curry.

Computation step for an operation-rooted expression:

$$\frac{}{HEval[\![e; l \to r : P]\!] \Rightarrow h[r]}\ h\colon l \to e \quad \text{(H-Rew)}$$

$$\frac{HEval[\![e|_p]\!] \Rightarrow s}{HEval[\![e; l \to r : P]\!] \Rightarrow e[s]_p} \begin{array}{l}p \text{ leftmost outermost position s.t.}\\ l|_p \text{ is } \mathcal{C}\text{-rooted and } e|_p \text{ is not } \mathcal{C}\text{-rooted}\end{array} \quad \text{(H-Dem)}$$

$$\frac{HEval[\![e; P]\!] \Rightarrow e'}{HEval[\![e; R : P]\!] \Rightarrow e'}\ R \text{ unfeasible rule for } e$$

The computation step relation for an op-rooted expression scans the program looking for the first feasible rule. Once the first feasible rule $l \to r$ is found, either the rule can directly rewrite (H-Rew), or there is (at least) a subterm which has to be evaluated first, and (H-Dem) evaluates the leftmost outermost subterm which is demanded in the sense of Point 2 of Definition 2.1.1.

Note that this is no ordinary leftmost outermost evaluation strategy.

[3] The rule selection strategy of Haskell is fundamental to ensure confluence in presence of overlapping rules.

[4] Haskell evaluations can be made only over ground terms, thus $e|_p$ must be is op-rooted

2.2 Modeling the small-step behaviour of Curry: Small-step Tree Semantics

In this section we introduce the concrete small-step semantics of our framework, which is suitable to *model correctly* the typical cogent features of modern functional-logic languages (like Curry [51] or Toy [65]):

- permit non-confluent and non-terminating programs that define non-deterministic non-strict functions;
- the call-time choice behaviour (where the values of the arguments of an operation are determined before the operation is evaluated)

Let us formalize the notion of "model correctly". Correctness of a semantics has to be referred to a criterion of observability for the computations of admissible expressions (in a given specific program). The collection of (fully detailed) computations for expressions, frequently called *concrete behaviour*, is mostly expressed in terms of traces of (official) small-step operational semantics. It can sometimes be expressed in even more concrete terms, for instance traces of an abstract machine implementing the language. Criteria of observability can then be expressed in term of functions over the concrete behaviour which "filter out" the information of interest. These functions are called *observables* (e.g. [63]) or *observable properties*. The application of the observable property to the concrete behaviour is called observation; we prefer the term *observable behaviour* or simply *behaviour*.

To sum up, the (observable) behaviour is a property of interest that can be actually observed about an expression which is solved by an abstract executor.

A semantics $\mathcal{S}$ *models correctly* a behaviour $\mathcal{B}$, or (equivalently) semantics $\mathcal{S}$ is *correct* w.r.t. the program equivalence induced by behaviour $\mathcal{B}$, if

$$\mathcal{S}[\![P_1]\!] = \mathcal{S}[\![P_2]\!] \implies \mathcal{B}[\![P_1]\!] = \mathcal{B}[\![P_2]\!]$$

Equivalently, the program equivalence induced by the semantics is not coarser than the program equivalence relation induced by the behaviour. When the two program equivalences coincide $\mathcal{S}$ is *fully abstract* w.r.t. $\mathcal{B}$.

In Subsection 2.2.1 we define as reference concrete behaviour the small-step (operational) behaviour $\mathcal{B}^{ss}[\![P]\!]$ of a program $P \in \mathbb{P}_\Sigma$ as the collection all the (substantive) sequences of computations steps for all possible initial expressions.

Actually the aim of our work is a little broader than just guaranteeing semantics correctness w.r.t. $\mathcal{B}^{ss}$. Specifically our goal is:

1. defining a denotation $\mathcal{S}$ for Curry programs which
   - is *fully abstract* w.r.t. the small-step behaviour of evaluation of expressions, i.e., $\mathcal{S}[\![P_1]\!] = \mathcal{S}[\![P_2]\!] \iff \mathcal{B}^{ss}[\![P_1]\!] = \mathcal{B}^{ss}[\![P_2]\!]$
   - has a goal-independent definition
   - is the fixpoint of a bottom-up construction
2. defining an abstraction α hiding all irrelevant intermediate steps of a small-step computation in order to have a "big-step" behaviour of evaluation of expressions (essentially the outcomes of small-step computations, given by constructor forms reached by computations).

3. (by means of the same α) obtain an abstract denotation (for programs) $\mathcal{S}^\alpha$ which

   - is fully abstract w.r.t. the big-step behaviour, i.e.,

     $$\mathcal{S}^\alpha[\![P_1]\!] = \mathcal{S}^\alpha[\![P_2]\!] \iff \alpha(\mathcal{B}^{ss}[\![P_1]\!]) = \alpha(\mathcal{B}^{ss}[\![P_2]\!])$$

   - has a goal-independent definition
   - is the fixpoint of a bottom-up construction
   - is as condensed as possible.

   These requirements, especially the condensedness, are *particularly* relevant when the semantics is to be employed to make abstract (approximate) versions via Abstract Interpretation, since it involves using the join operation of the abstract domain at each iteration in parallel onto all components of rules instead of using several subsequent applications for all components. This has a twofold benefit. On one side, it speeds up convergence of the abstract fixpoint computation. On the other side, it considerably improves precision.

In the following of this section we will obtain a semantics with the characteristics of Point 1 in several steps. Essentially we will successively transform the behaviour $\mathcal{B}^{ss}$ by removing the redundancy and (simultaneously) will define the semantics operations that can reconstruct the redundant variants. Section 3.1 will tackle Points 2 and 3.

We prefer to follow a gradual approach since:

- It is easier to understand the rationale of the resulting semantics.
- We derive several interesting properties along the way (that are needed to prove correctness).
- We will (re)use in the future the (small-step) concrete semantics $\mathcal{S}$ to synthesize (with a different abstraction Φ) a semantics $\mathcal{S}^\Phi$ to model "functional dependencies" of computed results. This semantics is needed in order to tackle pre-post conditions and then be able to define Abstract Verification for Curry, along the lines of [24].
- It is actually the road that we followed to reach the result, after that all previous direct approaches that we tried did fail[5].

2.2.1 The small-step behaviour of Curry

As anticipated, we will define the small-step (operational) behaviour $\mathcal{B}^{ss}[\![P]\!]$ of a set of definitional trees P, w.r.t. the needed narrowing strategy with call-time choice semantics, as the collection all the sequences of computations steps for all possible initial expressions (Definition 2.2.6). We need first to introduce some notions before we can state its formal definition.

[5]This is another interesting outcome of the Abstract Interpretation approach. We actually obtained the desired semantics (a precise one, not an approximation) by abstraction of a more concrete one, after all previous attempts to achieve the same goal by direct definition did fail, essentially because it is not at all evident which is the (minimal) information that has to be kept into denotations.

Definition 2.2.1 *Given t, s two terms, σ an idempotent $\mathcal{C}$-linear substitution and p a position of an outermost op-rooted subterm of t, we have a* small-step $t \overset{\sigma}{\underset{p}{\Longrightarrow}} s$ *if*

1. *either $\sigma = \varepsilon$ and there exists u s.t. $s \equiv t[u]_p$, or*
2. *$\sigma = \{x/c(\overrightarrow{y_n})\}$ where $\overrightarrow{y_n} \in \mathcal{V}$ and $s \equiv t\sigma$.*

A sequence $t_0 \overset{\sigma_1}{\underset{p_1}{\Longrightarrow}} t_1 \overset{\sigma_2}{\underset{p_2}{\Longrightarrow}} \dots \overset{\sigma_n}{\underset{p_n}{\Longrightarrow}} t_n$ is called small-step sequence.

$t_0 \overset{\sigma}{\Longrightarrow}^* t_n$ denotes the existence of a small-step sequence where $\sigma = (\sigma_1 \cdots \sigma_n)\restriction_{t_0}$. σ is the computed answer. *When all p_i are identical to p we write $t_0 \overset{\sigma}{\underset{p}{\Longrightarrow}}^* t_n$.*

We use as label of an edge $t \overset{\sigma}{\underset{p}{\Rightarrow}} s$ the same elements of a Curry evaluation step $Eval[\![t]\!] \overset{p,\sigma}{\Longrightarrow} s$ and the requirements on σ and s comes from rules (Rw) and (Ins) (page 24) corresponding to pure rewriting and instantiation steps.

The collection of all the possible evolutions of computations (which are small-step sequences obeying the rules of the operational semantics) for all initial expressions is the *full small-step behaviour*. Namely,

Definition 2.2.2 (full small-step behaviour of programs) *Let $P \in \mathbb{P}_\Sigma$. Then*

$$\mathcal{B}^{fss}[\![P]\!] := \left\{ e_0 \overset{\sigma_1}{\underset{p_1}{\Longrightarrow}} e_1 \overset{\sigma_2}{\underset{p_2}{\Longrightarrow}} \dots \;\middle|\; e_0 \in \mathcal{T}(\Sigma, \mathcal{V}), \forall i.\ Eval[\![e_i]\!] \overset{p_i,\sigma_i}{\Longrightarrow} e_{i+1} \right\}$$

We will call derivation (sequence) of e *any small-step sequence $e \overset{\sigma}{\Longrightarrow}^* e' \in \mathcal{B}^{fss}[\![P]\!]$. When $e' \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ the sequence is called* succeeding derivation, e' computed value *and the pair $\sigma \cdot e'$* computed result[6].

We indicate with $\approx_{fss}$ the program equivalence relation induced by $\mathcal{B}^{fss}$, namely $P_1 \approx_{fss} P_2 \Leftrightarrow \mathcal{B}^{fss}[\![P_1]\!] = \mathcal{B}^{fss}[\![P_2]\!]$.

Multiple outermost op-rooted subterms

The official small-step operational semantics definition does not specify that any particular selection rule has to be adopted, and is left to implementation to choose one. We can easily prove that any specific choice is actually irrelevant.

Let $\approx^{\lambda}_{ss}$ be the equivalence induced by the small-step behaviour considering only sequences according to a selection rule λ.

Lemma 2.2.3 *Let λ be a selection rule and $P_1, P_2 \in \mathbb{P}_\Sigma$. $P_1 \approx_{fss} P_2$ if and only if $P_1 \approx^{\lambda}_{fss} P_2$.*

[6] The term derivation comes from the LP tradition. In the FLP community *computed answer* and *computed value* are standard names. In some literature on FL programming, the term "solution" is used for referring to a pair of answer and result, e.g. in [60]. However, to the best of our knowledge, there is no well established name for such pairs, thus we chose *computed result*.

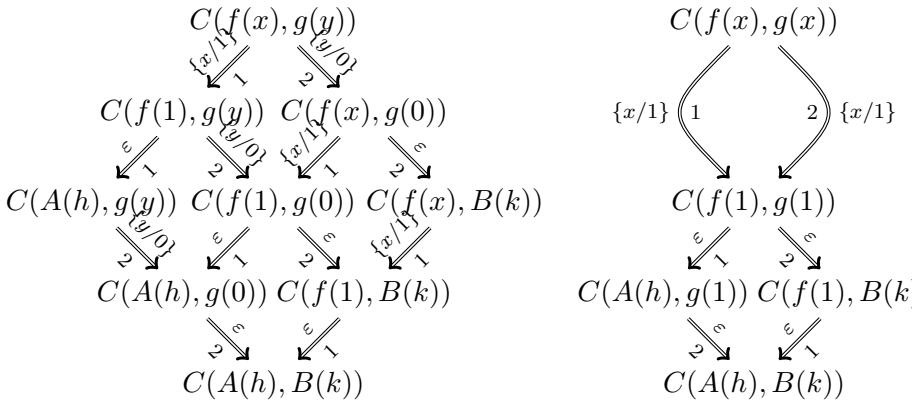


Figure 2.1: Small-step derivations for $C(f(x), g(y))$ and $C(f(x), g(x))$.

Canonical (sequential) small-step derivations

There is an important property about small-step derivations that is crucial for the (compact and easier) definition of the big-step abstraction. Even when we start with a term having a single op-rooted subterm, during execution it in not unlikely that multiple outermost op-rooted subterms have to be reduced. The small-step operational semantics definition allows any sort of possible interleaving in their reduction. Regardless of all actual choices, the computed result are ensured to be the same. However there is an even stronger property for all intermediate steps (which we rely on in the following) that we are now going to evidence.

As a concrete example to start the discussion consider the program $P := \{f(1) \to A(h),\ g(1) \to B(k),\ g(0) \to B(k),\ h \to \cdots,\ k \to \cdots\}$. In Figure 2.1 are shown all the derivations for the goal $e = C(f(x), g(y))$. One can easily verify that, for all possible reducts s of e, all possible different reduction paths from e to s compute the same answer (i.e., the diagrams commute). The two "external" derivations work sequentially first on one outermost op-rooted subterm until it is reduced to a head normal form and then on the other one (until it is reduced to a head normal form too). All other possibilities interleave reductions on the two positions and, when they reach the two head normal forms, their computed answer will be the same of the sequential versions. The same holds when we have shared subterms as one can see for the goal $C(f(x), g(x))$.

In general, for every derivation $e_0 \overset{\vartheta_1}{\underset{p_1}{\Longrightarrow}} \ldots \overset{\vartheta_n}{\underset{p_n}{\Longrightarrow}} e_n$ working "interleavingly" between more than one outermost op-rooted subterm *eventually* reducing the outermost op-rooted subterm of e_0 at position p to a head normal form, there will be another derivation $e_0 = s_0 \overset{\sigma_1}{\underset{q_1}{\Longrightarrow}} \ldots \overset{\sigma_i}{\underset{q_i}{\Longrightarrow}} s_i \overset{\sigma_{i+1}}{\underset{q_{i+1}}{\Longrightarrow}} \ldots \overset{\sigma_n}{\underset{q_n}{\Longrightarrow}} s_n = e_n$ working "sequentially" on p until s_i, where $s_i|_p$ is a head normal form, and than reduces to e_n, with the same computed answer ($\sigma_1 \cdots \sigma_n = \vartheta_1 \cdots \vartheta_n$).

Let us state this formally

Proposition 2.2.4 *Let $P \in \mathbb{P}_\Sigma$ and $e_0 \overset{\vartheta_1}{\underset{p_1}{\Longrightarrow}} \ldots \overset{\vartheta_n}{\underset{p_n}{\Longrightarrow}} e_n \in \mathcal{B}^{fss}[\![P]\!]$ a derivation which eventually reduces an outermost op-rooted subterm at position p of e_0 in head normal form. Then, there exists $s_0 \overset{\sigma_1}{\underset{q_1}{\Longrightarrow}} \ldots \overset{\sigma_i}{\underset{q_i}{\Longrightarrow}} s_i \overset{\sigma_{i+1}}{\underset{q_{i+1}}{\Longrightarrow}} \ldots \overset{\sigma_n}{\underset{q_n}{\Longrightarrow}} s_n \in \mathcal{B}^{ss}[\![P]\!]$ such that*

- $e_0 = s_0$, $e_n = s_n$ *and* $\vartheta_1 \cdots \vartheta_n = \sigma_1 \cdots \sigma_n$,
- $s_i|_p$ *is in head normal form,*
- *for all* $1 \leq j \leq i$, $s_j|_p$ *is not in head normal form and* $q_j = p$, [7]
- *for all* $i < j \leq n$, $q_j \neq p$.

Notation 2.2.5 *In the following we denote with* $s_0 \overset{\sigma}{\underset{p}{\hookrightarrow}} s_i$ *a small-step sequence (not necessarily a derivation)* $s_0 \overset{\sigma_1}{\underset{p}{\Rightarrow}} \ldots \overset{\sigma_i}{\underset{p}{\Rightarrow}} s_i$ *where* $s_i|_p$ *is in head normal form, for all* $1 \leq j \leq i$, $s_j|_p$ *is not in head normal form and* $\sigma = \sigma_1 \cdots \sigma_i$.

Thus Proposition 2.2.4 can be reformulated as "Let ... Then, there exists $e_0 \overset{\sigma}{\underset{p}{\hookrightarrow}} s_i \overset{\eta}{\Rightarrow}{}^* e_n$ ..." where $\sigma = \sigma_1 \cdots \sigma_i$ and $\eta = \sigma_{i+1} \cdots \sigma_n$.

By repeated application of Proposition 2.2.4 for *any* derivation $e_0 \overset{\vartheta_1}{\underset{p_1}{\Rightarrow}} \ldots \overset{\vartheta_n}{\underset{p_n}{\Rightarrow}} e_n$ we have corresponding ones which works contiguously, i.e.,

$$e_0 \overset{\sigma_1}{\underset{p_1}{\hookrightarrow}} u_1 \overset{\sigma_2}{\underset{p_2}{\hookrightarrow}} \ldots \overset{\sigma_k}{\underset{p_k}{\hookrightarrow}} u_k \overset{\eta}{\Rightarrow}{}^* e_n$$

where all $u_i|_{p_i}$ are head normal forms and $\vartheta_1 \cdots \vartheta_n = \sigma_1 \cdots \sigma_k \eta$. We may possibly need a "leftover" tail $u_k \overset{\eta}{\Rightarrow}{}^* e_n$ in case in e_n not all outermost op-rooted subterms of e_0 are reduced in head normal form.

In the following sequences of this form will be said *in canonical form* (or simply *canonical*). Note that canonicity does not imply that the small-steps have to satisfy the $Eval[\![\cdot]\!] \Rightarrow \cdot$ relation.

Definition 2.2.6 (canonical form small-step behaviour of programs) *Let* $P \in \mathbb{P}_\Sigma$,

$$\mathcal{B}^{ss}[\![P]\!] := \left\{ d \;\middle|\; d \in \mathcal{B}^{fss}[\![P]\!], d \text{ canonical} \right\}$$

We indicate with $\approx_{ss}$ *the program equivalence relation induced by* $\mathcal{B}^{ss}$, *namely* $\forall P_1, P_2 \in \mathbb{P}_\Sigma.\ P_1 \approx_{ss} P_2 \Leftrightarrow \mathcal{B}^{ss}[\![P_1]\!] = \mathcal{B}^{ss}[\![P_2]\!]$.

We have that $\mathcal{B}^{fss}[\![P]\!]$ is isomorphic to $\mathcal{B}^{ss}[\![P]\!]$ and thus that the (induced) equivalences $\approx_{fss}$ and $\approx_{ss}$ are identical.

Proposition 2.2.7 *For all* $P \in \mathbb{P}_\Sigma$, $\mathcal{B}^{fss}[\![P]\!]$ *is isomorphic to* $\mathcal{B}^{ss}[\![P]\!]$. *Moreover, for all* $P_1, P_2 \in \mathbb{P}_\Sigma$, $P_1 \approx_{ss} P_2 \Leftrightarrow P_1 \approx_{fss} P_2$.

As a matter of fact, all information that can be derived (and abstracted) from $\mathcal{B}^{fss}$ can be derived (and abstracted) from $\mathcal{B}^{ss}$ as well. Thus we can just use $\mathcal{B}^{ss}$ instead of $\mathcal{B}^{fss}$. One could wander if we can "safely" omit some more redundancy. Actually there can be several canonical alternatives (in $\mathcal{B}^{ss}$) which reduce to the same values/normal forms. Thus, w.r.t. computed results, $\mathcal{B}^{ss}$ has indeed redundant information. However, as will be clear from what follows, we cannot leave out any of the alternatives, because it would be impossible to ensure to be able to (correctly) reconstruct the behaviour of nested expressions from the components' semantics (because of laziness). This is the reason why, in the following, we will use $\mathcal{B}^{ss}$ as *the* reference behaviour for Curry.

[7] Technically node p may no longer exist in s_{i-1}. However any path that reaches p in e_0 is a path that reaches q_j in s_{j-1} and vice versa. The equality and inequality on positions has to be intended in this sense.

2.2.2 The semantic domain

Now that we have settled which is the reference behaviour of Curry (Definition 2.2.6) that we want to model, we start to define our semantics by introducing the semantic domain of "Small-step Trees".

Definition 2.2.8 (Small-step Trees) *Given a term $e \in \mathcal{T}(\Sigma, \mathcal{V})$, a* small-step tree *$T$ for e is a (not necessarily finite) labelled tree, with terms of $\mathcal{T}(\Sigma, \mathcal{V})$ in nodes, rooted in e where*

1. *Paths are canonical small-step sequences*
2. *Sibling subtrees must have different root terms.*

We denote with $\mathbb{SST}_\Sigma$ (or simply $\mathbb{SST}$ when it is clear from the context) the set of all the small-step trees (over Σ).

Moreover, for any $e \in \mathcal{T}(\Sigma, \mathcal{V})$, we denote with $\mathbb{SST}_e$ the set of all small-step trees for e.

Point 2 ensures that all sibling steps in a small-step tree are pairwise distinct and thus that we cannot have two different paths of the tree with the same terms and labels.

Definition 2.2.9 (Variance on $\mathbb{SST}$) *Let $e \in \mathcal{T}(\Sigma, \mathcal{V})$ and $T_1, T_2 \in \mathbb{SST}_e$, then T_1 and T_2 are* variants *if there exists an isomorphism $\iota \colon \mathcal{N} \to \mathcal{N}$ (i.e., renaming of variables and nodes) such that $\iota[T_1] = T_2$ and $\iota[e] = e$* [8].

Intuitively, two small-step trees are variants if and only if they have the same root e and their steps are equal up to renaming of nodes and variables which do not occur in e.

Note 2.2.10 *From now on, with an abuse of notation, by $\mathbb{SST}$ we will actually indicate its quotient w.r.t. variance. Moreover all small-step trees presented in the following will actually be an arbitrary representative of an equivalence class.*

Definition 2.2.11 (Order on $\mathbb{SST}$) *Let denote with $paths(T)$ the set of all the paths in T starting from the root.*

Let T_1 and T_2 be two small-step trees, then $T_1 \sqsubseteq T_2$ if and only if $paths(T_1) \subseteq paths(T_2)$.

Given $\mathcal{T} \subseteq \mathbb{SST}_e$, the least upper bound $\bigsqcup \mathcal{T}$ is the tree whose paths are $\bigcup_{T \in \mathcal{T}} paths(T)$. Dually for the greatest lower bound $\bigsqcap$.

It is worth noticing that, for any $e \in \mathcal{T}(\Sigma, \mathcal{V})$, $\mathbb{SST}_e$ is a complete lattice.

By Point 2 of Definition 2.2.8, *paths* is injective, thus it establishes an order preserving isomorphism $(\mathbb{SST}_e, \sqsubseteq) \underset{paths}{\overset{prfxtree}{\twoheadleftarrow\!\!\twoheadrightarrow}} (paths(\mathbb{SST}_e), \subseteq)$, where the adjoint of *paths*, *prfxtree*, builds a tree from a set of paths by merging all common prefixes.

Moreover, if we denote by $maxpaths(T)$ the set of all the *maximal* paths starting from e, we have another order preserving isomorphism $(\mathbb{SST}_e, \sqsubseteq) \underset{maxpaths}{\overset{prfxtree}{\twoheadleftarrow\!\!\twoheadrightarrow}} (maxpaths(\mathbb{SST}_e), \trianglelefteq)$, where $S \trianglelefteq S'$ if for all $s \in S$ there exists $s' \in S'$ such that s is a prefix of s' (or, equivalently, $prfxtree(S) \sqsubseteq prfxtree(S')$).

[8] Recall that, in order to tackle explicitly the sharing within terms, we use the explicit notation and notions of [37] (introduced in Subsection 1.4.1).

So we have three isomorphic representations of small-step trees and in the following we will use in each specific case the representation that seems to be more convenient (i.e., clearer or smaller). Moreover, for the same reason, we can simply write $d \in T$ for $d \in \mathit{paths}(T)$.

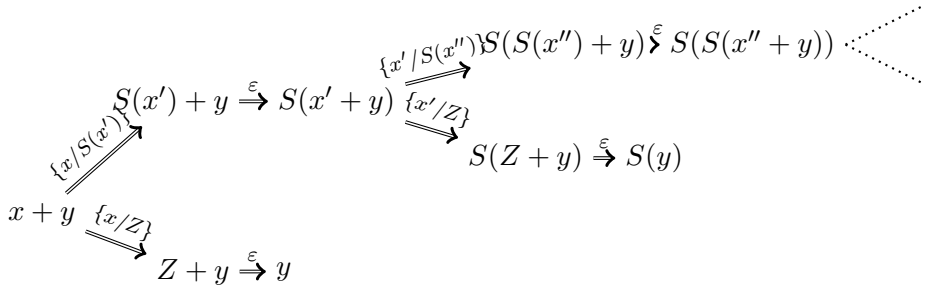
Definition 2.2.12 (Small-step Tree of a term in a program) *Given $e \in \mathcal{T}(\Sigma, \mathcal{V})$ and $P \in \mathbb{P}_\Sigma$, a small-step tree $\mathcal{N}[\![e \mathit{\ in\ } P]\!]$ for the term t in program P is a small-step tree of $\mathbb{SST}_e$ whose steps correspond to the computation steps of e:*

$$\mathcal{N}[\![e \mathit{\ in\ } P]\!] := \{d \mid d \in \mathcal{B}^{ss}[\![P]\!],\ d \textit{ starts from } e\}/_{\cong}$$

Intuitively, $\mathcal{N}[\![e \mathit{\ in\ } P]\!]$ denotes the small-step operational behaviour of t in the program P modulo variance (i.e., local variables are up to renaming).

Example 2.2.13

Given the program $P_+ := \{Z + y \to y,\ S(x) + y \to S(x + y)\}$[9] the small-step tree $\mathcal{N}[\![x + y \mathit{\ in\ } P_+]\!]$ is



In the following we indicate with $\mathcal{N}_k[\![e \mathit{\ in\ } P]\!]$ the operational behaviour of t up to k pure rewriting steps, namely

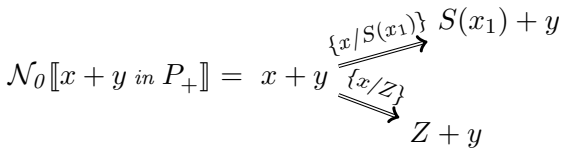
$$\mathcal{N}_k[\![e \mathit{\ in\ } P]\!] := \{d \in \mathcal{N}[\![e \mathit{\ in\ } P]\!] \mid d \text{ has at most } k \text{ pure rewriting steps}\} \tag{2.2.1}$$

It can be easily shown that

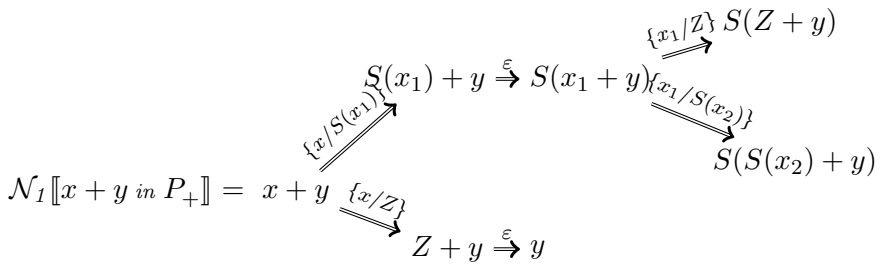
$$\mathcal{N}[\![e \mathit{\ in\ } P]\!] = \bigsqcup\{\mathcal{N}_k[\![e \mathit{\ in\ } P]\!] \mid k \geq 0\}. \tag{2.2.2}$$

Example 2.2.14

For the program P_+ of Example 2.2.13 we have



[9] Recall that in examples we will implicitly use the definitional trees obtained by the algorithm of [51, App. D.5] over programs specified just as sets of rules.



The following result shows that small-step trees "capture" the behaviour of the operational semantics of all expressions, i.e., the induced program equivalences coincide.

Theorem 2.2.15 *Let P_1, $P_2 \in \mathbb{P}_\Sigma$. Then $P_1 \approx_{ss} P_2$ if and only if, for every term e, $\mathcal{N}[\![e \text{ in } P_1]\!] = \mathcal{N}[\![e \text{ in } P_2]\!]$.*

It is important to note that another consequence of the repeated application of Proposition 2.2.4 is that given a canonical sequence that reduces redexes in a certain order, then there will be another (equivalent) canonical sequence reducing the same redexes in any other permutation of the order. Thus trees $\mathcal{N}[\![e \text{ in } P]\!]$ are also closed upon rearrangement of order of reduction. Hence if we collect all (partial) evaluations of all possible expressions for all possible programs

$$\mathbb{WSST}_\Sigma := \{\mathcal{N}_k[\![e \text{ in } P]\!] \mid \forall e \in \mathcal{T}(\Sigma, \mathcal{V}), \forall P \in \mathbb{P}_\Sigma, \forall k \in \mathbb{N}\} \tag{2.2.3}$$

we do not exhaust all $\mathbb{SST}_\Sigma$, because there are several "ill-formed" small-step trees in $\mathbb{SST}_\Sigma$ which cannot correspond to any behaviour of some expression. Trees of $\mathbb{WSST}$ will be called well-formed. For any $e \in \mathcal{T}(\Sigma, \mathcal{V})$, $\mathbb{WSST}_e$ (the subset of $\mathbb{WSST}$ with root e) is a sublattice of $\mathbb{SST}_e$.

$\mathbb{WSST}$ is the semantic domain which we use in the following. From now on we will implicitly consider only well-formed small-step trees, without explicitly mentioning it.

The collection for all $e \in \mathcal{T}(\Sigma, \mathcal{V})$ of $\mathcal{N}[\![e \text{ in } P]\!]$ is essentially a tree representation of the operational behaviour $\mathcal{B}^{ss}[\![P]\!]$, merely identifying all possible operational derivations that chose different fresh variable names while they are developed. Thus it is not the semantics we are looking for, since it still has a goal-dependent definition (it is however a first step toward our goal). Now we should look for a denotation of a program P which could be used to retrieve, with suitable semantic operations, $\mathcal{N}[\![e \text{ in } P]\!]$ for any e.

2.2.3 Operational Denotations of Programs

The operational denotation of a program (function definitions without an initial goal) can be defined as an interpretation giving meaning to defined symbols over (well-formed) small-step trees "modulo variance".

Definition 2.2.16 (Interpretations) *Let $\mathbb{MGC} := \{f(\overrightarrow{x_n}) \mid f/n \in \mathcal{D}, \overrightarrow{x_n}$ are distinct variables$\}$. We call any $\pi \in \mathbb{MGC}$* most general call.

Two functions $I, J \colon \mathbb{MGC} \to \mathbb{WSST}$ are variants*, denoted by $I \cong J$, if for each $\pi \in \mathbb{MGC}$ there exists an isomorphism ι (i.e., a renaming of variables and node names) such that $\iota[I(\pi)] = J(\iota[\pi])$.*

An interpretation *is a function* $\mathcal{I} \colon \mathbb{MGC} \to \mathbb{WSST}$ *modulo variance*[10] *such that, for every* $\pi \in \mathbb{MGC}$, $\mathcal{I}(\pi)$ *is a small-step tree for* π.

The semantic domain $\mathbb{I}_\Sigma$ *(or simply* $\mathbb{I}$ *when clear from the context) is the set of all interpretations ordered by the pointwise extension of* $\sqsubseteq$.

It is important to note that $\mathbb{MGC}$ has the same cardinality of $\mathcal{D}$ (and is thus finite).

In the following, any $\mathcal{I} \in \mathbb{I}_\Sigma$ is implicitly considered as an arbitrary function $\mathbb{MGC} \to \mathbb{WSST}$ obtained by choosing an arbitrary representative of the elements of $\mathcal{I}$ in the equivalence class generated by $\cong$. Actually, in the following, all the operators that we use on $\mathbb{I}_\Sigma$ are also independent of the choice of the representative. Therefore, we can define any operator on $\mathbb{I}_\Sigma$ in terms of its counterpart defined on functions $\mathbb{MGC} \to \mathbb{WSST}$.

Moreover, we also implicitly assume that the application of an interpretation $\mathcal{I}$ to a most general call π, denoted by $\mathcal{I}(\pi)$, is the application $I(\pi)$ of any representative I of $\mathcal{I}$ which is defined exactly on π. For example if $\mathcal{I} = (\lambda f(x,y).\, f(x,y) \xrightarrow[p]{\{x/c(z)\}} c(y,z))/_\cong$ then $\mathcal{I}(f(u,v)) = f(u,v) \xrightarrow[p]{\{u/c(z)\}} c(v,z)$.

The partial order on $\mathbb{I}$ formalizes the evolution of the computation process. $(\mathbb{I}, \sqsubseteq)$ is a complete lattice and its least upper bound and greatest lower bound are the pointwise extension of $\bigsqcup$ and $\bigsqcap$, respectively. In the following we abuse the notations for $\mathbb{WSST}$ for $\mathbb{I}$ as well. The bottom element of $\mathbb{I}$ is $\bot_\mathbb{I} := \lambda\pi.\, \pi$ (for each $\pi \in \mathbb{MGC}$).

Since $\mathbb{MGC}$ is finite, we will often explicitly write interpretations by cases, like

$$\mathcal{I} := \begin{cases} \pi_1 \mapsto T_1 & \mathcal{I}(\pi_1) := T_1 \\ \vdots \quad \text{for} & \quad \vdots \\ \pi_n \mapsto T_n & \mathcal{I}(\pi_n) := T_n \end{cases}$$

While defined symbols have to be interpreted according to program rules, constructor symbols are meant to be interpreted as themselves. In order to treat them as a generic case of function application, we assume that *any* interpretation $\mathcal{I}$ is also implicitly extended on constructors as $\mathcal{I}(c(\overrightarrow{x_n})) := c(\overrightarrow{x_n})$. In the following we will use φ when we refer to a generic symbol either constructor or defined.

Definition 2.2.17 (Operational Denotation of Programs) *Let* $P \in \mathbb{P}_\Sigma$, *then the* operational denotation *of* P *is*

$$\mathcal{O}[\![P]\!] := \big(\lambda f(\overrightarrow{x_n}).\, \mathcal{N}[\![f(\overrightarrow{x_n}) \mathit{\ in\ } P]\!]\big)/_\cong \tag{2.2.4}$$

Intuitively, $\mathcal{O}$ collects the small-step operational behaviour of each most general call $f(\overrightarrow{x_n})$ in P, abstracting from the particular choice of the variable names.

The small-step operational behaviour of any term e can be "reconstructed" from $\mathcal{O}[\![P]\!]$ by means of the following *evaluation function* $\mathcal{E}$.

Evaluation Function

When we have the interpretation with the small-step operational behaviour of a most general call $f(\overrightarrow{x_n})$ we can easily reconstruct the behaviour of any $f(\overrightarrow{t_n})$ for $\overrightarrow{t_n} \in \mathcal{T}(\mathcal{C}, \mathcal{V})$

[10] i.e., a family of elements of $\mathbb{WSST}$, indexed by $\mathbb{MGC}$, modulo variance.

by simply replacing the most general bindings along a derivation by the suitable (more instantiated) versions. However, with general nested expressions things gets more involved. First, because of laziness, the subexpressions need not to be fully evaluated, hence one cannot simply "plug" the derivation corresponding to an argument as a sub-derivation of the main call. In practice we have an interleaving of parts of all sub-derivations corresponding to most general calls of arguments, leaded by the most general derivation of the main call. Furthermore we have sharing which constraints steps performed for a shared subterm reached through a certain term path to be performed synchronously also in all other possible alternative term paths. Intuitively our proposal consists in defining an embedding operation that mimics parameter passing, namely taken two small-step trees T_1, T_2 and a variable x of (the root of) T_1, the tree-embedding operation $T_1[x/T_2]$ transforms T_1 by replacing steps accordingly to steps of T_2 which provides specific actual parameter values to x in places where x in T_1 was originally "freely" instantiated.

The problem of sharing can simply be solved by introducing a fresh variable and then embedding the tree for the shared term in place of the fresh variable.

Formally, let us define $knots(e)$ as the set of non-root positions p of e which are not labelled with a variable. Moreover let $baseknots(e)$ be the subset of $knots(e)$ of positions that cannot be reached by another position in $knots(e)$.

The evaluation of a term e w.r.t. an interpretation $\mathcal{I}$, namely $\mathcal{E}[\![e]\!]_{\mathcal{I}}$, is defined by induction on the size of $knots(e)$ as follows[11]

$$\mathcal{E}[\![x]\!]_{\mathcal{I}} := x \tag{2.2.5a}$$

$$\mathcal{E}[\![\varphi(\overrightarrow{x_n})]\!]_{\mathcal{I}} := \mathcal{I}(\varphi(\overrightarrow{y_n}))[y_1/x_1]\dots[y_n/x_n] \qquad \overrightarrow{y_n} \text{ fresh distinct} \tag{2.2.5b}$$

$$\mathcal{E}[\![e]\!]_{\mathcal{I}} := \mathcal{E}[\![e[y]_p]\!]_{\mathcal{I}}[y/\,\mathcal{E}[\![e|_p]\!]_{\mathcal{I}}] \qquad y \text{ fresh, leftmost } p \in baseknots(e) \tag{2.2.5c}$$

where the *tree-embedding* operation $G[x/B]$, given two small-step trees G and B rooted in g and b respectively, such that

1. g and b are two compatible terms which do not share any $\mathcal{D}$-labelled node, namely $p \in \mathcal{N}_g \cap \mathcal{N}_b \implies \mathcal{L}(p) \notin \mathcal{D}$;
2. G and B do not share any local node and local variable;
3. x is a variable which doesn't occur in B

is defined as

$$G[x/B] := \{d \,|\, d_g \in G, d_b \in B, d_g[x/d_b]_\varepsilon \vdash d,\ d \text{ canonical}\} \tag{2.2.6}$$

where, for all linear constructor substitution ϑ and all small-step sequences d_g and d_b, whose heads are g and b, such that there exists a homomorphism $h\colon x\vartheta \to b$, $d_g[x/d_b]_\vartheta \vdash d$ is the least relation that satisfies the rules

$$\frac{}{d_g[x/d_b]_\vartheta \vdash h[g]} \tag{2.2.7a}$$

$$\frac{d_g \vdash^\sigma d' \qquad d'[x/d_{b'}]_{\vartheta\sigma} \vdash d''}{d_g[x/b \overset{\sigma}{\underset{p}{\Rightarrow}} d_{b'}]_\vartheta \vdash h[g] \overset{\sigma}{\underset{p}{\Rightarrow}} d''}\ h[g]|_p \text{ outermost op-rooted subterm} \tag{2.2.7b}$$

[11] For the sake of simplicity Equation (2.2.5c) is actually given in term of leftmost position, even if it can be proved that the result would be independent of any other possible choice.

$$\frac{d_{g'}[x/d_b]_\vartheta \vdash d'}{(g \underset{p}{\overset{\varepsilon}{\Rightarrow}} d_{g'})[x/d_b]_\vartheta \vdash h[g] \underset{p}{\overset{\varepsilon}{\Rightarrow}} d'} \tag{2.2.7c}$$

$$\frac{(g \underset{p}{\overset{\sigma}{\Rightarrow}} d_{g'}) \vdash^\eta d' \qquad d'[x/d_{b'}]_{\vartheta\eta} \vdash d''}{(g \underset{p}{\overset{\sigma}{\Rightarrow}} d_{g'})[x/b \underset{q}{\overset{\eta}{\Rightarrow}} d_{b'}]_\vartheta \vdash h[g] \underset{p}{\overset{\eta}{\Rightarrow}} d''} \sigma = \{y/c(\overrightarrow{z_n})\},\ h[y] \text{ rooted in } q \tag{2.2.7d}$$

$$\frac{d_{g'}[x/d_b]_{\vartheta\sigma} \vdash d'}{(g \underset{p}{\overset{\sigma}{\Rightarrow}} d_{g'})[x/d_b]_\vartheta \vdash d'} \sigma = \{y/c(\overrightarrow{z_n})\},\ h[y] \text{ is } \mathcal{C}\text{-rooted} \tag{2.2.7e}$$

$$\frac{d_b[w/c(\overrightarrow{z_n})]_\varepsilon \vdash d' \qquad d_{g'}[x/d']_{\vartheta\sigma} \vdash d''}{(g \underset{p}{\overset{\sigma}{\Rightarrow}} d_{g'})[x/d_b]_\vartheta \vdash h[g] \underset{p}{\overset{\{w/c(\overrightarrow{z_n})\}}{\Longrightarrow}} d''} \sigma = \{y/c(\overrightarrow{z_n})\},\ h[y] = w \in \mathcal{V} \tag{2.2.7f}$$

provided that $d \vdash^\sigma d'$ is a shorthand for $d = d'$, when $\sigma = \varepsilon$, and $d[x/t]_\varepsilon \vdash d'$, when $\sigma = \{x/t\}$.

Broadly speaking, the role of ϑ in a statement $d_g[x/d_b]_\vartheta \vdash d$ is that of the "parameter pattern" responsible to constrain "freely" instantiated formal parameters in d_g to the actual parameters values which are actually "coming" from d_b. More specifically, Rules 2.2.7 govern the inlaying of the steps of a derivation d_b for a variable x into a derivation d_g. This is done by means of a sort of parameter passing, handled as the application of the homomorphism $h\colon x\vartheta \to b$ on the head of d_g.

In particular

- the axiom (2.2.7a) stops any further possible inlaying;
- the rule (2.2.7b) considers the case when the first step of $b \underset{p}{\overset{\sigma}{\Rightarrow}} d_{b'}$ concerns the subterm of b corresponding to an outermost op-rooted subterm of $h[g]$. We recursively compute the inlaying of $d_{b'}$ into the update of d_g with σ.
- the rule (2.2.7c) considers the case when the first step of d_g is a pure rewriting step. It this case there is no need to perform any step in an actual parameter before that of g. Thus, we inlay the derivation d_b into $d_{g'}$.
- rules (2.2.7d), (2.2.7e) and (2.2.7f) consider the case when the first step of $g \underset{p}{\overset{\sigma}{\Rightarrow}} d_{g'}$ is an instantiation step which binds a formal parameter y, with a linear constructor term $c(\overrightarrow{z_n})$, and its corresponding actual value
  - is an outermost op-rooted subterm of b ($h[y]$ is rooted in q) and the sequence for the parameter $b \underset{q}{\overset{\eta}{\Rightarrow}} d_{b'}$ actually performs a step on q. Rule (2.2.7d) inlays $d_{b'}$ into the update of $g \underset{p}{\overset{\sigma}{\Rightarrow}} d_{g'}$ with η.
  - matches. Hence the first step $\underset{p}{\overset{\sigma}{\Rightarrow}}$ is producing a binding "superseded" by the actual value $h[y]$, and thus rule (2.2.7e) simply discards it.
  - is a variable too, namely $h[y] = w$. Rule (2.2.7f) first performs an instantiation step with a fresh variant of $c(\overrightarrow{z_n})$ and then continues to inlay the updated version of d_b into $d_{g'}$.

Example 2.2.18

Consider the interpretation

$$\mathcal{I} = \left\{ zeros \mapsto zeros \overset{\varepsilon}{\Rightarrow} Z : zeros \right.$$

The evaluation of $\mathcal{E}[\![Z : zeros]\!]_{\mathcal{I}}$ is

$$\begin{array}{ll}
\mathcal{E}[\![Z : zeros]\!]_{\mathcal{I}} = & [\text{ by Equation (2.2.5c)}] \\
\mathcal{E}[\![y_1 : zeros]\!]_{\mathcal{I}}[y_1/\mathcal{E}[\![Z]\!]_{\mathcal{I}}] = & [\text{since } \mathcal{I}(Z) = Z, \text{ by Equation (2.2.5b)}] \\
\mathcal{E}[\![y_1 : zeros]\!]_{\mathcal{I}}[y_1/Z] = & [\text{ by Equation (2.2.5c)}] \\
(\mathcal{E}[\![y_1 : y_2]\!]_{\mathcal{I}}[y_1/Z])[y_2/\mathcal{E}[\![zeros]\!]_{\mathcal{I}}] = & [\text{since } \mathcal{I}(y_1 : y_2) = y_1 : y_2, \text{ by Equation (2.2.5b)}] \\
((y_1 : y_2)[y_1/Z])[y_2/\mathcal{E}[\![zeros]\!]_{\mathcal{I}}] = & [\text{since, by Equation (2.2.7a)}, (y_1 : y_2)[y_1/Z]_{\varepsilon} \vdash Z : y_2] \\
(Z : y_2)[y_2/\mathcal{E}[\![zeros]\!]_{\mathcal{I}}] = & [\text{since } \mathcal{E}[\![zeros]\!]_{\mathcal{I}} = \mathcal{I}(zeros)] \\
(Z : y_2)[y_2/zeros \overset{\varepsilon}{\Rightarrow} Z : zeros] = & [\text{ by Equation (2.2.8)}] \\
Z : zeros \overset{\varepsilon}{\Rightarrow} Z : Z : zeros &
\end{array}$$

$$\text{(2.2.7b)}\ \frac{\text{(2.2.7a)}\ \dfrac{}{(Z : y_2)[y_2/Z : zeros]_{\varepsilon} \vdash Z : Z : zeros}}{(Z : y_2)[y_2/zeros \overset{\varepsilon}{\Rightarrow} Z : zeros]_{\varepsilon} \vdash Z : zeros \overset{\varepsilon}{\Rightarrow} Z : Z : zeros} \tag{2.2.8}$$

It can be clarifying to give an alternative definition of tree-embedding, in terms of "traditional" terms and substitutions (with the same preconditions). However note that in this way we actually remove the sharing in favor of duplicating copies. Thus it is correct w.r.t. call-time choice only when there are no shared subterms, where, given the substitution $\sigma = \{x/t\}$, $G\sigma$ is a shorthand for $G[x/(\!|\, t; \emptyset \,|\!)]$.

It can be noted that Rules 2.2.7 produce "spurious" non-canonical derivations, which are then filtered out by Equation (2.2.6). Actually Rules 2.2.7 could be changed to construct only canonical derivations, but at the cost of a much more complex definition. However, since the purpose of this semantics is just to define (and formally prove the correctness of) the big-step abstraction, this change is not necessary, indeed it is counterproductive, as for the correctness proofs the presented form is better.

Properties of the Program Operational Denotation

The following result states formally that the evaluation function can reconstruct the small-step tree of any term.

Theorem 2.2.19 *For all $P \in \mathbb{P}_{\Sigma}$ and $e \in \mathcal{T}(\Sigma, \mathcal{V})$,*

$$\mathcal{E}[\![e]\!]_{\mathcal{O}[\![P]\!]} = \mathcal{N}[\![e \text{ in } P]\!].$$

A straightforward consequence of Theorems 2.2.19 and 2.2.15 is

Corollary 2.2.20 *For all P_1, $P_2 \in \mathbb{P}_{\Sigma}$, $P_1 \approx_{ss} P_2$ if and only if $\mathcal{O}[\![P_1]\!] = \mathcal{O}[\![P_2]\!]$.*

Thus semantics $\mathcal{O}$ is fully abstract w.r.t. $\approx_{ss}$. Now we should find a bottom-up goal-independent equivalent definition of $\mathcal{O}$. It definition is essentially given in terms of $\mathcal{E}$ of rules' right hand side.

2.2.4 Fixpoint Denotations of Programs

We will now show a bottom-up goal-independent denotation which is equivalent to $\mathcal{O}$ and thus adequate to characterize the small-step behaviour for Curry programs. It will be defined as the fixpoint of an abstract immediate operator over interpretations. This operator, given an interpretation $\mathcal{I}$, essentially consists in

- building an initial sequence of instantiation steps of a most general call, according to the definitional tree, and then
- applying the evaluation operator of the bodies of program rules over $\mathcal{I}$

Formally, $\mathcal{P} : \mathbb{P}_\Sigma \to \mathbb{I} \to \mathbb{I}$ is defined as

$$\mathcal{P}[\![P]\!]_\mathcal{I} := \lambda f(\overrightarrow{x_n}).\ \xi[\![T_f]\!]_\mathcal{I} \tag{2.2.9}$$

where each T_f is a fresh definitional tree for f s.t. $pat(T_f) = f(\overrightarrow{x_n})$ and the evaluation $\xi[\![T]\!]_\mathcal{I}$ of a definitional tree T w.r.t. $\mathcal{I}$ is defined as

$$\xi[\![rule(\pi \to r)]\!]_\mathcal{I} := \pi \xRightarrow[\mathcal{R}oot_\pi]{\varepsilon} \mathcal{E}[\![r]\!]_\mathcal{I} \tag{2.2.10a}$$

$$\xi[\![or(T_1, T_2)]\!]_\mathcal{I} := \xi[\![T_1]\!]_\mathcal{I} \sqcup \xi[\![T_2]\!]_\mathcal{I} \tag{2.2.10b}$$

$$\xi[\![branch(\pi, p, \overrightarrow{T_k})]\!]_\mathcal{I} := \bigsqcup \left\{ \pi \xRightarrow[\mathcal{R}oot_\pi]{\sigma} \xi[\![T_i]\!]_\mathcal{I} \;\middle|\; \pi\sigma = pat(T_i), 1 \le i \le k \right\} \tag{2.2.10c}$$

Until we do not reach a leaf of a definitional tree, Equation (2.2.10c) builds suitable instantiation steps. Then (when we reach a leaf) Equation (2.2.10a) evaluates the body of a rule over the (current) interpretation. For all *or* nodes Equation (2.2.10b) simply collects all alternatives.

Example 2.2.21

Consider the program P

```
coin = Head                    zeros = Z : zeros
coin = Tail
```

(by what shown in Example 2.2.18) the iterates of $\mathcal{P}[\![P]\!]$ are

$$\mathcal{P}[\![P]\!]\uparrow 1 = \begin{cases} zeros \mapsto zeros \xRightarrow{\varepsilon} Z : zeros \\ coin \mapsto\ coin \xRightarrow{\varepsilon} Head \\ \qquad\qquad\quad\ \ \xRightarrow{\varepsilon} Tail \end{cases}$$

$$\mathcal{P}[\![P]\!]\uparrow 2 = \begin{cases} zeros \mapsto zeros \xRightarrow{\varepsilon} Z : zeros \xRightarrow{\varepsilon} Z : Z : zeros \\ coin \mapsto\ coin \xRightarrow{\varepsilon} Head \\ \qquad\qquad\quad\ \ \xRightarrow{\varepsilon} Tail \end{cases}$$

$\vdots$

$$\mathcal{P}[\![P]\!]\uparrow\omega = \begin{cases} zeros \mapsto zeros \overset{\varepsilon}{\Rightarrow} Z : zeros \overset{\varepsilon}{\Rightarrow} Z : Z : zeros \overset{\varepsilon}{\Rightarrow} Z : Z : Z : zeros \overset{\varepsilon}{\Rightarrow} \dots \\ coin \mapsto coin \overset{\varepsilon}{\underset{\varepsilon}{\rightrightarrows}} \begin{matrix} Head \\ Tail \end{matrix} \end{cases}$$

Example 2.2.22

The first two iterates for the program P_+ of Example 2.2.13 are

$$\mathcal{P}[\![P_+]\!]\uparrow 1 = \{ x_1 + y_0 \mapsto x_1 + y_0 \overset{\{x_1/S(x_0)\}}{\Rightarrow} S(x_0) + y_0 \overset{\varepsilon}{\to} S(x_0 + y_0) \;;\; x_1 + y_0 \overset{\{x_1/Z\}}{\Rightarrow} Z + y_0 \overset{\varepsilon}{\to} y_0 \}$$

$$\mathcal{P}[\![P_+]\!]\uparrow 2 = \{ x_2 + y_0 \mapsto x_2 + y_0 \overset{\{x_2/Z\}}{\Rightarrow} Z + y_0 \overset{\varepsilon}{\Rightarrow} y_0 \;;\; x_2 + y_0 \overset{\{x_2/S(x_1)\}}{\Rightarrow} S(x_1) + y_0 \overset{\varepsilon}{\Rightarrow} S(x_1 + y_0) \overset{\{x_1/Z\}}{\Rightarrow} S(Z + y_0) \overset{\varepsilon}{\Rightarrow} S(y_0) \;;\; S(x_1 + y_0) \overset{\{x_1/S(x_0)\}}{\Rightarrow} S(S(x_0) + y_0) \overset{\varepsilon}{\Rightarrow} S(S(x_0 + y_0)) \}$$

Note that, to highlight the construction order of the various subtrees, we used indices in variables names that respect the order of introduction and boxed the corresponding subtrees along the iterations.

By continuing the computation of the iterates, we have that $(\mathcal{P}[\![P_+]\!]\uparrow\omega)(x + y) = \mathcal{O}[\![P_+]\!](x + y) = \mathcal{N}[\![x + y \; in \; P_+]\!]$ (of Example 2.2.13). This is not a casualty, because of Theorem 2.2.25.

Proposition 2.2.23 *For all $P \in \mathbb{P}_\Sigma$, $\mathcal{P}[\![P]\!]$ is continuous.*

Since $\mathcal{P}[\![P]\!]$ is continuous we can define our fixpoint semantics as

$$\mathcal{F}[\![P]\!] := \mathcal{P}[\![P]\!]\uparrow\omega = lfp(\mathcal{P}[\![P]\!]) \tag{2.2.11}$$

Example 2.2.24

Consider the program P

```
f (A x) = B (loop x)          loop (C x) = loop x
g (A x) = B (h A)             h B = C
```

the fixpoint $\mathcal{F}[\![P]\!]$ is

$$\begin{cases} h(x) \mapsto h(x) \xrightarrow{\{x/B\}} h(B) \stackrel{\varepsilon}{\Longrightarrow} C \\ g(x) \mapsto g(x) \xrightarrow{\{x/A(x_1)\}} f(A(x_1)) \stackrel{\varepsilon}{\Longrightarrow} B(h(A)) \\ loop(x_0) \mapsto loop(x_0) \xrightarrow{\{x/C(x_1)\}} loop(C(x_1)) \stackrel{\varepsilon}{\Longrightarrow} loop(x_1) \xrightarrow{\{x_1/C(x_2)\}} \\ \qquad\qquad loop(C(x_2)) \stackrel{\varepsilon}{\Longrightarrow} loop(x_2) \cdots \\ f(x) \mapsto f(x) \xrightarrow{\{x/A(x_1)\}} f(A(x_1)) \stackrel{\varepsilon}{\Longrightarrow} B(loop(x_1)) \xrightarrow{\{x_1/C(x_2)\}} \\ \qquad\qquad B(loop(C(x_2))) \stackrel{\varepsilon}{\Longrightarrow} B(loop(x_2)) \cdots \end{cases}$$

This example shows two functions f, g which do not produce computed results, one because it has *only* an infinite derivation and the other because it uses a non total function. We will see that f and g will have the same big-step semantics.

Properties of the Program Fixpoint Denotation

The top-down goal-dependent denotation $\mathcal{O}$ and the bottom-up goal-independent denotation $\mathcal{F}$ are actually equivalent.

Theorem 2.2.25 *For all $P \in \mathbb{P}_\Sigma$, $\mathcal{F}[\![P]\!] = \mathcal{O}[\![P]\!]$.*

A straightforward consequence of Theorem 2.2.25 and Corollary 2.2.20 is

Corollary 2.2.26 (Full abstraction of $\mathcal{F}$ w.r.t. $\approx_{ss}$) *For all $P_1, P_2 \in \mathbb{P}_\Sigma$, $P_1 \approx_{ss} P_2$ if and only if $\mathcal{F}[\![P_1]\!] = \mathcal{F}[\![P_2]\!]$.*

This result states that the given fixed-point characterization is fully abstract w.r.t. the small-step operational behaviour.

2.A Proofs

Proof of Lemma 2.2.3.
Let us define the (syntactic) program equivalence $P_1 \stackrel{pt}{=} P_2$ identifying programs that for each definitional tree T_f of each defined symbol f have, along any path from the root of T_f to a leaf node, the same patterns. In other terms have identical definitional trees up to rearrangement of or-nodes.

Note that the possible different contributions of rules of the small-step operational semantics depends solely on patterns in leaf-nodes or branch-nodes (or-nodes just collect all the other contributions). Hence definitional trees for the same rules that have the same patterns along root-to-leaves paths, must produce the same derivations. Thus

$$P_1 \stackrel{pt}{=} P_2 \Longleftrightarrow P_1 \approx_{fss} P_2. \tag{1}$$

Now observe that if we take any derivation $f(\overrightarrow{x_n}) \underset{p}{\overset{\sigma_1}{\Rightarrow}} \dots \underset{p}{\overset{\sigma_n}{\Rightarrow}} f(\overrightarrow{x_n})\sigma_1 \dots \sigma_n \underset{p}{\overset{\varepsilon}{\Rightarrow}} s \in \mathcal{B}^{ss}[\![P]\!]$, $f(\overrightarrow{x_n}), f(\overrightarrow{x_n})\sigma_1, \dots, f(\overrightarrow{x_n})\sigma_1 \cdots \sigma_{n-1}$ are the patterns of the branch nodes encountered along a path from the root of T_f to a leaf node $rule(\pi \to s)$ such that $\pi = f(\overrightarrow{x_n})\sigma_1 \cdots \sigma_n$.

Hence $P_1 \approx_{ss} P_2 \iff P_1 \overset{pt}{=} P_2$ and thus, with (1) we conclude.

Lemma 2.A.1 *Let $P \in \mathbb{P}_\Sigma$ and $t_0 \underset{p}{\overset{\sigma_1}{\Rightarrow}} t_1 \underset{q}{\overset{\sigma_2}{\Rightarrow}} \cdots \underset{q}{\overset{\sigma_n}{\Rightarrow}} t_n$ a derivation in $\mathcal{B}^{ss}[\![P]\!]$ such that $p, q \in \mathcal{N}_{t_0}$, $p \neq q$ and $(t_n)|_q$ is in head normal form. Then, there exists a derivation $s_0 \underset{q}{\overset{\vartheta_1}{\Rightarrow}} \cdots \underset{q}{\overset{\vartheta_{n-1}}{\Rightarrow}} s_{n-1} \underset{o}{\overset{\vartheta_n}{\Rightarrow}} s_n$ in $\mathcal{B}^{ss}[\![P]\!]$ such that $s_0 = t_0$, $s_n = t_n$, $\vartheta_1 \cdots \vartheta_n = \sigma_1 \cdots \sigma_n$ and $o \in \{p, q\}$.*

Proof.
We first notice that for any derivation $t_0 \underset{p}{\overset{\sigma_1}{\Rightarrow}} t_1 \underset{q}{\overset{\sigma_2}{\Rightarrow}} t_2$ there exist a derivation $s_0 \underset{q}{\overset{\vartheta_1}{\Rightarrow}} s_1 \underset{o}{\overset{\vartheta_2}{\Rightarrow}} s_2$ such that $s_0 = t_0$, $s_2 = t_2$, $\vartheta_1\vartheta_2 = \sigma_1\sigma_2$ and $o \in \{p, q\}$. Actually, let $p_1, \dots, p_n$ the nodes traversed in the proof for $Eval[\![t_0]\!] \overset{p,\sigma_1}{\Longrightarrow} t_1$, and $q_1, \dots, q_n$ those traversed in the proof for $Eval[\![t_1]\!] \overset{q,\sigma_1}{\Longrightarrow} t_2$. Let p'_n the node which takes place of p_n in t_1. If $p'_n \in \{q_1, \dots, q_n\}$, by $Eval[\![t_1]\!] \overset{q,\sigma_2}{\Longrightarrow} t_2$, there must exists $Eval[\![t_0]\!] \overset{q,\sigma_1}{\Longrightarrow} t_1$. Otherwise, if $p'_n \notin \{q_1, \dots, q_n\}$, the two steps are disjoint and the order in which they are performed can be "swapped". This means that it exists a derivation $t_0 \underset{q}{\overset{\sigma_2}{\Rightarrow}} s_1 \underset{p}{\overset{\sigma_1}{\Rightarrow}} s_2$ where $s_2 = s_1[s']_p = (t_0[s'']_q)[s']_p = (t_0[s']_p)[s'']_q = t_1[s'']_q = t_2$ for some s', s''.

Now we are ready to prove the main result. We proceed by induction on the number k of steps labelled with q:

Base Case: ($k = 1$) Straightforward, by that we stated before.

Inductive Step: ($k > 1$) As previously stated, there exists a derivation $s_0 \underset{q}{\overset{\vartheta_1}{\Rightarrow}} s_1 \underset{o}{\overset{\vartheta_2}{\Rightarrow}} s_2$ such that $s_0 = t_0$, $s_2 = t_2$, $\vartheta_1\vartheta_2 = \sigma_1\sigma_2$ and $o \in \{p, q\}$. We consider two cases:

***o* = *q*)** Immediate, and $(s_0 =)t_0 \underset{q}{\overset{\vartheta_1}{\Rightarrow}} s_1 \underset{q}{\overset{\vartheta_2}{\Rightarrow}} (s_2 =)t_2 \cdots \underset{q}{\overset{\sigma_n}{\Rightarrow}} t_n$ is a witness.

***o* = *p*)** consider $(s_0 =)t_0 \underset{q}{\overset{\vartheta_1}{\Rightarrow}} s_1 \underset{p}{\overset{\vartheta_2}{\Rightarrow}} (s_2 =)t_2 \cdots \underset{q}{\overset{\sigma_n}{\Rightarrow}} t_n$. The thesis follows by inductive hypothesis on the sub-derivation $s_1 \underset{p}{\overset{\vartheta_2}{\Rightarrow}} \cdots \underset{q}{\overset{\sigma_n}{\Rightarrow}} t_n$.

Proof of Proposition 2.2.4.
For the sake of clarity, we make a slight abuse of notation denoting the positions which are not equivalent to p with $\bar{p}$. Let k be the index of the last step labelled with a position p. Under this assumption the starting derivation is denoted as

$$e_0 \underset{p_1}{\overset{\vartheta_1}{\Rightarrow}} \cdots \underset{p_{k-1}}{\overset{\vartheta_{k-1}}{\Rightarrow}} e_{k-1} \underset{p}{\overset{\vartheta_k}{\Rightarrow}} e_k \underset{\bar{p}}{\overset{\vartheta_{k+1}}{\Rightarrow}} \cdots \underset{\bar{p}}{\overset{\vartheta_n}{\Rightarrow}} e_n\,,$$

where the positions $p_1, \dots, p_{k-1}$ can be either equivalent to p or not. We proceed by induction on the number h of steps between e_0 and e_k which are denoted with a position $\bar{p}$.

Base Case: $(h = 0)$ immediate.

Inductive Step: $(h > 0)$ Let i the index of the first step labelled with $\bar{p}$ moving backward from e_k. $e_0 \overset{\vartheta_1}{\underset{p_1}{\Longrightarrow}} \cdots \overset{\vartheta_{i-1}}{\underset{p_{i-1}}{\Longrightarrow}} e_{i-1} \overset{\vartheta_i}{\underset{\bar{p}}{\Longrightarrow}} e_i \overset{\vartheta_{i+1}}{\underset{p}{\Longrightarrow}} \cdots \overset{\vartheta_k}{\underset{p}{\Longrightarrow}} e_k \overset{\vartheta_{k+1}}{\underset{\bar{p}}{\Longrightarrow}} \cdots \overset{\vartheta_n}{\underset{\bar{p}}{\Longrightarrow}} e_n$, by Lemma 2.A.1 there exists a derivation $s_{i-1} \overset{\sigma_i}{\underset{p}{\Longrightarrow}} s_i \overset{\sigma_{i+1}}{\underset{p}{\Longrightarrow}} \cdots \overset{\sigma_k}{\underset{o}{\Longrightarrow}} s_k$ such that $s_{i-1} = t_{i-1}$, $s_k = e_k$, $\sigma_i \cdots \sigma_k = \vartheta_i \cdots \vartheta_k$ and $o \in \{p, \bar{p}\}$. Thus there exist $e_0 \overset{\vartheta_1}{\underset{p_1}{\Longrightarrow}} \cdots \overset{\vartheta_{i-1}}{\underset{p_{i-1}}{\Longrightarrow}} s_{i-1} \overset{\sigma_i}{\underset{p}{\Longrightarrow}} s_i \overset{\sigma_{i+1}}{\underset{p}{\Longrightarrow}} \cdots \overset{\sigma_k}{\underset{o}{\Longrightarrow}} s_k \overset{\vartheta_{k+1}}{\underset{\bar{p}}{\Longrightarrow}} \cdots \overset{\vartheta_n}{\underset{\bar{p}}{\Longrightarrow}} e_n$, such that $\vartheta_1 \cdots \vartheta_{i-1}\sigma_i \cdots \sigma_k \vartheta_{k+1} \cdots \vartheta_n = \vartheta_1 \cdots \vartheta_n$. There are two possible cases:

***o* = *p*)** we just need to apply the inductive hypothesis on the sub-derivation $e_0 \overset{\vartheta_1}{\underset{p_1}{\Longrightarrow}} \cdots \overset{\vartheta_{i-1}}{\underset{p_{i-1}}{\Longrightarrow}} s_{i-1} \overset{\sigma_i}{\underset{p}{\Longrightarrow}} s_i \overset{\sigma_{i+1}}{\underset{p}{\Longrightarrow}} \cdots \overset{\sigma_k}{\underset{p}{\Longrightarrow}} s_k$;

***o* = $\bar{p}$)** again, we just apply the inductive hypothesis on the sub-derivation $e_0 \overset{\vartheta_1}{\underset{p_1}{\Longrightarrow}} \cdots \overset{\vartheta_{i-1}}{\underset{p_{i-1}}{\Longrightarrow}} s_{i-1} \overset{\sigma_i}{\underset{p}{\Longrightarrow}} s_i \overset{\sigma_{i+1}}{\underset{p}{\Longrightarrow}} \cdots \overset{\sigma_{k-1}}{\underset{p}{\Longrightarrow}} s_{k-1}$.

Proof of Proposition 2.2.7.

First we show that for all $P \in \mathbb{P}_\Sigma$ it holds that

$$
\begin{aligned}
&\mathcal{B}^{fss}[\![P]\!] = \\
&\qquad [\text{by Definition 2.2.2}] \\
&= \left\{ e_0 \overset{\sigma_0}{\underset{p_0}{\Longrightarrow}} e_1 \overset{\sigma_1}{\underset{p_1}{\Longrightarrow}} \ldots \;\middle|\; \forall i.\ \mathit{Eval}[\![e_i]\!] \overset{p_i,\sigma_i}{\Longrightarrow} e_{i+1} \right\} \\
&\qquad [\text{by Definition 2.2.6 and since any single step sequence is canonical}] \\
&= \left\{ e_0 \overset{\sigma_0}{\underset{p_0}{\Longrightarrow}} e_1 \overset{\sigma_1}{\underset{p_1}{\Longrightarrow}} \ldots \;\middle|\; \forall i.\ e_i \overset{\sigma_i}{\underset{p_i}{\Longrightarrow}} e_{i+1} \in \mathcal{B}^{ss}[\![P]\!] \right\} \qquad (1)
\end{aligned}
$$

Thus $\mathcal{B}^{fss}[\![P]\!]$ can be retrieved from $\mathcal{B}^{ss}[\![P]\!]$.

From this the thesis becomes immediate, since

$$
\begin{aligned}
&P_1 \approx_{fss} P_2 && [\text{by definition of } \approx_{fss}] \\
&\iff \mathcal{B}^{fss}[\![P_1]\!] = \mathcal{B}^{fss}[\![P_2]\!] && [\text{by Definition 2.2.6 and (1)}] \\
&\iff \mathcal{B}^{ss}[\![P_1]\!] = \mathcal{B}^{ss}[\![P_2]\!] && [\text{by definition of } \approx_{ss}] \\
&\iff P_1 \approx_{fss} P_2
\end{aligned}
$$

Proof of Theorem 2.2.15.

By Definitions 2.2.12 and 2.2.9 it is straightforward to see that for any $P \in \mathbb{P}_\Sigma$ the following equality holds

$$
\mathcal{B}^{ss}[\![P]\!] = \bigcup_{e \in \mathcal{T}(\Sigma,\mathcal{V})} \left\{ \iota[d] \;\middle|\; \begin{array}{l} d \in \mathit{paths}(\mathcal{N}[\![e \text{ in } P]\!]) \\ \iota \colon \mathcal{N} \to \mathcal{N} \text{ isomorphism s.t. } \iota[e] = e \end{array} \right\}
$$

Then, $\mathcal{B}^{ss}[\![P_1]\!] = \mathcal{B}^{ss}[\![P_2]\!]$ if and only if $\forall e \in \mathcal{T}(\Sigma, \mathcal{V})$. $\mathcal{N}[\![e \text{ in } P_1]\!] = \mathcal{N}[\![e \text{ in } P_2]\!]$.

In order to be more concise in our statements we introduce the notion of we called *embeddable pair*.

Definition 2.A.2 (embeddable pair) *$e, e' \in \mathcal{T}(\Sigma, \mathcal{V})$ is said* embeddable pair *if e and e' are compatible and do not share any $\mathcal{D}$-labelled node.*

An embeddable pair is a pair of terms which fulfills the minimal requirements that ensure both the soundness (see Lemma 2.A.4) and the completeness (see Lemma 2.A.6) of the small-step tree-embedding operation (Equation (2.2.6)) w.r.t. the operational semantics.

Lemma 2.A.3 *For all small-step derivation $d_g \in \mathcal{N}[\![g \text{ in } P]\!]$ and for all $\mathcal{C}$-linear substitutions σ either of the form ε or $\{x/c(\overrightarrow{z_n})\}$, if $d_g \vdash^{\sigma} d$ then the head of d is $g\sigma$.*

Proof of Lemma 2.A.3.
By definition of $\vdash^{\sigma}$ if $\sigma = \varepsilon$ then $d = d_g$ and the thesis trivially holds.

If $\sigma = \{x/t\}$, d is a derivation which holds $d_g[x/t]_{\varepsilon} \vdash d$. By construction there exists $h\colon x \to t$ and $h[g] = g\sigma$. The thesis hold since every rule in 2.2.7 infers a sequence whose head is $h[g]$.

Lemma 2.A.4 *Let $P \in \mathbb{P}_{\Sigma}$, let g, b be an embeddable pair, $x \in var(g) \setminus var(b)$, and let ϑ be a $\mathcal{C}$-linear substitution s.t. there exists a homomorphism $h\colon x\vartheta \to b$. Then, $\mathcal{N}[\![g \text{ in } P]\!][x/\mathcal{N}[\![b \text{ in } P]\!]] \sqsubseteq \mathcal{N}[\![g\{x/b\} \text{ in } P]\!]$*

Proof.
To keep clean the notation in the following d_t will denote a small-step derivation whose head is the term t.

By Equation (2.2.6) it suffices to prove that, given $d_g \in \mathcal{N}[\![g \text{ in } P]\!]$ and $d_b \in \mathcal{N}[\![b \text{ in } P]\!]$, if it holds $d_g[x/d_b]_{\varepsilon} \vdash d$ and d is canonical then $d \in \mathcal{N}[\![g\{x/b\} \text{ in } P]\!]$.

To this aim we will prove a stronger result: let g, b be an embeddable pair, $x \notin var(b)$, and let ϑ be a $\mathcal{C}$-linear substitution such that $img(\theta)$ does not share any node with b and there exists a homomorphism $h\colon x\vartheta \to b$. Then, for any $d_g \in \mathcal{N}[\![g \text{ in } P]\!]$ and $d_b \in \mathcal{N}[\![b \text{ in } P]\!]$ such that $d_g[x/d_b]_{\vartheta} \vdash d$ and d is canonical then $d \in \mathcal{N}[\![h[g] \text{ in } P]\!]$. We proceed by induction on the height of the proof for $d_g[v/d_b]_{\vartheta} \vdash d$.

Base Case: immediate by rule 2.2.7a.

Inductive Step: First of all note that, by Definition 2.2.12, for every step $t_i \overset{\sigma_i}{\underset{p_i}{\Rightarrow}} t_{i+1}$ of d_g or d_b it holds $Eval[\![t_i]\!] \overset{p_i, \sigma_i}{\Longrightarrow} t_{i+1}$. Now we proceed with the proof by cases on the first rule applied in the proof of $d_g[v/d_b]_{\vartheta} \vdash d$.

rule 2.2.7b) we have that d_b is of the form $b \overset{\sigma}{\underset{p}{\Rightarrow}} d_{b'}$ and $h[g]|_p$ is an outermost op-rooted subterm. By (Cntx) $Eval[\![b]\!] \overset{p, \sigma}{\Longrightarrow} b'$ if and only if $Eval[\![b|_p]\!] \overset{p, \sigma}{\Longrightarrow} s$ and $b' = b[s]_p$. Without loss of generality we assume the nodes introduced in the step (i.e., $\mathcal{N}_s \setminus \mathcal{N}_b$) are renamed apart both from $x\vartheta$ and d_g. By the conditions on rule 2.2.7b, $p \in \mathcal{N}_{h[g]}$; by hypothesis, b and g do not share $\mathcal{D}$-labelled nodes, and ϑ is a $\mathcal{C}$-linear substitution

s.t. $h\colon x\vartheta \to b$, therefore $h[g]|_p = b|_p$. Since $h[g]|_p$ is an outermost op-rooted subterm of $h[g]$, by (Cntx), $Eval[\![b|_p]\!] \overset{p,\sigma}{\Longrightarrow} s$ if and only if

$$Eval[\![h[g]]\!] \overset{p,\sigma}{\Longrightarrow} (h[g])[s]_p. \tag{1}$$

By Lemma 2.A.3 and inductive hypothesis $d' \in \mathcal{N}[\![g\sigma \mathit{\ in\ } P]\!]$. Again, by inductive hypothesis

$$d'' \in \mathcal{N}[\![h'[g\sigma] \mathit{\ in\ } P]\!] \tag{2}$$

where $h'\colon x\vartheta\sigma \to b'$. If $\sigma = \varepsilon$, since $b' = b[s]_p$ we have that $(h[g])[s]_p = h'[g] = h'[g\sigma]$. On the other hand, if $\sigma \neq \varepsilon$, we have that $(h[g])[s]_p = (h[g])\sigma = h'[g\sigma]$. By $h'[g\sigma] = (h[g])[s]_p$, (1) and (2) we have that if $h[g] \overset{\sigma}{\underset{p}{\Rightarrow}} d''$ is canonical derivation then $h[g] \overset{\sigma}{\underset{p}{\Rightarrow}} d'' \in \mathcal{N}[\![h[g] \mathit{\ in\ } P]\!]$.

rule 2.2.7c) we have that d_g is of the form $g \overset{\varepsilon}{\underset{p}{\Rightarrow}} d_{g'}$, that is $g \overset{\varepsilon}{\underset{p}{\Rightarrow}} g'$ is a pure rewriting step. Therefore there exist in g a position q reachable from p, a rule $l \to r$ and a matcher $m\colon l \to g|_q$ such that $g' = g[m[r]]_q$. By hypothesis $h\colon x\vartheta \to b$, and $x\vartheta$ is a constructor term thus for every path in g from the root to q there exists a path in $h[g]$ with the same labels, obtained mapping each node of the former path with h. Thus a proof for $Eval[\![h[g]]\!] \overset{h(p),\varepsilon}{\Longrightarrow} h[g']$ can be given by mapping the proof for $Eval[\![g]\!] \overset{p,\varepsilon}{\Longrightarrow} g'$ with h. By construction h is $\mathcal{D}$-preserving, thus $h(p) = p$ and $Eval[\![h[g]]\!] \overset{p,\varepsilon}{\Longrightarrow} h[g']$. By inductive hypothesis $d' \in \mathcal{N}[\![h[g'] \mathit{\ in\ } P]\!]$, therefore if $h[g] \overset{\sigma}{\underset{p}{\Rightarrow}} d'$ is canonical we conclude that $h[g] \overset{\sigma}{\underset{p}{\Rightarrow}} d' \in \mathcal{N}[\![h[g] \mathit{\ in\ } P]\!]$.

rule 2.2.7e) we have that d_g is of the form $g \overset{\sigma}{\underset{p}{\Rightarrow}} d_{g'}$, with $\sigma = \{y/c(\overrightarrow{z_n})\}$, that is $g \overset{\sigma}{\underset{p}{\Rightarrow}} g'$ is an instantiation step and $g' = g\sigma$. Without loss of generality both the nodes and the variables of $c(\overrightarrow{z_n})$ are supposed to be renamed apart from both from $x\vartheta$ and d_b. By hypothesis $h\colon x\vartheta \to b$, $h[y]$ is $\mathcal{C}$-rooted, and $d_{g'}[x/d_b]_{\vartheta\sigma} \vdash d'$ holds, hence there exists $h'\colon x\vartheta\sigma \to b$. By construction $h'[g'] = h[g]$. By inductive hypothesis if d' is canonical then $d' \in \mathcal{N}[\![h'[g'] \mathit{\ in\ } P]\!]$ and therefore $d' \in \mathcal{N}[\![h[g] \mathit{\ in\ } P]\!]$.

rule 2.2.7f) we have that d_g is of the form $g \overset{\sigma}{\underset{p}{\Rightarrow}} d_{g'}$, with $\sigma = \{y/c(\overrightarrow{z_n})\}$, that is $g \overset{\sigma}{\underset{p}{\Rightarrow}} g'$ is an instantiation step and $g' = g\sigma$. Without loss of generality both the nodes and the variables of $c(\overrightarrow{z_n})$ are supposed to be renamed apart from both from $x\vartheta$ and d_b. By hypothesis $h[y] = w \in \mathcal{V}$ and $h\colon x\vartheta \to b$ thus there exists a homomorphism $h'\colon x\vartheta\sigma \to b\{w/c(\overrightarrow{z_n})\}$. By construction, $h'[g'] = h[g'] = h[g]\{w/c(\overrightarrow{z_n})\}$. It is immediate to see that a proof for $Eval[\![h[g]]\!] \overset{p,\{w/c(\overrightarrow{z_n})\}}{\Longrightarrow} h[g']$ can be given by mapping the proof for $Eval[\![g]\!] \overset{p,\sigma}{\Longrightarrow} g'$ with h, note that $\{w/c(\overrightarrow{z_n})\} = \{h[y]/c(\overrightarrow{z_n})\}$ and $h(p) = p$ since h is $\mathcal{D}$-preserving. By inductive hypothesis $d' \in \mathcal{N}[\![b\{w/c(\overrightarrow{z_n})\} \mathit{\ in\ } P]\!]$. By this and construction of h' we apply the inductive hypothesis on $d_{g'}[x/d']_{\vartheta\sigma} \vdash d''$ concluding that if d'' is canonical then $d'' \in \mathcal{N}[\![h'[g'] \mathit{\ in\ } P]\!]$. Thus, if $h[g] \overset{\{w/c(\overrightarrow{z_n})\}}{\underset{p}{\Longrightarrow}} d''$ is canonical then it belongs to $\mathcal{N}[\![h[g] \mathit{\ in\ } P]\!]$.

rule 2.2.7d) we have that d_g is of the form $g \overset{\sigma}{\underset{p}{\Rightarrow}} d_{g'}$, with $\sigma = \{y/c(\overrightarrow{z_n})\}$, and d_b is of the form $b \overset{\eta}{\underset{q}{\Rightarrow}} d_{b'}$. Thus $g \overset{\sigma}{\underset{p}{\Rightarrow}} g'$ is an instantiation step and $g' = g\sigma$, moreover $b|_q$ is an outermost op-rooted subterm of b. By hypothesis $h\colon x\vartheta b$ and ϑ is a $\mathcal{C}$-linear substitution, thus $h[x] = b|_q$. Without loss of generality both the nodes and the variables of $c(\overrightarrow{z_n})$ are supposed to be renamed apart from both from $x\vartheta$ and d_b.

It is straightforward to see that the proof of $Eval[\![g]\!] \overset{p,\vartheta}{\Longrightarrow} g'$ can be lifted with h up to the next to the last rule application. Rather than (Ins), it can only be applied (Dem) which makes this proof for $h[g]$ depends on one for $h[x]$. We said that $h[x] = b|_q$ is as outermost op-rooted subterm of b. By (Cntx), $Eval[\![b|_q]\!] \overset{q,\eta}{\Longrightarrow} s$ holds if and only if it holds $Eval[\![b]\!] \overset{q,\eta}{\Longrightarrow} b'$, where $b' = b[s]_q$. Hence $Eval[\![h[g]]\!] \overset{p,\sigma}{\Longrightarrow} (h[g])[s]_q$ admits a proof.

By Lemma 2.A.3 and inductive hypothesis $d' \in \mathcal{N}[\![g\eta \ in \ P]\!]$. Again, by inductive hypothesis $d'' \in \mathcal{N}[\![h'[g\eta] \ in \ P]\!]$ where $h'\colon x\vartheta\eta \to b'$. Similarly to the case for rule 2.2.7b (just replacing σ with η), we have that $h'[g\eta] = (h[g])[s]_q$ thus if $h[g] \overset{\eta}{\underset{p}{\Rightarrow}} d''$ is canonical then it belongs to $\mathcal{N}[\![h[g] \ in \ P]\!]$.

Lemma 2.A.5 *Let $P \in \mathbb{P}_\Sigma$, let σ be an idempotent $\mathcal{C}$-linear substitution of the form $\{x/c(\overrightarrow{y_n})\}$ and let t be a term renamed apart from $c(\overrightarrow{y_n})$. Then, for all $k \geq 0$ and for all $d_{g\sigma} \in \mathcal{N}_k[\![g\sigma \ in \ P]\!]$, it exists a derivation $d_g \in \mathcal{N}_k[\![g \ in \ P]\!]$ such that $d_g \vdash^\sigma d_{g\sigma}$.*

Proof.
The proof is similar to that of Lemma 2.A.6 but simpler, since the cases corresponding to rule 2.2.7b and 2.2.7d do not have to be considered.

Lemma 2.A.6 *Let $P \in \mathbb{P}_\Sigma$, let g, b be an embeddable pair, $x \in var(g) \setminus var(b)$, and let ϑ be a $\mathcal{C}$-linear substitution s.t. there exists a homomorphism $h\colon x\vartheta \to b$. Then, $\mathcal{N}_k[\![g\{x/b\} \ in \ P]\!] \sqsubseteq \mathcal{N}_k[\![g \ in \ P]\!][x/\mathcal{N}_k[\![b \ in \ P]\!]]$ for every $k \geq 0$*

Proof.
We show that for any given a $\mathcal{C}$-linear substitution ϑ such that $h\colon x\vartheta \to b$, for all $k \geq 0$ and $d \in \mathcal{N}_k[\![h[g] \ in \ P]\!]$ there exist two derivations $d_g \in \mathcal{N}_k[\![g \ in \ P]\!]$ and $d_b \in \mathcal{N}_k[\![b \ in \ P]\!]$ s.t. $d_g[x/d_b]_\vartheta \vdash d$ holds. We proceed by structural induction on the derivation d:

Base Case: d is a zero-step derivation of the form $h[g]$. This case immediately holds by rule 2.2.7a.

Inductive step: then d is of the form $h[g] \overset{\sigma}{\underset{p}{\Rightarrow}} d'$. By hypothesis the statement

$$Eval[\![h[g]]\!] \overset{p,\sigma}{\Longrightarrow} s \tag{1}$$

where s is the head of d' holds. Let $p_1, \ldots, p_n$ be the nodes traversed in the proof of (1). These nodes describe a path in $h[g]$ from the root to either a $\mathcal{V}$-labelled node or a $\mathcal{D}$-labelled one (respectively when the proof ends with (Ins) or (Rw)). The extension of the homomorphism $h\colon x\vartheta \to b$ on g is a homomorphism $h\colon g \to h[g]$. Let $q_1, \ldots, q_m$ with $m \leq n$ be the longest path in g such that $h(q_i) = p_i$ for every $1 \leq i \leq m$. From this, we distinguish two cases:

$\forall 1 \leq i \leq m.\ \mathcal{L}_g(q_i) \notin \mathcal{D}$: then, by (1) there exists an index $m < j \leq n$ such that $p_j = p$ and $\mathcal{L}_{h[g]}(p) \in \mathcal{D}$. Thus $\mathcal{L}_g(q_m) = y$ is a critical variable of g also belonging to $var(x\vartheta)$, and $h[y] = b|_{p_m}$. Therefore $h[g]|_p = h[y]|_p = (b|_{p_m})|_p$ is an outermost op-rooted subterm of $h[g]$. By (Cntx), a sub-tree of the proof tree of (1) is a proof tree for $Eval[\![b|_{p_m}]\!] \overset{p,\sigma}{\Longrightarrow} s'$, and $s = (h[g])[s']_{p_m}$. By $h\colon x\vartheta \to b$, ϑ $\mathcal{C}$-linear, $b|_p$ is an outermost op-rooted subterm of b. Hence, by (Cntx) also $Eval[\![b]\!] \overset{p,\sigma}{\Longrightarrow} b'$ with $b' = b[s']_{p_m}$ holds.

If (1) is a pure rewriting step then $\sigma = \varepsilon$ and it is easy to see that it exists an homomorphism $h'\colon x\vartheta \to b'$ s.t. $h'[g] = (h[g])[s']_{p_m} = s$. By hypothesis $d' \in \mathcal{N}_{k-1}[\![h'[g] \ in\ P]\!]$ then, by inductive hypothesis there exist $d_g \in \mathcal{N}_{k-1}[\![g \ in\ P]\!]$ and $d'_b \in \mathcal{N}_{k-1}[\![b' \ in\ P]\!]$ such that $d_g[x/d_{b'}]_{\vartheta\sigma} \vdash d'$. Therefore, by rule 2.2.7b, $d_g[x/b \overset{\sigma}{\underset{p}{\Rightarrow}} d_{b'}]_\vartheta \vdash d$ holds.

On the other hand, if (1) is an instantiation step then $\sigma \neq \varepsilon$, $s = h[g]\sigma$ and $b' = b\sigma$. By $h\colon x\vartheta \to b$, it also exists a homomorphism $h'\colon x\vartheta \to b\sigma$. By construction σ is idempotent, thus $h'[g\sigma] = h[g]\sigma = s$. Since (1) is an instantiation step, $d' \in \mathcal{N}_k[\![h'[g\sigma] \ in\ P]\!]$ then, by inductive hypothesis, there exist $d_{g\sigma} \in \mathcal{N}_{k-1}[\![g\sigma \ in\ P]\!]$ and $d'_b \in \mathcal{N}_{k-1}[\![b' \ in\ P]\!]$ such that $d_{g\sigma}[x/d_{b'}]_{\vartheta\sigma} \vdash d'$. By Lemma 2.A.5 there exists $d_g \in \mathcal{N}_{k-1}[\![g \ in\ P]\!]$ such that $d_g \vdash^\sigma d_{g\sigma}$. Again, by Equation (2.2.7b), $d_g[x/b \overset{\sigma}{\underset{p}{\Rightarrow}} d_{b'}]_\vartheta \vdash d$ holds.

Since $\mathcal{N}_{k-1}[\![g \ in\ P]\!] \sqsubseteq \mathcal{N}_k[\![g \ in\ P]\!]$ then $d_g \in \mathcal{N}_k[\![g \ in\ P]\!]$ and the thesis holds for both the cases.

$\exists 1 \leq i \leq m.\ \mathcal{L}_g(q_i) \in \mathcal{D}$: let j be the least index s.t. $\mathcal{L}_g(q_j) \in \mathcal{D}$. By construction $h[g]|_{p_j}$ is an outermost op-rooted subterm of $h[g]$, thus $p = p_j = h(q_j)$ in (1). By hypothesis ϑ is $\mathcal{C}$-linear, thus $h\colon x\vartheta \to b$ is $\mathcal{D}$-preserving, which means that $h(q_j) = q_j$ and $p = q_j$. We distinguish two possible cases

$m = n$) If $\mathcal{L}_g(q_n) \in \mathcal{D}$, by (1), $Eval[\![g]\!] \overset{p,\varepsilon}{\Longrightarrow} g'$ holds and (1) is a pure rewriting step, that is $\sigma = \varepsilon$ and $s = h[g']$. Then $d' \in \mathcal{N}_{k-1}[\![h[g'] \ in\ P]\!]$ and, by inductive hypothesis, there exist $d_{g'} \in \mathcal{N}_{k-1}[\![g' \ in\ P]\!]$ and $d_b \in \mathcal{N}_{k-1}[\![b \ in\ P]\!]$ s.t. $d_{g'}[x/d_b]_\vartheta \vdash d'$ holds. Hence, $(g \overset{\sigma}{\underset{p}{\Rightarrow}} d_{g'})[x/d_b]_\vartheta \vdash d$ holds by rule 2.2.7c.

Otherwise, we have that $\mathcal{L}_g(q_n) = y \in \mathcal{V}$. Then (1) is an instantiation step, that is $\sigma = \{y/c(\overrightarrow{z_n})\}$, $s = h[g]\sigma$ and $h[y] = w \in \mathcal{V}$. By (1) it follows that the statement $Eval[\![g]\!] \overset{p,\sigma}{\Longrightarrow} g'$ where $g' = g\sigma$, holds. By hypothesis $h\colon x\vartheta \to b$, thus it exists $h'\colon x\vartheta\sigma \to b\{w/c(\overrightarrow{z_n})\}$ and $h[g]\sigma = h'[g']$. By hypothesis $d' \in \mathcal{N}_{k-1}[\![h'[g'] \ in\ P]\!]$ then, by inductive hypothesis there exist $d_{g'} \in paths(\mathcal{N}_{k-1}[\![g' \ in\ P]\!])$ and $d'' \in paths(\mathcal{N}_{k-1}[\![b\sigma \ in\ P]\!])$ such that $d_{g'}[x/d'']_{\vartheta\sigma} \vdash d'$ holds. By Lemma 2.A.5, there exists $d_b \in \mathcal{N}_{k-1}[\![b \ in\ P]\!]$ such that $d_b[y/c(\overrightarrow{z_n})]_\varepsilon \vdash d''$ holds. Therefore, by rule 2.2.7f, $d_g[x/d_b]_\vartheta \vdash d$ holds.

By $\mathcal{N}_{k-1}[\![b \ in\ P]\!] \sqsubseteq \mathcal{N}_k[\![b \ in\ P]\!]$, $d_b \in \mathcal{N}_k[\![b \ in\ P]\!]$, therefore the thesis holds for both the former sub-cases.

$m < n$) In this case $\mathcal{L}_g(q_m) = y \in var(g) \cap var(x\vartheta)$ and, by Equation (1), there must be an instantiation step for g whose substitution θ is of the form $\{y/c(\overrightarrow{z_n})\}$, that is, the statement $Eval[\![g]\!] \overset{p,\theta}{\Longrightarrow} g'$ where $g' = g\theta$ holds.

There are three possible cases depending on the root symbol of $h[y] = b|_{p_m}$, namely $\mathcal{L}_b(p_m)$:

$\mathcal{L}_b(p_m) = w \in \mathcal{V}$: then (1) is an instantiation step, that is, $\sigma = \{w/c(\overrightarrow{z_n})\}$ and $s = h[g]\sigma$. As in the former case, the thesis holds by rule 2.2.7f.

$\mathcal{L}_b(p_m) \in \mathcal{D}$: by (1), $Eval[\![b|_{p_m}]\!] \xrightarrow{p_m,\sigma} s'$ where $s = (h[g])[s']_{p_m}$ holds. Since ϑ is $\mathcal{C}$-linear, $b|_{p_m}$ is an outermost op-rooted subterm of b, then the statement $Eval[\![b]\!] \xrightarrow{p_m,\sigma} b'$, where $b' = b[s']_{p_m}$, holds.

Let h' be $h'(q_m) := \mathcal{R}oot_{s'}$ and $h'(q) := h(q)$ for any node $q \neq q_m$. It can be proved that h' is a homomorphism $h' \colon x\vartheta \to b'$ and $(h[g])[s']_{p_m} = h'[g]$.

If (1) is a pure rewriting step then $d' \in \mathcal{N}_{k-1}[\![h'[g] \mathit{in} P]\!]$ and, by inductive hypothesis, there exist $d_g \in \mathcal{N}_{k-1}[\![g \mathit{\ in\ } P]\!]$ and $d_{b'} \in \mathcal{N}_{k-1}[\![b' \mathit{\ in\ } P]\!]$ such that $d_g[x/d_{b'}]_{\vartheta\sigma} \vdash d'$. In particular, by (1) and $d \in \mathcal{N}_k[\![h[g] \mathit{\ in\ } P]\!]$, d_g must be of the form $g \overset{\theta}{\underset{p}{\Rightarrow}} d_{g'}$, where $d_{g'} = \mathcal{N}[\![g' \mathit{\ in\ } P]\!]$. Thus, by rule 2.2.7d, $d_g[x/d_b]_\vartheta \vdash d$ holds. Since $\mathcal{N}_{k-1}[\![b \mathit{\ in\ } P]\!] \sqsubseteq \mathcal{N}_k[\![b \mathit{\ in\ } P]\!]$ then $d_b \in \mathit{paths}(\mathcal{N}_k[\![b \mathit{\ in\ } P]\!])$ hence the thesis holds for this sub-case.

If (1) is an instantiation step then $b' = b\sigma$. By hypothesis $d' \in \mathcal{N}_{k-1}[\![h'[g\sigma] \mathit{\ in\ } P]\!]$, hence by inductive hypothesis there exist $d_{g\sigma} \in \mathcal{N}_{k-1}[\![g\sigma \mathit{in} P]\!]$ and $d_{b'} \in \mathcal{N}_{k-1}[\![b' \mathit{in} P]\!]$ s.t. $d_{g\sigma}[x/d_{b'}]_{\vartheta\sigma} \vdash d'$. By Lemma 2.A.5, there exist $d_g \in \mathcal{N}_{k-1}[\![g \mathit{\ in\ } P]\!]$ such that $d_g \vdash^\sigma d_{g\sigma}$. As the previously mentioned d_g must be of the form $g \overset{\theta}{\underset{p}{\Rightarrow}} d_{g'}$, where $d_{g'} = \mathcal{N}[\![g' \mathit{\ in\ } P]\!]$. Thus by rule 2.2.7d, $d_g[x/b \overset{\sigma}{\underset{p}{\Rightarrow}} d_{b'}]_\vartheta \vdash d$ holds and therefore the thesis.

$\mathcal{L}_b(p_m) \in \mathcal{C}$: $\mathcal{L}_b(p_m) = c$ since (1) holds. Thus, by $h \colon x\vartheta \to b$ there exists a homomorphism $h' \colon x\vartheta\theta \to b$, hence $h'[g\theta] = h'[g'] = h[g]$. By (1), $Eval[\![h'[g']]\!] \xrightarrow{p,\sigma} s$ holds. Then, by rule 2.2.7e the problem is reduced to check whether exist $d_{g'} \in \mathcal{N}_{k-1}[\![g' \mathit{\ in\ } P]\!]$ and $d_b \in \mathcal{N}_{k-1}[\![b \mathit{\ in\ } P]\!]$ such that $d_{g'}[x/d_{b'}]_{\vartheta\theta} \vdash d$. Actually, by (1), $\mathcal{L}_{h[g]}(p_n) \notin \mathcal{C}$ thus this check is eventually reduces in one of the previous cases.

Now we can state the small-step operational denotations are "compositional" in this sense:

Corollary 2.A.7 *Let $P \in \mathbb{P}_\Sigma$, let g, b be an embeddable pair, $x \in var(g) \setminus var(b)$, and let ϑ be a $\mathcal{C}$-linear substitution s.t. there exists a homomorphism $h \colon x\vartheta \to b$. Then, $\mathcal{N}[\![g \mathit{\ in\ } P]\!][x/\mathcal{N}[\![b \mathit{\ in\ } P]\!]] = \mathcal{N}[\![g\{x/b\} \mathit{\ in\ } P]\!]$.*

Proof of Corollary 2.A.7.
The proof directly follows from Lemmata 2.A.4 and 2.A.6 since for all $P \in \mathbb{P}_\Sigma$ and for all $e \in \mathcal{T}(\Sigma, \mathcal{V})$, $\mathcal{N}[\![e \mathit{\ in\ } P]\!] = \bigsqcup\{\mathcal{N}_k[\![e \mathit{\ in\ } P]\!] \mid k \geq 0\}$.

A straightforward consequence of Corollary 2.A.7 is Theorem 2.2.19.

Proof of Theorem 2.2.19.
We proceed by induction on the cardinality of $\mathit{baseknots}(e)$

Base Case: There are two possible cases for e:

$\boldsymbol{e \in \mathcal{V}}$) Immediate by Equation (2.2.5a)

$\boldsymbol{e = \varphi(\overrightarrow{x_n})}$) By induction on n. The case $n = 0$ is straightforward by Equations (2.2.5b) and (2.2.4). For $n > 0$ the following hold

$$\begin{aligned}
&\mathcal{E}[\![\varphi(\overrightarrow{x_n})]\!]_{\mathcal{O}[\![P]\!]} && [\text{by Equation (2.2.5b)}]\\
&= \mathcal{O}[\![P]\!](\varphi(\overrightarrow{y_n}))[y_1/x_1]\dots[y_n/x_n] && [\text{by inductive hypothesis on } n]\\
&= \mathcal{N}[\![\varphi(\overrightarrow{x_{n-1}}, y_n) \mathit{\ in\ } P]\!][y_n/x_n] && [x_n = \mathcal{N}[\![x_n \mathit{\ in\ } P]\!] \text{ and Corollary 2.A.7}]\\
&= \mathcal{N}[\![\varphi(\overrightarrow{x_n}) \mathit{\ in\ } P]\!]
\end{aligned}$$

Inductive Step: Let p be the leftmost node in $\mathit{baseknots}(e)$ and y a fresh variable. We have that

$$\begin{aligned}
&\mathcal{E}[\![e]\!]_{\mathcal{O}[\![P]\!]} = && [\text{Equation (2.2.5c)}]\\
&= \mathcal{E}[\![e[y]_p]\!]_{\mathcal{O}[\![P]\!]}[y/\,\mathcal{E}[\![e|_p]\!]_{\mathcal{O}[\![P]\!]}] && [\text{by inductive hypothesis}]\\
&= \mathcal{N}[\![e[y]_p \mathit{\ in\ } P]\!][y/\,\mathcal{N}[\![e|_p \mathit{\ in\ } P]\!]] && [\text{by Corollary 2.A.7}]\\
&= \mathcal{N}[\![e \mathit{\ in\ } P]\!]
\end{aligned}$$

Proof of Corollary 2.2.20.
Straightforward consequence of Theorems 2.2.19 and 2.2.15.

Proof of Proposition 2.2.23.
It is straightforward to prove that $\mathcal{P}[\![P]\!]$ is monotone and finitary, thus it is continuous.

Proof of Theorem 2.2.25.
$\sqsubseteq$) First we prove that given $f \in \mathcal{D}$ the definitional tree T for f in P,

$$\xi[\![T]\!]_{\mathcal{O}[\![P]\!]} \sqsubseteq \mathcal{N}[\![pat(T) \mathit{\ in\ } P]\!]. \tag{1}$$

We proceed structural by induction on T.

***T* = *rule*($\boldsymbol{\pi \to r}$):** By construction $\pi \to r \in P$ is a rule for f in P. By (2.2.10a) $\xi[\![T]\!]_{\mathcal{O}[\![P]\!]} = \pi \xrightarrow[\mathcal{R}oot_l]{\varepsilon} \mathcal{E}[\![r]\!]_{\mathcal{O}[\![P]\!]}$. Then, by (Rw) and Theorem 2.2.19, $\xi[\![T]\!]_{\mathcal{O}[\![P]\!]} \sqsubseteq \mathcal{N}[\![pat(T) \mathit{\ in\ } P]\!]$.

***T* = *or*($\boldsymbol{T_1, T_2}$):** By construction $pat(T) = pat(T_1) = pat(T_2)$ and, by inductive hypothesis, $T_i \sqsubseteq \mathcal{N}[\![pat(T_i) \mathit{\ in\ } P]\!]$ for $i = 1, 2$. Therefore, by Equation (2.2.10b), $\xi[\![T]\!]_{\mathcal{O}[\![P]\!]} \sqsubseteq \mathcal{N}[\![pat(T) \mathit{\ in\ } P]\!]$.

***T* = *branch*($\boldsymbol{\pi, p, \overrightarrow{T_m}}$):** by Equation (2.2.10c)

$$\xi[\![branch(\pi, p, \overrightarrow{T_m})]\!]_{\mathcal{O}[\![P]\!]} = \bigsqcup_{i=1}^{m} \left\{ \pi \xrightarrow[\mathcal{R}oot_\pi]{\{x/\pi'\}} \xi[\![T_i]\!]_{\mathcal{O}[\![P]\!]} \;\middle|\; \begin{array}{l} x = \pi|_p \\ \pi' = pat(T_i)|_p \end{array} \right\}$$

By Equation (Ins) and Definition 1.4.1, $\mathit{Eval}[\![\pi]\!] \xrightarrow{p,\sigma_i} pat(T_i)$ with $\sigma_i := \{\pi|_p/\, pat(T_i)|_p\}$ for every $1 \le i \le m$. By inductive hypothesis we conclude that $\xi[\![T]\!]_{\mathcal{O}[\![P]\!]} \sqsubseteq \mathcal{N}[\![pat(T) \mathit{\ in\ } P]\!]$.

We conclude the proof showing that $\mathcal{O}[\![P]\!]$ is a pre-fixed point of $\mathcal{P}[\![P]\!]$:

$$
\begin{aligned}
&\mathcal{P}[\![P]\!]_{\mathcal{O}[\![P]\!]} = \\
&\quad [\text{by Equation (2.2.9)}] \\
&= \lambda\varphi(\overrightarrow{x_n}).\begin{cases} \varphi(\overrightarrow{x_n}) & \text{if } \varphi \in \mathcal{C} \\ \xi[\![T]\!]_{\mathcal{O}[\![P]\!]} & \text{otherwise, } T \text{ fresh definitional tree for } \varphi \\ & \text{s.t. } pat(T) = \varphi(\overrightarrow{x_n}) \end{cases} \\
&\quad [\text{by (1)}] \\
&\sqsubseteq \mathcal{O}[\![P]\!]
\end{aligned}
$$

$\sqsupseteq$) By contradiction. Assume that there exists $\varphi_{/n} \in \Sigma$ and a path d such that $d \sqsubseteq \mathcal{O}[\![P]\!](\varphi(\overrightarrow{x_n}))$ but $d \not\sqsubseteq \mathcal{F}[\![P]\!](\varphi(\overrightarrow{x_n}))$. Let indicate with $N_m := \lambda\varphi(\overrightarrow{x_n}).\, \mathcal{N}_m[\![\varphi(\overrightarrow{x_n})\ in\ P]\!]$ for any $m \geq 0$. Suppose d be one of the shortest paths among those holding the former condition and let k be the number of rewriting steps of d. Consider the two possible cases:

- $\boldsymbol{k = 0}$) by hypothesis $d \sqsubseteq N_0(\varphi(\overrightarrow{x_n}))$ but $d \not\sqsubseteq \mathcal{F}[\![P]\!](\varphi(\overrightarrow{x_n}))$. By Definition 2.2.10 and (2.2.9), $N_0 \sqsubseteq \mathcal{P}[\![P]\!]_{\perp_{\mathbb{I}}}$. By monotonicity of $\mathcal{P}[\![P]\!]$ and $\mathcal{P}[\![P]\!]_{\mathcal{F}[\![P]\!]} = \mathcal{F}[\![P]\!]$, $N_0 \sqsubseteq \mathcal{F}[\![P]\!]$, which is absurd.
- $\boldsymbol{k > 0}$) by hypothesis $d \sqsubseteq N_k(\varphi(\overrightarrow{x_n}))$ but $d \not\sqsubseteq \mathcal{F}[\![P]\!](\varphi(\overrightarrow{x_n}))$. In fact $N_{k-1} \sqsubseteq \mathcal{F}[\![P]\!]$, otherwise there exists a shorter path holding the former condition. By Equation (2.2.9) and Lemma 2.A.6 $N_k \sqsubseteq \mathcal{P}[\![P]\!]_{N_{k-1}}$. By monotonicity of $\mathcal{P}[\![P]\!]$ and $\mathcal{P}[\![P]\!]_{\mathcal{F}[\![P]\!]} = \mathcal{F}[\![P]\!]$, $N_k \sqsubseteq \mathcal{F}[\![P]\!]$, which is absurd.

Proof of Corollary 2.2.26.
Straightforward consequence of Theorem 2.2.25 and Corollary 2.2.20.

3

Big-step semantics

Abstract

We present a condensed, goal-independent, bottom-up fixpoint semantics that is fully abstract w.r.t. results computed for Curry expressions. This work is motivated by the fact that a (condensed) goal-independent semantics for functional logic languages, which is essential for the development of efficacious semantics-based program manipulation tools (e.g. automatic program analyzers and debuggers), does not exist. The operational or the rewriting logic semantics that are most commonly considered in the functional logic paradigm are unnecessarily oversized, as they contains many "semantically useless" elements that can be retrieved from a smaller set of "essential" elements.

We believe that the condensedness of the semantics makes it particularly suitable for applications. Actually we have already developed some applications of the presented semantics to the automatic debugging field which gave interesting results.

3.1 Modeling the Big-Step Operational Behaviour of Curry programs

In this section, as anticipated in the introduction, we derive (by abstraction of the semantics of Subsection 2.2.4) a fixpoint big-step semantics.

We first have to formally define the concept of big-step operational behaviour, which can simply be the collection of all outcomes corresponding to value-terminating small-step derivations[1]. In the FLP case the outcome, corresponding to value-terminating small-step derivations, consists of a computed answer ϑ *and* a computed value v. To the best of our knowledge in the literature there is no well established name for this pair, thus in the following we will call it *computed result* (and denote it by $\vartheta \cdot v$).

Definition 3.1.1 (Computed result behaviour of programs) *Let* $P \in \mathbb{P}_\Sigma$,

$$\mathcal{B}^{cr}[\![P]\!] := \lambda e.\left\{\sigma \cdot \wr v \int \;\middle|\; e \overset{\sigma}{\Rightarrow}{}^{*}\, v \in \mathcal{B}^{ss}[\![P]\!],\, v \in \mathcal{T}(\mathcal{C},\mathcal{V})\right\} \tag{3.1.1}$$

We indicate with $\approx_{cr}$ *the program equivalence relation induced by* $\mathcal{B}^{cr}$*, namely* $P_1 \approx_{cr} P_2 \Leftrightarrow \mathcal{B}^{cr}[\![P_1]\!] = \mathcal{B}^{cr}[\![P_2]\!]$.

[1] A direct definition of the big-step semantics with big-step rules as those given for the small-step version is unnecessary for our current purposes

Note that in (3.1.1) we need to unravel the v computed by the small-step semantics ($\wr v \int$) because we *must not* distinguish sharing in non-variable nodes.

Unfortunately, unlike the small-step case, because of laziness it is not enough to collect just the computed results of most general calls in order to provide a goal-independent denotation correct w.r.t. the behaviour of computed results. Consider for example the programs

```
f x = S (g x)
g (S x) = 0
```

```
f (S x) = S 0
g (S x) = 0
```

which have the same computed results for most general calls, namely

$$\begin{cases} f(x) \mapsto \{\{x/s(x')\} \cdot s(0)\} \\ g(x) \mapsto \{\{x/s(x')\} \cdot 0\} \end{cases}$$

For the goal $g(f(x))$ the former program computes $\varepsilon \cdot 0$ but the latter computes $\{x/s(x')\} \cdot 0$. Thus a collecting semantics maintaining *only* the information of computed result for most general calls would erroneously identify this two non-equivalent programs. Some additional information about the way in which a computed result is obtained must be kept. However we do not want to resort to use all the information of a small-step derivation.

In this section we develop a goal-independent big-step semantics by optimal abstraction of the (much more) concrete fixpoint semantics in Subsection 2.2.4 through an abstraction which hides the information about intermediate defined function symbols from the observable behaviour of the program, while retaining the information which is needed in order to preserve correctness w.r.t. the behaviour of computed results.

The idea is to look within all abstract interpretations which are obtained from a tree abstraction $\alpha\colon (\mathbb{WSST}, \sqsubseteq) \to (\mathbb{A}, \leq)$. This tree abstraction can by systematically lifted to a Galois Insertion $\mathbb{I} \underset{\bar{\alpha}}{\overset{\bar{\gamma}}{\leftrightarrows}} [\mathbb{MGC} \to \mathbb{A}]$ by function composition (i.e., $\bar{\alpha}(f) = \alpha \circ f$).

Then we can derive the optimal abstract version of $\mathcal{P}^\alpha[\![P]\!]$ simply as $\mathcal{P}^\alpha[\![P]\!] := \bar{\alpha} \circ \mathcal{P}[\![P]\!] \circ \bar{\gamma}$.

As recalled in Section 1.5, abstract interpretation theory assures that $\mathcal{F}^\alpha[\![P]\!] := \mathit{lfp}_{\mathbb{A}}(\mathcal{P}^\alpha[\![P]\!])$ is the best correct approximation of $\mathcal{F}[\![P]\!]$. Correct means $\bar{\alpha}(\mathcal{F}[\![P]\!]) \leq \mathcal{F}^\alpha[\![P]\!]$ and best means that it is the minimum (w.r.t. $\leq$) of all correct approximations.

If $\bar{\alpha}$ is *precise*, i.e., $\mathcal{P}^\alpha[\![P]\!] \circ \bar{\alpha} = \bar{\alpha} \circ \mathcal{P}[\![P]\!]$ (or equivalently $\bar{\alpha} \circ \mathcal{P}[\![P]\!] \circ \bar{\gamma} \circ \bar{\alpha} = \bar{\alpha} \circ \mathcal{P}[\![P]\!]$), then $\bar{\alpha}(\mathcal{F}[\![P]\!]) = \mathcal{F}^\alpha[\![P]\!]$. This means that $\mathcal{F}^\alpha[\![P]\!]$ is not just a generic (safe) approximation, but a (precise) abstract *semantics*.

Expressed it this terminology our goal is to find a precise abstraction α such that the induced abstract semantics $\mathcal{F}^\alpha[\![P]\!]$ is fully abstract w.r.t. $\approx_{cr}$.

3.1.1 The semantic domain

To deal with non-strict operations, without considering all the details about expressions still to be solved of the small-step behaviour, [49, 50, 64] consider constructor signatures $\mathcal{C}_\perp$ extended by a special constructor symbol $\perp$ to represent undefined values. Such partial terms are considered as finite approximations of possibly infinite values. However, by using just one abstraction symbol $\perp$ it is impossible to distinguish between different occurrences of the same abstractions. So we use instead an additional denumerable set of variables $\mathcal{V}_\varrho$, obviously disjoint from $\mathcal{V}$, and we abstract a term graph by replacing all its op-rooted

subterms with (fresh) variables from $\mathcal{V}_\varrho$. When it is not clear from the context, in analogy with [50], we will call variables in $\mathcal{V}_\varrho$ *bottom variables*, while variables in $\mathcal{V}$ *data variables*.

Moreover in the following we denote by $\varrho Substs$ the set of substitutions from $\mathcal{V}_\varrho$ to $\mathcal{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho)$.

Formally, the term abstraction $\tau : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathcal{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho)$ is defined as

$$\tau(e) := e[\varrho_1]_{p_1} \dots [\varrho_n]_{p_n} \tag{3.1.2}$$

where $p_1, \dots, p_n$ are the positions of *all* outermost op-rooted subterms of e and $\varrho_1, \dots, \varrho_n$ are fresh bottom variables. A partial computed result $\sigma \cdot e$ is abstracted to $\sigma \cdot \tau(e)$ (since $\sigma \in \mathcal{C}Substs$ does not contain op-rooted subterms). We call the latter *τ-abstract partial computed result* or simply partial computed result for brevity, when its clear from the context.

Abstracting small-step trees to evolving result trees

Conceptually, given a canonical small-step evaluation sequence $d = e_0 \xhookrightarrow[p_1]{\vartheta_1} \cdots \xhookrightarrow[p_n]{\vartheta_n} e_n$, we can think of applying the term abstraction τ to all terms of d. All terms s_j of d which are between e_i and e_{i+1} in position p_i will be abstracted to the same bottom variable ϱ (i.e., $\tau(s_j)|_{p_i} = \tau(e_i)|_{p_i} = \varrho \in \mathcal{V}_\varrho$) as all evaluation steps that change an op-rooted subterm into another op-rooted subterm are abstractly identical. With regard to neededness of a redex, all these op-rooted subterms are equivalent. Things change when evaluation introduces a new constructor, i.e., when we reach a head normal form (and the evaluation position changes to p_{i+1}). In this case we have a different term abstraction as ϱ changes to a non-bottom variable (i.e., $\tau(e_{i+1})|_{p_i} \notin \mathcal{V}_\varrho$).

Intuitively, starting from a term e, we want to collapse together all intermediate small steps related to the same op-rooted subterm which lead to an head normal form. All positions along the way (of these sub-derivations) refer to an op-rooted subterm that is abstracted to some bottom variable ϱ and we need to explicitly remember this information (otherwise we could not distinguish needed redexes from subterms that change because of sharing). Hence we keep ϱ as label of an abstract computation step $\vartheta \cdot \bar{t} \xrightarrow{\varrho} \sigma \cdot \bar{s}$ going from the abstract partial computed result $\vartheta \cdot \bar{t}$ to $\sigma \cdot \bar{s}$.

We can see the process from the point of view of a partial computed result $\vartheta \cdot \bar{t}$, which contains a bottom variable ϱ in position p, corresponding on the small-step side to an outermost op-rooted subterm in position p. What we do is to collapse together all small-steps until at least another constructor is added (see Examples 3.1.2, 3.1.3, 3.1.4 and 3.1.5). Then $\bar{t}$ becomes $\bar{s} \equiv \bar{t}\{\varrho/\bar{r}\}$ and also ϑ becomes $\sigma \equiv \vartheta\eta$, giving the abstract step $\vartheta \cdot \bar{t} \xrightarrow{\varrho} \sigma \cdot \bar{s}$.

Altogether we obtain a sequence of abstract steps $\theta_0 \cdot \bar{t}_0 \xrightarrow{\varrho_1} \dots \xrightarrow{\varrho_m} \theta_m \cdot \bar{t}_m$ which we call big-step sequence.

Given a set V of relevant variables and an initial substitution σ, with $dom(\sigma) \subseteq V$, we can formally define the abstraction (into a big-step sequence) of a small-step sequence d, starting from t, by structural induction as

$$\partial_V^\sigma(d) := \begin{cases} \sigma \cdot \bar{t} \xrightarrow{\varrho} \partial_V^\eta(d') & \text{if } d = e \xhookrightarrow[p]{\vartheta} d' \\ \sigma \cdot \bar{t} & \text{otherwise} \end{cases} \tag{3.1.3}$$

where $\bar{e} = \tau(t)$, $h\colon e \to \bar{t}$, $\varrho = \mathcal{L}(h(p))$ and $\eta = (\sigma\vartheta)\restriction_V$.

The intuition of Equation (3.1.3) is that when we abstract $\cdots t \overset{\vartheta}{\underset{p}{\Rightarrow}} s \cdots$ (having an accumulated substitution σ) we have the following alternatives:

1. subterm $s|_p$ is an head normal form. Thus we build an abstract step over the bottom variable of position p (and we continue with accumulator $\sigma\vartheta$);
2. subterm $s|_p$ is not an head normal form. Thus we just skip to next term (updating the accumulator with ϑ).

Definition (3.1.3) is extended to trees T of $\mathbb{WSST}_e$ as

$$\partial_{var(e)}(T) := prfxtree(\bigcup\left\{\partial^{\varepsilon}_{var(e)}(d) \;\middle|\; d \in paths(T)\right\}) \tag{3.1.4}$$

where for any set S of big-step sequences $prfxtree(S)$ computes the tree (whose nodes are abstract partial computed results and edges are labelled over $\mathcal{V}_\varrho$) whose paths are exactly S.

Example 3.1.2

Consider $\mathcal{F}[\![P]\!]$ of Example 2.2.21. We have

$$\partial^{\varepsilon}_{\emptyset}(coin \overset{\varepsilon}{\Rightarrow} Head) = \varepsilon \cdot \varrho \xrightarrow{\varrho} \varepsilon \cdot Head \qquad \partial^{\varepsilon}_{\emptyset}(coin \overset{\varepsilon}{\Rightarrow} Tail) = \varepsilon \cdot \varrho \xrightarrow{\varrho} \varepsilon \cdot Tail$$

and thus

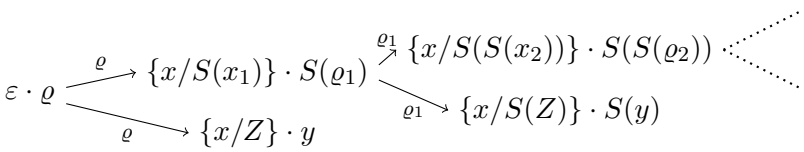
$$\partial_{\emptyset}\left(\begin{array}{c} coin \overset{\varepsilon}{\Rightarrow} Head \\ coin \underset{\varepsilon}{\Rightarrow} Tail \end{array}\right) = \begin{array}{c} \varepsilon \cdot \varrho \xrightarrow{\varrho} \varepsilon \cdot Head \\ \varepsilon \cdot \varrho \xrightarrow[\varrho]{} \varepsilon \cdot Tail \end{array}$$

Moreover

$$\partial_{\emptyset}(zeros \overset{\varepsilon}{\Rightarrow} Z : zeros \overset{\varepsilon}{\Rightarrow} Z : Z : zeros \overset{\varepsilon}{\Rightarrow} Z : Z : Z : zeros \overset{\varepsilon}{\Rightarrow} \ldots) =$$
$$\varepsilon \cdot \varrho_1 \xrightarrow{\varrho_1} \varepsilon \cdot Z : \varrho_2 \xrightarrow{\varrho_2} \varepsilon \cdot Z : Z : \varrho_3 \xrightarrow{\varrho_3} \varepsilon \cdot Z : Z : Z : \varrho_4 \xrightarrow{\varrho_4} \ldots$$

Example 3.1.3

The abstraction of $\mathcal{N}[\![x + y \ in \ P_+]\!]$ of Example 2.2.13 is

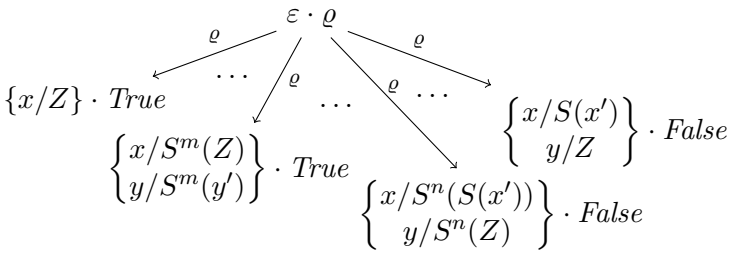


Example 3.1.4

The abstraction of the small-step semantics of the program $P_{\leq}$

```
leq (S x) Z = False
leq (S x) (S y) = leq x y
```

is



These examples show that big-step trees can be infinite in two ways.

- The one associated to $x + y$ has *infinite height*, since $+$ builds the solution one S at a time.
- The one associated to $leq(x, y)$ has height 1 but *infinite width*, since *leq* delivers just value true or false, while binding x and y in infinitely many different combinations.

Example 3.1.5

The abstraction of small-step tree of Example 2.2.24 is

$$\begin{cases} f(x) \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} \{x/A(x_1)\} \cdot B(\varrho_1) \\ loop(x) \mapsto \varepsilon \cdot \varrho \\ g(x) \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} \{x/A(x_1)\} \cdot B(\varrho_1) \\ h(x) \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} \{x/B\} \cdot C \end{cases}$$

Note that (as anticipated in Example 2.2.24) the abstractions of f and g are the same.

In general normal forms, which are not values, simply disappears in the abstraction. Furthermore, also non terminating (infinite) small-step paths disappear in the abstraction as well, and thus become indistinguishable from normal forms which are not values (both situations correspond to some sort of anomaly in the program).

This will mean that we cannot handle certain kinds of non-termination, which should not come as a surprise as this derivations do not produce any computed results thus do not appear in the big-step behaviour. With this abstraction we clearly cannot "observe" derivations which do not produce computed results, either because we reach a non-value normal form (a non total function) or because we are in a loop which is not producing constructor bindings either in the result or in the free variables.

Evolving result trees

The trees that can be obtained by ∂_V abstraction have the property that any path describes the evolution of data constructors as a partial computed result is built by a computation.

In other words, they describe, along a path, how a partial computed result evolves. Thus, as a matter of terminology, we call such trees *evolving result trees*.

Our fixpoint semantics is based on interpretations that consist of families of evolving result trees, indexed by $\mathbb{MGC}$ (analogously to the small-step case). The *evolving result tree abstraction* is obtained by pointwise lifting of ∂_V as

$$\partial(\mathcal{I}) := \lambda\pi.\ \partial_{var(\pi)}(\mathcal{I}(\pi)). \tag{3.1.5}$$

As happens frequently in abstract interpretation, in the following we derive systematically the abstract domain (both its support and order), as well as the corresponding Galois insertion, from the abstraction ∂.

First we define the (support of the) domain of evolving result tree semantics as $\mathbb{I}_{\mathbb{ERT}} := \partial(\mathbb{I}) = [\mathbb{MGC} \to \mathbb{ERT}]$. As done in the concrete case we denote with $\mathbb{ERT}_{\overrightarrow{x_n}}$ the sub-lattice of evolving result trees referring to variables $\overrightarrow{x_n}$, namely[2]

$$\mathbb{ERT}_{\overrightarrow{x_n}} := \{\partial_{\{\overrightarrow{x_n}\}}(T) \,|\, T \in \mathbb{WSST}_{f(\overrightarrow{x_n})}, f/n \in \mathcal{D}\} \tag{3.1.6}$$

For the sake of comprehension, we can give some properties of $\mathbb{ERT}$ elements by means of the following (auxiliary) definition.

Definition 3.1.6 (evolving relation) *Given two partial computed results $\vartheta \cdot \bar{s}$, $\vartheta' \cdot \bar{s}'$ and $\varrho \in \mathcal{V}_\varrho$ we say that $\vartheta' \cdot \bar{s}'$* is an evolution of $\vartheta \cdot \bar{s}$ w.r.t. ϱ, *written $\vartheta \cdot \bar{s} \twoheadrightarrow_\varrho \vartheta' \cdot \bar{s}'$, if and only if $\varrho \in var(\bar{s})$ and there exist $\sigma \in \mathcal{C}Substs$ and $\hat{\rho} \in \varrho Substs$ such that $\varrho\hat{\rho} \notin \mathcal{V}_\varrho$, $\vartheta\sigma = \vartheta'$, and $\bar{s}\sigma\hat{\rho} = \bar{s}'$.*

Proposition 3.1.7 ($\mathbb{ERT}_{\overrightarrow{x_n}}$ properties) *For all $\bar{T} \in \mathbb{ERT}_{\overrightarrow{x_n}}$*

1. *The substitution of the root of $\bar{T}$ is ε.*
2. *for any edge $a \xrightarrow{\varrho} a'$ of $\bar{T}$, $a \twoheadrightarrow_\varrho a'$,*
3. *all sibling nodes in $\bar{T}$ are different, and*
4. *for any path in $\bar{T}$ from a to b such that $a \twoheadrightarrow_\varrho b$ there is a path (in $\bar{T}$) from a to a suitable variant b' of b whose first edge is labelled ϱ.*

Point 4 specifies that in case a node a has more than one bottom variable (in presence of multiple outermost op-rooted redexes), and one of these bottom variables ϱ is eventually reduced along some path, then there will be an equivalent path starting with an immediate reduction of ϱ. In other words, like the small-step trees case, there is a sort of closedness upon permutation of the order of instantiation (reduction) of multiple bottom variables. However note that different orders in reduction may lead to paths of different length (but with the same leaves, i.e., computed result) since, because of sharing, more than one bottom variable can change simultaneously in a single abstract step (as shown by the following example).

Example 3.1.8

Consider the program P

[2] Note that the only relevant information in this construction is the arity of the defined symbol since, for all symbols of the same arity, the result is the same.

```
main = g (id B)                    h B  = A
g x  = C x (h x)
```

the fixpoint $\mathcal{F}[\![P]\!]$ is

$$\begin{cases} h(x) \mapsto h(x) \xrightarrow{\{x/B\}} h(B) \overset{\varepsilon}{\Rightarrow} A \\ g(x) \mapsto g(x) \overset{\varepsilon}{\Rightarrow} C(x, h(x)) \xrightarrow[2]{\{x/B\}} C(B, h(B)) \overset{\varepsilon}{\underset{2}{\Rightarrow}} C(B, A) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad C(B, h(B)) \overset{\varepsilon}{\underset{2}{\nRightarrow}} C(B, A) \\ main \mapsto main \overset{\varepsilon}{\Rightarrow} g(id(B)) \overset{\varepsilon}{\Rightarrow} \; C(id(B), h(id(B))) \overset{\varepsilon}{\underset{1}{\Rightarrow}} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad C(B, h(B)) \overset{\varepsilon}{\underset{2}{\nRightarrow}} C(B, A) \end{cases}$$

and its abstraction is

$$\begin{cases} h(x) \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} \{x/B\} \cdot A \\ g(x) \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} C(x, \varrho_1) \xrightarrow{\varrho_1} \{x/B\} \cdot C(B, A) \\ main \mapsto \; \varepsilon \cdot \varrho \xrightarrow{\varrho} \varepsilon \cdot C(\varrho_1, \varrho_2) \overset{\varrho_2}{\underset{\varrho_1}{\rightrightarrows}} \varepsilon \cdot C(B, A) \xrightarrow{\varrho_2} \varepsilon \cdot C(B, A) \end{cases}$$

Now we can derive systematically order, *lub* and *glb* of $\mathbb{I}_{\mathbb{ERT}}$ as

$$\mathcal{I}_1^\partial \preccurlyeq \mathcal{I}_2^\partial :\Longleftrightarrow \mathcal{I}_1^\partial \curlyvee \mathcal{I}_2^\partial = \mathcal{I}_2^\partial \tag{3.1.7}$$

$$\bigvee \Im := \partial(\bigsqcup\{\mathcal{I} \in \mathbb{I} \,|\, \partial(\mathcal{I}) \in \Im\}) \tag{3.1.8}$$

$$\bigwedge \Im := \partial(\bigsqcap\{\mathcal{I} \in \mathbb{I} \,|\, \partial(\mathcal{I}) \in \Im\}) \tag{3.1.9}$$

We have that $\mathbb{I}_{\mathbb{ERT}}$ is, by construction, a complete lattice. The order on $\mathbb{I}_{\mathbb{ERT}}$ also induces an order on $\mathbb{ERT}_{\overrightarrow{x_n}}$ as

$$\bar{T}_1 \preccurlyeq \bar{T}_2 :\Longleftrightarrow \bar{T}_1 \curlyvee \bar{T}_2 = \bar{T}_2 \tag{3.1.10}$$

$$\bigvee \bar{\mathcal{T}} := \partial_{\{\overrightarrow{x_n}\}}(\bigsqcup\left\{T \in \mathbb{WSST}_{f(\overrightarrow{x_n})} \;\middle|\; f/n \in \mathcal{D},\, \partial_{\{\overrightarrow{x_n}\}}(T) \in \bar{\mathcal{T}}\right\}) \tag{3.1.11}$$

$$\bigwedge \bar{\mathcal{T}} := \partial_{\{\overrightarrow{x_n}\}}(\bigsqcap\left\{T \in \mathbb{WSST}_{f(\overrightarrow{x_n})} \;\middle|\; f/n \in \mathcal{D},\, \partial_{\{\overrightarrow{x_n}\}}(T) \in \bar{\mathcal{T}}\right\}) \tag{3.1.12}$$

These (indirect) definitions may seem quite obscure, but the following result shows that the (actual) direct characterization of these notions is absolutely clear.

Proposition 3.1.9 ($\preccurlyeq$ characterization) *Given $\bar{T}_1, \bar{T}_2 \in \mathbb{ERT}_{\overrightarrow{x_n}}$ then $\bar{T}_1 \preccurlyeq \bar{T}_2$ if and only if $paths(\bar{T}_1) \subseteq paths(\bar{T}_2)$.*

$\bigvee \bar{\mathcal{T}}$ is the tree whose paths are $\bigcup_{\bar{T} \in \bar{\mathcal{T}}} paths(\bar{T})$. Dually for $\bigwedge \bar{\mathcal{T}}$.

Order, lub and glb over $\mathbb{I}_{\mathbb{ERT}}$ are the pointwise lifting of those over $\mathbb{ERT}_{\overrightarrow{x_n}}$.

The concretization function $\partial^\gamma \colon [\mathbb{MGC} \to \mathbb{ERT}] \to \mathbb{I}$ can be provided by adjunction, obtaining

$$\partial^\gamma(\mathcal{I}^\partial) = \lambda\pi.\; \partial^\gamma_\pi(\mathcal{I}^\partial(\pi)) \tag{3.1.13}$$

where, for all $e \in \mathcal{T}(\Sigma, \mathcal{V})$,

$$\partial_e^\gamma(\bar{T}) := \bigsqcup \{ T \in \mathbb{WSST}_e \mid \partial_{var(e)}(T) \preccurlyeq \bar{T} \} \tag{3.1.14}$$

3.1.2 The semantics induced by the evolving result tree abstraction

The optimal abstract version of $\mathcal{P}[\![P]\!]$, $\mathcal{P}^\partial[\![P]\!] := \partial \circ \mathcal{P}[\![P]\!] \circ \partial^\gamma$, is

$$\mathcal{P}^\partial[\![P]\!]_{\mathcal{I}^\partial} = \lambda f(\overrightarrow{x_n}). \bigvee_{f(\overrightarrow{t_n}) \to r \ll P} \xi^\partial_{\{\overrightarrow{x_n}\}} [\![r]\!]^{\{\overrightarrow{x_n}/\overrightarrow{t_n}\}}_{\mathcal{I}^\partial} \tag{3.1.15}$$

where

$$\xi^\partial_V [\![r]\!]^\vartheta_{\mathcal{I}^\partial} := \begin{cases} zaproot(\vartheta \lfloor \mathcal{E}^\partial[\![r]\!]_{\mathcal{I}^\partial} \rfloor_V) & \text{if } r \text{ is } \mathcal{D}\text{-rooted} \\ \varepsilon \cdot \varrho \xrightarrow{\varrho} \vartheta \lfloor \mathcal{E}^\partial[\![r]\!]_{\mathcal{I}^\partial} \rfloor_V & \text{otherwise} \end{cases} \tag{3.1.16}$$

$$\vartheta \lfloor T \rfloor_V := \{ \vartheta \lfloor a_0 \rfloor_V \xrightarrow{\varrho_1} \ldots \xrightarrow{\varrho_n} \vartheta \lfloor a_n \rfloor_V \mid a_0 \xrightarrow{\varrho_1} \ldots \xrightarrow{\varrho_n} a_n \in T \} \tag{3.1.17}$$

$$\vartheta \lfloor \sigma \cdot \bar{s} \rfloor_V := (\vartheta\sigma) \restriction_V \cdot \bar{s} \tag{3.1.18}$$

and $zaproot(T)$ replaces the substitution of the root of tree T with ε.

The abstract evaluation $\mathcal{E}^\partial[\![e]\!]_{\mathcal{I}^\partial}$ of a term e w.r.t. an interpretation $\mathcal{I}^\partial$, is defined by induction on the size of $knots(e)$ as follows

$$\mathcal{E}^\partial[\![x]\!]_{\mathcal{I}^\partial} := \varepsilon \cdot x \tag{3.1.19a}$$

$$\mathcal{E}^\partial[\![\varphi(\overrightarrow{x_n})]\!]_{\mathcal{I}^\partial} := \mathcal{I}^\partial(\varphi(\overrightarrow{y_n}))[y_1/\varepsilon \cdot x_1]_{V_1} \ldots [y_n/\varepsilon \cdot x_n]_{V_n} \tag{3.1.19b}$$
$$\overrightarrow{y_n} \text{ fresh distinct, } V_i := \{x_1, \ldots, x_i, y_{i+1}, \ldots, y_n\}$$

$$\mathcal{E}^\partial[\![e]\!]_{\mathcal{I}^\partial} := \mathcal{E}^\partial[\![e[y]_p]\!]_{\mathcal{I}^\partial}[y/\, \mathcal{E}^\partial[\![e|_p]\!]_{\mathcal{I}^\partial}]_{var(e)} \tag{3.1.19c}$$
$$y \text{ fresh, leftmost } p \in baseknots(e)$$

provided that the abstract tree-embedding operation $G[x/B]_V$ over two evolving result trees is

$$G[x/B]_V := \{ d \mid d_g \in G, d_b \in B, d_g[x/d_b]_V \vdash d \} \tag{3.1.20}$$

where, for all abstract derivations $d_{\bar{g}}$ and $d_{\bar{b}}$ whose heads are $\vartheta \cdot \bar{g}$ and $\eta \cdot \bar{b}$, such that there exists a $\mathcal{V}_\varrho$-preserving $h = \vartheta \uparrow \eta\{x/\bar{b}\}$, $d_g[x/d_b]_V \vdash d$ is the least relation that satisfies the rules

$$\frac{}{d_{\bar{g}}[x/d_{\bar{b}}]_V \vdash \sigma_h \restriction_V \cdot h[\bar{g}]} \tag{3.1.21a}$$

$$\frac{d_{\bar{g}}[x/d_{\bar{b}'}]_V \vdash d}{d_{\bar{g}}[x/(\eta \cdot \bar{b} \xrightarrow{\varrho} d_{\bar{b}'})]_V \vdash \sigma_h \restriction_V \cdot h[\bar{g}] \xrightarrow{\varrho} d} \; \varrho \in var(h[\bar{g}]) \tag{3.1.21b}$$

$$\frac{(\vartheta \cdot \bar{g} \xrightarrow{\varrho} d_{\bar{g}'})[x/d_{\bar{b}'}]_V \vdash \theta \cdot \bar{s} \xrightarrow{\varrho} d}{(\vartheta \cdot \bar{g} \xrightarrow{\varrho} d_{\bar{g}'})[x/(\eta \cdot \bar{b} \xrightarrow{\varrho'} d_{\bar{b}'})]_V \vdash \sigma_h \restriction_V \cdot h[\bar{g}] \xrightarrow{\varrho} d} \; \begin{array}{l} \vartheta' \cdot \bar{g}' \text{ head of } d_{\bar{g}'}, \\ h' = \vartheta' \uparrow \eta\{x/\bar{b}\}, \\ h'[\varrho'] \text{ is } \mathcal{C}\text{-rooted} \end{array} \tag{3.1.21c}$$

$$\frac{d_{\bar{g}'}[x/d_{\bar{b}}]_V \vdash d}{(\vartheta \cdot \bar{g} \xrightarrow{\varrho} d_{\bar{g}'})[x/d_{\bar{b}}]_V \vdash \sigma_h \restriction_V \cdot h[\bar{g}] \xrightarrow{\varrho} d} \; \begin{array}{l} \theta \cdot \bar{s} \text{ head of } d,\ h' : h[\bar{g}] \to \bar{s}, \\ h'[\varrho] \notin \mathcal{V}_\varrho \end{array} \tag{3.1.21d}$$

$$\frac{d_{\bar{g}'}[x/d_{\bar{b}}]_V \vdash \theta \cdot \bar{s} \xrightarrow{\varrho'} d}{(\vartheta \cdot \bar{g} \xrightarrow{\varrho} d_{\bar{g}'})[x/d_{\bar{b}}]_V \vdash \sigma_h{\restriction_V} \cdot h[\bar{g}] \xrightarrow{\varrho} d} \; h' : h[\bar{g}] \to \bar{s},\ \varrho' = h'[\varrho] \tag{3.1.21e}$$

Rules 3.1.21 govern the inlaying of the steps of an abstract derivation $d_{\bar{b}}$ for a variable x into an abstract derivation $d_{\bar{g}}$. This is done by a sort of parameter passing, handled as the application of the homomorphism $h = \vartheta \uparrow \eta\{x/\bar{b}\}$ to the head of $d_{\bar{g}}$. The existence of h both constrains the heads of $d_{\bar{g}}$ and $d_{\bar{b}}$ to be compatible on their substitutions, and constrains the actual parameter value "coming" from $d_{\bar{b}}$ to be sufficiently evolved as the "parameter pattern" $x\vartheta$ demands. Note that we require h to be $\mathcal{V}_\varrho$-preserving, in order to prevent the inlaying of the nodes of successive steps, without before performing the step.

The inlay is done by taking the steps of $d_{\bar{g}}$ and $d_{\bar{b}}$ which are labelled with a bottom variable occurring in $h[\bar{g}]$, while ensuring at the same time that only compatible inlays are performed. In particular

- the axiom (3.1.21a) stops any further possible inlaying;
- the rule (3.1.21b) performs a step into a bottom variable ϱ appearing in $h[\bar{g}]$ by employing a step of $\eta \cdot \bar{b} \xrightarrow{\varrho} d_{\bar{b}'}$ (which is evolving the same ϱ);
- the rule (3.1.21c) considers the case when in $\vartheta \cdot \bar{g} \xrightarrow{\varrho} d_{\bar{g}'}$ the parameter $\bar{b}$ is sufficiently evolved to be inlaid into $\vartheta \cdot \bar{g}$ and then it scans the steps of $d_{\bar{b}}$ until we have a node which is sufficiently evolved to be inlaid into $d_{\bar{g}'}$;
- rules (3.1.21d) and (3.1.21e) both considers the case when the actual parameter $\bar{b}$ is sufficiently evolved to be inlaid directly into $d_{\bar{g}'}$. We scan the steps of the resulting inlaying until we actually change the bottom variable ϱ to a non-bottom variable term (to ensure to build a step which evolves ϱ).

It is worth noting that $\mathcal{P}^\partial$ results to be defined just in term of program rules, independently upon each specific choice of definitional tree. This is an expected outcome: information about op-rooted subterms disappeared so we observe only reduction to head normal form which does not depend upon the actual structure of definitional trees but just on its leaves (thus program rules).

It is also important to note that Equation (3.1.19c) of the abstract evaluation function could be defined with any possible choice of positions in $baseknots(t)$ (rather than the leftmost) and it can be proved that the result is independent upon the chosen order. This suggests that is also possible to give an alternative definition with a simultaneous parallel evaluation of all independent $knots(t)$. However, it would certainly need a more complex definition than the sequential version (3.1.19) (which is already not that simple), thus for the sake of simplicity we stick to Definition 3.1.19. The parallelism is nevertheless a very important property that is inherited in further abstractions and we will explicitly employ it (see, for instance, the Groundness abstraction example of Subsection 4.2.1).

Since $\mathcal{P}^\partial[\![P]\!]$ is monotonic by construction, we can define the (abstract) big-step fixpoint denotation as

$$\mathcal{F}^\partial[\![P]\!] := lfp(\mathcal{P}^\partial[\![P]\!]) \tag{3.1.22}$$

Example 3.1.10

The first iterate for the program of Example 2.2.13 is

$$\mathcal{P}^{\partial}[\![P_+]\!]\uparrow 1 = \left\{ x_1 + y_1 \mapsto \ \varepsilon \cdot \varrho_1 \begin{array}{l} \xrightarrow{\varrho_1} \{x_1/S(x_0)\} \cdot S(\varrho_0) \\ \xrightarrow{\varrho_1} \{x_1/Z\} \cdot y_1 \end{array} \right.$$

since

$$\begin{aligned}
&\mathcal{P}^{\partial}[\![P_+]\!]_{\perp_{\mathbb{ERT}}} = \\
&\quad [\text{by Equation (3.1.15), given } V := \{x_1, y_1\} \text{ and } \sigma := \{x_1/S(x_0)\}] \\
&= \left\{ x_1 + y_1 \mapsto \xi^{\partial}_V [\![y_1]\!]^{\{x_1/Z\}}_{\perp_{\mathbb{ERT}}} \curlyvee \xi^{\partial}_V [\![S(x_0 + y_1)]\!]^{\sigma}_{\perp_{\mathbb{ERT}}} \right. \\
&\quad [\text{by Equations (3.1.16) and (3.1.19c), given } V' := \{x_0, y_1\}] \\
&= \left\{ x_1 + y_1 \mapsto \begin{array}{l} \xi^{\partial}_V [\![y_1]\!]^{\{x_1/Z\}}_{\perp_{\mathbb{ERT}}} \curlyvee \\ \varepsilon \cdot \varrho_1 \xrightarrow{\varrho_1} \sigma \lfloor \mathcal{E}^{\partial}[\![S(z)]\!]_{\perp_{\mathbb{ERT}}} [z/\, \mathcal{E}^{\partial}[\![x_0 + y_1]\!]_{\perp_{\mathbb{ERT}}}]_{V'} \rfloor_V \end{array} \right. \\
&\quad [\perp_{\mathbb{ERT}}(S(z_0)) = \varepsilon \cdot S(z_0), \text{ by Equations (3.1.19b) and (3.1.23)}] \\
&= \left\{ x_1 + y_1 \mapsto \begin{array}{l} \xi^{\partial}_V [\![y_1]\!]^{\{x_1/Z\}}_{\perp_{\mathbb{ERT}}} \curlyvee \\ \varepsilon \cdot \varrho_1 \xrightarrow{\varrho_1} \sigma \lfloor \varepsilon \cdot S(z) [z/\, \mathcal{E}^{\partial}[\![x_0 + y_1]\!]_{\perp_{\mathbb{ERT}}}]_{V'} \rfloor_V \end{array} \right. \\
&\quad [\perp_{\mathbb{ERT}}(z_1 + z_2) = \varepsilon \cdot \varrho_0, \text{ by Equations (3.1.19b), (3.1.24) and (3.1.25)}] \\
&= \left\{ x_1 + y_1 \mapsto \xi^{\partial}_V [\![y_1]\!]^{\{x_1/Z\}}_{\perp_{\mathbb{ERT}}} \curlyvee \left(\varepsilon \cdot \varrho_1 \xrightarrow{\varrho_1} \sigma \lfloor \varepsilon \cdot S(z) [z/\varepsilon \cdot \varrho_0]_{V'} \rfloor_V \right) \right. \\
&\quad [\text{by Equations (3.1.17) and (3.1.26)}] \\
&= \left\{ x_1 + y_1 \mapsto \xi^{\partial}_V [\![y_1]\!]^{\{x_1/Z\}}_{\perp_{\mathbb{ERT}}} \curlyvee \left(\varepsilon \cdot \varrho_1 \xrightarrow{\varrho_1} \{x_1/S(x_0)\} \cdot S(\varrho_0) \right) \right. \\
&\quad [\text{by Equations (3.1.16), (3.1.17) and (3.1.19a)}] \\
&= \left\{ x_1 + y_1 \mapsto \left(\varepsilon \cdot \varrho_1 \xrightarrow{\varrho_1} \{x_1/Z\} \cdot y_1 \right) \curlyvee \left(\varepsilon \cdot \varrho_1 \xrightarrow{\varrho_1} \{x_1/S(x_0)\} \cdot S(\varrho_0) \right) \right. \\
&= \left\{ x_1 + y_1 \mapsto \ \varepsilon \cdot \varrho_1 \begin{array}{l} \xrightarrow{\varrho_1} \{x_1/S(x_0)\} \cdot S(\varrho_0) \\ \xrightarrow{\varrho_1} \{x_1/Z\} \cdot y_1 \end{array} \right.
\end{aligned}$$

$$\frac{}{\varepsilon \cdot S(z_0)[z_0/\varepsilon \cdot z]_{\{z\}} \vdash \varepsilon \cdot S(z)} \text{ (3.1.21a)} \tag{3.1.23}$$

$$\frac{}{\varepsilon \cdot \varrho_0[z_1/\varepsilon \cdot x_0]_{\{x_0, z_2\}} \vdash \varepsilon \cdot \varrho_0} \text{ (3.1.21a)} \tag{3.1.24}$$

$$\frac{}{\varepsilon \cdot \varrho_0[z_2/\varepsilon \cdot y_1]_{V'} \vdash \varepsilon \cdot \varrho_0} \text{ (3.1.21a)} \tag{3.1.25}$$

$$\frac{}{\varepsilon \cdot S(z)[z/\varepsilon \cdot \varrho_0]_{V'} \vdash \varepsilon \cdot S(\varrho_0)} \text{ (3.1.21a)} \tag{3.1.26}$$

The second iterate is

$$\mathcal{P}^{\partial}[\![P_+]\!]\uparrow 2 = \left\{ x_2 + y_2 \mapsto \ \varepsilon \cdot \varrho_2 \begin{array}{l} \xrightarrow{\varrho_2} \{x_2/Z\} \cdot y_2 \\ \searrow^{\varrho_2} \quad\quad \{x_2/S(S(x_0))\} \cdot S(S(\varrho_0)) \\ \quad \{x_2/S(x_1)\} \cdot S(\varrho_1) \begin{array}{l} \xrightarrow{\varrho_1} \\ \searrow_{\varrho_1} \end{array} \{x_2/S(Z)\} \cdot y_2 \end{array} \right.$$

The fixpoint $\mathcal{F}^{\partial}[\![P_+]\!](x+y)$ gives the same result shown in Example 3.1.3 (not a casualty because of Theorem 3.1.13).

Example 3.1.11

The first two iterates for the union of the programs of Examples 2.2.21 and 3.1.4 are

$$\mathcal{P}^{\partial}[\![P]\!]\uparrow 1 = \begin{cases} coin \mapsto\ \varepsilon \cdot \varrho_1 \xrightarrow{\varrho_1} \varepsilon \cdot Head \\ \qquad\qquad\quad\ \xrightarrow{\varrho_1} \varepsilon \cdot Tail \\ zeros \mapsto \varepsilon \cdot \varrho_1 \xrightarrow{\varrho_1} \varepsilon \cdot (Z : \varrho) \\ leq \triangleright x, y \mapsto\ \varepsilon \cdot \varrho_1 \xrightarrow{\varrho_1} \{x/Z, y/S(y_1)\} \cdot True \\ \qquad\qquad\quad\ \xrightarrow{\varrho_1} \{x/S(x_1), y/Z\} \cdot False \end{cases}$$

$$\mathcal{P}^{\partial}[\![P]\!]\uparrow 2 = \begin{cases} coin \mapsto \mathcal{P}^{\partial}[\![P]\!]\uparrow 1(coin) \\ zeros \mapsto \varepsilon \cdot \varrho_2 \xrightarrow{\varrho_2} \varepsilon \cdot (Z : \varrho_1) \xrightarrow{\varrho_1} \varepsilon \cdot (Z : Z : \varrho) \\ leq(x, y) \mapsto\ \varepsilon \cdot \varrho_2 \xrightarrow{\varrho_2} \{x/Z, y/S(y_1)\} \cdot True \\ \qquad\qquad\quad\ \xrightarrow{\varrho_2} \{x/S(Z), y/S(S(y_1))\} \cdot True \\ \qquad\qquad\quad\ \xrightarrow{\varrho_2} \{x/S(S(x_1)), y/S(Z)\} \cdot False \\ \qquad\qquad\quad\ \xrightarrow{\varrho_2} \{x/S(x_1), y/Z\} \cdot False \end{cases}$$

while the fixpoint on $leq(x, y)$ is the same of Example 3.1.4 and on *coin* and *zeros* is the same of Example 3.1.2 (which is not a casualty, because of Corollary 3.1.14).

Example 3.1.12

Let us consider an artificial example which manifests all the different aspects of $\mathcal{P}^{\partial}$ computation. Consider the program P

```
main x = val (h x)

val y = (y, isleaf y)

h x = N 1 (hson x) V

isleaf (N w V V) = True

hson 0 = V
hson 1 = U
```

Its evolving result tree semantics is obtained in three iterations.

$$\mathcal{P}^{\partial}[\![P]\!]\uparrow 1 = \begin{cases} val(x) \mapsto\ \varepsilon \cdot \varrho_1 \xrightarrow{\varrho_1} \varepsilon \cdot (x, \varrho_0) \\ isleaf(x) \mapsto \varepsilon \cdot \varrho_1 \xrightarrow{\varrho_1} \{x/N(v_1, V, V)\} \cdot True \\ h(x) \mapsto\ \varepsilon \cdot \varrho_1 \xrightarrow{\varrho_1} \varepsilon \cdot N(1, \varrho_0, V) \\ hson(x) \mapsto\ \varepsilon \cdot \varrho_1 \xrightarrow{\varrho_1} \{x/0\} \cdot V \\ \qquad\qquad\quad\ \xrightarrow{\varrho_1} \{x/1\} \cdot U \\ main(x) \mapsto\ \varepsilon \cdot \varrho_1 \end{cases}$$

$$\mathcal{P}^{\partial}[\![P]\!]\uparrow 2 = \begin{cases} val(x) \mapsto \varepsilon \cdot \varrho_2 \xrightarrow{\varrho_2} \varepsilon \cdot (x, \varrho_1) \xrightarrow{\varrho_1} \{x/N(v,V,V)\} \cdot (N(v,V,V), \mathit{True}) \\ isleaf\,(x) \mapsto \mathcal{P}^{\partial}[\![P]\!]\uparrow 1(isleaf\,(x)) \\ h(x) \mapsto \ \varepsilon \cdot \varrho_2 \xrightarrow{\varrho_2} \varepsilon \cdot N(1, \varrho_1, V) \begin{array}{l} \xrightarrow{\varrho_1} \{x/0\} \cdot N(1,V,V) \\ \xrightarrow[\varrho_1]{} \{x/1\} \cdot N(1,U,V) \end{array} \\ hson(x) \mapsto \mathcal{P}^{\partial}[\![P]\!]\uparrow 1(hson(x)) \\ main(x) \mapsto \ \varepsilon \cdot \varrho_2 \xrightarrow{\varrho_2} \varepsilon \cdot (\varrho_1, \varrho_0) \xrightarrow{\varrho_1} \varepsilon \cdot (N(1, \varrho_0', V), \varrho_0) \end{cases}$$

$$\mathcal{P}^{\partial}[\![P]\!]\uparrow 3 = \begin{cases} val(x) \mapsto \mathcal{P}^{\partial}[\![P]\!]\uparrow 2(val(x)) \\ isleaf\,(x) \mapsto \mathcal{P}^{\partial}[\![P]\!]\uparrow 1(isleaf\,(x)) \\ h(x) \mapsto \mathcal{P}^{\partial}[\![P]\!]\uparrow 2(h(x)) \\ hson(x) \mapsto \mathcal{P}^{\partial}[\![P]\!]\uparrow 1(hson(x)) \\ main(x) \mapsto \ \varepsilon \cdot \varrho_3 \overset{\varrho_3}{\leftarrow} \varepsilon \cdot (\varrho_1, \varrho_2) \overset{\varrho_1}{\leftarrow} \varepsilon \cdot (N(1, \varrho_0, V), \varrho_2) \begin{array}{l} \xrightarrow{\varrho_2} \{x/0\} \cdot (N(1,V,V), \mathit{True}) \\ \xrightarrow[\varrho_0]{} \{x/1\} \cdot (N(1,U,V), \varrho_2) \\ \xrightarrow[\varrho_1]{} \{x/0\} \cdot (N(1,V,V), \varrho_2) \end{array} \\ \qquad\qquad \overset{\varrho_2}{\searrow} \qquad\qquad\qquad\qquad \overset{\varrho_0 \downarrow}{} \\ \qquad\qquad \{x/0\} \cdot (N(1,V,V), \mathit{True}) \qquad\quad \{x/0\} \cdot (N(1,V,V), \mathit{True}) \end{cases}$$

and finally $\mathcal{P}^{\partial}[\![P]\!]\uparrow 4 = \mathcal{P}^{\partial}[\![P]\!]\uparrow 3 = \mathcal{F}^{\partial}[\![P]\!]$.

By looking at the denotation of $main(x)$ we can observe that, along a fixpoint computation, trees does not evolve one level at a time. Nested expressions may need several steps of the interpretation of the involved defined symbols altogether in order to trigger a contribution.

Another interesting aspect is due to sharing. We have the nested expression $val(h(x))$ with a duplicating rule for val, which thus involves the reduction of $h(x)$ as needed redex of $isleaf\,(h(x))$ and simultaneously reduces the first component of the tuple as a side effect. So, because of sharing, more than one bottom variable can evolve in the same abstract step.

Let us analyze this situation more in detail. Consider $(\mathcal{P}^{\partial}[\![P]\!]\uparrow 3)(main(x))$ which has three distinct evolving derivations leading to the same abstract computed result, namely $\{x/0\} \cdot (N(1,V,V), true)$. All of them have been inferred during the computation of $\mathcal{E}^{\partial}[\![val(h(x))]\!]_{\mathcal{P}^{\partial}[\![P]\!]\uparrow 2}$ as the embedding $d_g^1[z/d_b^1]_{\{x\}} \vdash d$ of the derivations

$$d_g^1 := \varepsilon \cdot \varrho_3 \xrightarrow{\varrho_3} \overbrace{\varepsilon \cdot (z, \varrho_4) \xrightarrow{\varrho_4} \underbrace{\{z/N(v,V,V)\} \cdot (N(v,V,V), true)}_{d_g^3}}^{d_g^2}$$

$$d_b^1 := \varepsilon \cdot \varrho_2 \xrightarrow{\varrho_2} \overbrace{\varepsilon \cdot N(1, \varrho_1, V) \xrightarrow{\varrho_1} \underbrace{\{x/0\} \cdot N(1,V,V)}_{d_b^3}}^{d_b^2}$$

where $d_g^1 \in \mathcal{E}^\partial[\![val(z)]\!]_{\mathcal{P}^\partial[\![P]\!]\uparrow 2}$ and $d_b^1 \in \mathcal{E}^\partial[\![h(x)]\!]_{\mathcal{P}^\partial[\![P]\!]\uparrow 2}$. The three proof trees are the following

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{d_g^3[z/d_b^3]_{\{x\}} \vdash \{x/0\}\cdot(N(1,V,V),true)}\,(3.1.21a)}{d_g^2[z/d_b^3]_{\{x\}} \vdash \underbrace{\{x/0\}\cdot(N(1,V,V),\varrho_4) \xrightarrow{\varrho_4} \{x/0\}\cdot(N(1,V,V),true)}_{d}}\,(3.1.21d)}{d_g^2[z/d_b^2]_{\{x\}} \vdash \varepsilon\cdot(N(1,\varrho_1,V),\varrho_4) \xrightarrow{\varrho_1} d}\,(3.1.21b)}{d_g^2[z/d_b^1]_{\{x\}} \vdash \varepsilon\cdot(\varrho_2,\varrho_4) \xrightarrow{\varrho_2} \varepsilon\cdot(N(1,\varrho_1,V),\varrho_4) \xrightarrow{\varrho_1} d}\,(3.1.21b)}{d_g^1[z/d_b^1]_{\{x\}} \vdash \varepsilon\cdot\varrho_3 \xrightarrow{\varrho_3} \varepsilon\cdot(\varrho_2,\varrho_4) \xrightarrow{\varrho_2} \varepsilon\cdot(N(1,\varrho_1,V),\varrho_4) \xrightarrow{\varrho_1} d}\,(3.1.21d) \tag{3.1.27}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{d_g^3[z/d_b^3]_{\{x\}} \vdash \{x/0\}\cdot(N(1,V,V),true)}\,(3.1.21a)}{d_g^2[z/d_b^3]_{\{x\}} \vdash \{x/0\}\cdot(N(1,V,V),\varrho_4) \xrightarrow{\varrho_4} \{x/0\}\cdot(N(1,V,V),true)}\,(3.1.21d)}{d_g^2[z/d_b^2]_{\{x\}} \vdash \underbrace{\varepsilon\cdot(N(1,\varrho_1,V),\varrho_4) \xrightarrow{\varrho_4} \{x/0\}\cdot(N(1,V,V),true)}_{d}}\,(3.1.21c)}{d_g^2[z/d_b^1]_{\{x\}} \vdash \varepsilon\cdot(\varrho_2,\varrho_4) \xrightarrow{\varrho_2} d}\,(3.1.21b)}{d_g^1[z/d_b^1]_{\{x\}} \vdash \varepsilon\cdot\varrho_3 \xrightarrow{\varrho_3} \varepsilon\cdot(\varrho_2,\varrho_4) \xrightarrow{\varrho_2} d}\,(3.1.21d) \tag{3.1.28}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{d_g^3[z/d_b^3]_{\{x\}} \vdash \{x/0\}\cdot(N(1,V,V),true)}\,(3.1.21a)}{d_g^2[z/d_b^3]_{\{x\}} \vdash \{x/0\}\cdot(N(1,V,V),\varrho_4) \xrightarrow{\varrho_4} \{x/0\}\cdot(N(1,V,V),true)}\,(3.1.21d)}{d_g^2[z/d_b^2]_{\{x\}} \vdash \varepsilon\cdot(N(1,\varrho_1,V),\varrho_4) \xrightarrow{\varrho_4} \{x/0\}\cdot(N(1,V,V),true)}\,(3.1.21c)}{d_g^2[z/d_b^1]_{\{x\}} \vdash \varepsilon\cdot(\varrho_2,\varrho_4) \xrightarrow{\varrho_4} \{x/0\}\cdot(N(1,V,V),true)}\,(3.1.21c)}{d_g^1[z/d_b^1]_{\{x\}} \vdash \varepsilon\cdot\varrho_3 \xrightarrow{\varrho_3} \varepsilon\cdot(\varrho_2,\varrho_4) \xrightarrow{\varrho_4} \{x/0\}\cdot(N(1,V,V),true)}\,(3.1.21d) \tag{3.1.29}$$

Note that in (3.1.28) when ϱ_4 evolves also ϱ_1 does, while in (3.1.29) when ϱ_4 evolves also ϱ_2 does.

Properties of the evolving result tree semantics

The abstract evaluation operator $\mathcal{E}^\partial$ is precise.

Theorem 3.1.13 *For all $e \in \mathcal{T}(\Sigma,\mathcal{V})$ and all $\mathcal{I} \in \mathbb{I}$, $\partial_{var(e)}(\mathcal{E}[\![e]\!]_{\mathcal{I}}) = \mathcal{E}^\partial[\![e]\!]_{\partial(\mathcal{I})}$.*

With this result we can prove that $\mathcal{P}^\partial[\![P]\!]$ is precise as well.

Corollary 3.1.14 *For all $P \in \mathbb{P}_\Sigma$,*

1. $\mathcal{P}^\partial[\![P]\!] \circ \partial = \partial \circ \mathcal{P}[\![P]\!]$,
2. $\mathcal{P}^\partial[\![P]\!]$ *is continuous,*
3. $\mathcal{F}^\partial[\![P]\!] = \mathcal{P}^\partial[\![P]\!]\uparrow\omega$,
4. $\mathcal{F}^\partial[\![P]\!] = \partial(\mathcal{F}[\![P]\!])$.

Corollary 3.1.15 (correctness of $\mathcal{F}^{\partial}$ w.r.t. $\approx_{cr}$) *For all $P_1, P_2 \in \mathbb{P}_\Sigma$, $\mathcal{F}^{\partial}[\![P_1]\!] = \mathcal{F}^{\partial}[\![P_2]\!]$ implies $P_1 \approx_{cr} P_2$.*

The converse implication does not hold, as shown by the following example.

Example 3.1.16

Consider the programs P_1 and P_2

```
g x = A (h x)
h B = C
```

```
g B = A C
h B = C
```

Although $P_1 \approx_{cr} P_2$, $\mathcal{F}^{\partial}[\![P_1]\!] \neq \mathcal{F}^{\partial}[\![P_2]\!]$ since $\mathcal{F}^{\partial}[\![P_1]\!](g(x)) = \varepsilon \cdot A(\varrho) \xrightarrow{\varrho} \{x/B\} \cdot A(C)$ while $\mathcal{F}^{\partial}[\![P_2]\!](g(x)) = \varepsilon \cdot \varrho \xrightarrow{\varrho} \{x/B\} \cdot A(C)$.

Hence our semantics is not fully abstract w.r.t. $\approx_{cr}$. So, apparently, we failed our goal. However, is it reasonable to refer to the computed result program equivalence? While in the case of logic languages or eager functional languages it is perfectly reasonable, in the setting of lazy languages this is no longer the case. For instance, in the lazy setting infinite values are allowed, but $\approx_{cr}$ does not consider infinite values by definition. Thus a program defining an infinite value is equivalent to a program that computes nothing at all. However if we add a function that extracts a finite component then the two programs can be distinguished even with finite computed results. The issue, in general, is connected to the fact that equivalent expressions no longer need to be so when embedded in some context. For example, if we add the rule `f (A x) D = D` to both the programs of Example 3.1.16, the functions g become distinguishable, since the call `f (g x) x` exhibits different computed results in the two programs, namely $\{x/D\} \cdot D$ in P_1 and no computed results in P_2.

In general, because of laziness, we can have two functions that can be *called* (with any arbitrary non-value arguments) and give the same computed results, but can produce different computed results when *used* as arguments of another function call. Thus (as also observed by [63] and others) the program equivalence which *has reasonably to be considered* need to be a congruence w.r.t. "usage", i.e., for each "independent" context into which we embed the two behaviour equivalent expressions we still have the same behaviors. The rationale behind "program usage" is that we know the (same) API of two programs and we are able to *use* them as we please (make calls and observe their outputs), but we cannot tamper in any way with their code (by redefining some functions or by directly accessing to its internal structure). Let us define this formally.

Definition 3.1.17 (Congruence w.r.t. usage) *Let $\Sigma = \mathcal{D} \cup \mathcal{C}$ and $\Sigma' = \mathcal{D}' \cup \mathcal{C}'$ be two signatures.*

We define the set of using programs $\mathbb{UP}_\Sigma^{\Sigma'} := \{Q \mid \forall f(\overrightarrow{t_n}) \to r \in Q,\ f/n \in \mathcal{D}',\ \overrightarrow{t_n} \in \mathcal{T}(\mathcal{C} \cup \mathcal{C}', \mathcal{V}),\ r \in \mathcal{T}(\Sigma \cup \Sigma', \mathcal{V})\}$ *if Σ and Σ' are disjoint. Otherwise we consider it empty.*

We say that a program equivalence $\approx$ is a congruence w.r.t. usage *when, for all P_1, $P_2 \in \mathbb{P}_\Sigma$,*

$$P_1 \approx P_2 \iff \forall \Sigma'.\, \forall Q \in \mathbb{UP}_\Sigma^{\Sigma'}.\, Q \cup P_1 \approx Q \cup P_2 \tag{3.1.30}$$

Given any P_1, $P_2 \in \mathbb{P}_\Sigma$ *we define the* usage computed result program equivalence $\approx_{cr}^{us}$ *as*

$$P_1 \approx_{cr}^{us} P_2 :\Longleftrightarrow \forall \Sigma'. \forall Q \in \mathbb{UP}_\Sigma^{\Sigma'}. Q \cup P_1 \approx_{cr} Q \cup P_2 \tag{3.1.31}$$

By looking in detail, we can note that for a using program there is no technical restriction on the usage of constructors in patterns and no restriction at all on the RHSs. This is in contrast with the just stated rationale. First new constructors cannot be mixed with old ones without changing the datatype declarations of the original program. Similarly original functions cannot have as argument new functions which return new datatypes. However the expressive power of these two notions is identical because any instance of the apparently stronger notion can be converted to the other. Indeed, if we have a "bad" using program P, i.e., P admissible w.r.t. Definition 3.1.17 but violating the stated rationale, we can transform P by including a clone of $\mathcal{C}$ into $\mathcal{C}'$ and then add in-place conversion functions that convert old values into clones in all places of P that violates the type constraints. This conversion preserves computed results. Hence we can safely use Definition 3.1.17 which is technically easier.

The previous discussion on programs P_1, P_2 of Example 3.1.16 shows that $\approx_{cr}$ is not a congruence w.r.t. usage[3].

Clearly $\approx_{cr}^{us}$ is a refinement of $\approx_{cr}$ (since $\emptyset \in \mathbb{UP}_\Sigma^{\Sigma'}$) and, by definition, it is a congruence w.r.t. usage (since, $\forall Q \in \mathbb{UP}_\Sigma^{\Sigma'}$, $P_1 \approx_{cr}^{us} P_2$ implies $Q \cup P_1 \approx_{cr}^{us} Q \cup P_2$). Actually it is the minimal (coarser) congruence w.r.t. usage amongst all refinements of $\approx_{cr}$. Thus $\approx_{cr}^{us}$ is *the* reasonable program equivalence for the lazy functional logic paradigm.

Note that there is an isomorphism between our notion congruence w.r.t. usage program equivalence and the constructor form observable expression equivalence w.r.t. every possible context which is used by [63] as reference behaviour for its semantics full abstraction.

- First note that $P_e \in \mathbb{P}_{\Sigma\cup\Sigma'}$ safely extends $P \in \mathbb{P}_\Sigma$ if and only if $P_e \setminus P \in \mathbb{UP}_\Sigma^{\Sigma'}$. We technically need the extending rules separately because we have to add the same extension simultaneously to two programs.

- The constructor form observable equivalence of expressions e_1, e_2 w.r.t. program P of [63], with our notation can be formalized as: for all $Q \in \mathbb{UP}_\Sigma^{\Sigma'}$,

  $$\mathcal{B}^{cr}[\![Q \cup P]\!](C[e_1]) = \mathcal{B}^{cr}[\![Q \cup P]\!](C[e_2]), \quad \text{for all context } C \tag{3.1.32}$$

  By taking an extra function symbol f and $var(e_1) \cup var(e_2) = \{\overrightarrow{x_n}\}$ we can define $P_i := P \cup \{f(\overrightarrow{x_n}) \to e_i\}$. It is straightforward to prove that, for all $Q \in \mathbb{UP}_\Sigma^{\Sigma'}$, (3.1.32) is equivalent to

  $$\forall e \in \mathcal{T}(\Sigma \cup \Sigma', \mathcal{V}).\ \mathcal{B}^{cr}[\![Q \cup P_1]\!](e) = \mathcal{B}^{cr}[\![Q \cup P_2]\!](e) \tag{3.1.33}$$

  which actually is $P_1 \approx_{cr}^{us} P_2$.

  Vice versa, given $P_1, P_2 \in \mathbb{P}_\Sigma$, we can clone signature Σ into $\hat{\Sigma}$ and also clone accordingly P_2 into $\hat{P}_2$. For all $Q \in \mathbb{UP}_\Sigma^{\Sigma'}$ with Σ' disjoint with $\hat{\Sigma}$, $Q \cup P_1 \approx_{cr} Q \cup P_2$ if and only if, for all context C,

  $$\forall e \in \mathcal{T}(\Sigma \cup \Sigma', \mathcal{V}).\ \mathcal{B}^{cr}[\![Q \cup P_1]\!](C[e]) = \mathcal{B}^{cr}[\![Q \cup P_2]\!](C[e])$$

[3] take $Q := \{f(A(x), D) \to D\} \in \mathbb{UP}_\Sigma^{\{f\}}$ and $Q \cup P_1 \not\approx_{cr} Q \cup P_2$.

if and only if (given $\hat{e}$ the cloning of e)

$$\forall e \in \mathcal{T}(\Sigma \cup \Sigma', \mathcal{V}).\ \mathcal{B}^{cr}[\![Q \cup P_1]\!](C[e]) = \mathcal{B}^{cr}[\![Q \cup \hat{P}_2]\!](C[\hat{e}])$$

if and only if (since $\hat{\Sigma}$, $\Sigma'$and Σ are disjoint)

$$\forall e \in \mathcal{T}(\Sigma \cup \Sigma', \mathcal{V}).\ \mathcal{B}^{cr}[\![Q \cup P_1 \cup \hat{P}_2]\!](C[e]) = \mathcal{B}^{cr}[\![Q \cup P_1 \cup \hat{P}_2]\!](C[\hat{e}])$$

and thus, for program $P := P_1 \cup \hat{P}_2$, $e_1 := e$ and $e_2 := \hat{e}$, we get Equation (3.1.32).

We preferred the formulation of Definition 3.1.17 because with it is easier, for the reasoning at the level of program's rule definitions, to establish the full abstraction of program's denotations.

Actually our semantics $\mathcal{F}^{\partial}$ is correct w.r.t. $\approx^{us}_{cr}$ as stated formally by the following result.

Theorem 3.1.18 (correctness of $\mathcal{F}^{\partial}$ w.r.t. $\approx^{us}_{cr}$) *For all $P_1, P_2 \in \mathbb{P}_\Sigma$, $\mathcal{F}^{\partial}[\![P_1]\!] = \mathcal{F}^{\partial}[\![P_2]\!]$ implies $P_1 \approx^{us}_{cr} P_2$.*

However the converse implication does not hold either, as shown by the following example.

Example 3.1.19

Consider the programs P_1 and P_2

```
f x = A x                    f x = id (A (id x))
```

Although $P_1 \approx^{us}_{cr} P_2$, $\mathcal{F}^{\partial}[\![P_1]\!] \neq \mathcal{F}^{\partial}[\![P_2]\!]$ since $\mathcal{F}^{\partial}[\![P_1]\!](f(x)) = \varepsilon \cdot A(x)$ while $\mathcal{F}^{\partial}[\![P_2]\!](f(x)) = \varepsilon \cdot \varrho_1 \xrightarrow{\varrho_1} \varepsilon \cdot A(\varrho_2) \xrightarrow{\varrho_2} \varepsilon \cdot A(x)$.

Hence our semantics is not fully abstract even w.r.t. (the stronger) $\approx^{us}_{cr}$. So we still missed our goal. However we are indeed quite near. In Subsection 3.1.3 we will show a semantics, obtained with a further (slight) abstraction of $\mathcal{F}^{\partial}$, which is indeed fully abstract w.r.t. $\approx^{us}_{cr}$.

Relating evolving result trees to computed results

We can easily define an abstraction which collects the computed results of a small-step tree:

$$\chi(\mathcal{I}) := \lambda\pi.\left\{(\sigma_1 \cdots \sigma_n)\restriction_\pi \cdot v \;\middle|\; \pi \xRightarrow[p_1]{\sigma_1} \cdots \xRightarrow[p_n]{\sigma_n} v \in \mathcal{I}(\pi),\ v \in \mathcal{T}(\mathcal{C}, \mathcal{V})\right\} \tag{3.1.34}$$

Actually, as can be expected, it turns out that χ can be expressed as a further abstraction of ∂. Namely, $\chi = \beta \circ \partial$ where, given $\mathcal{I}^{\partial} \in \mathbb{I}_{\mathbb{ERT}}$,

$$\beta(\mathcal{I}^{\partial}) := \lambda\pi.\left\{\sigma \cdot v \;\middle|\; \sigma \cdot v \in \mathcal{I}^{\partial}(\pi),\ v \in \mathcal{T}(\mathcal{C}, \mathcal{V})\right\} \tag{3.1.35}$$

Let β^γ be β adjoint (and $\chi^\gamma := \partial^\gamma \circ \beta^\gamma$). The optimal abstraction $\beta \circ \mathcal{P}^{\partial}[\![P]\!] \circ \beta^\gamma = \chi \circ \mathcal{P}[\![P]\!] \circ \chi^\gamma$ turns out to be *not* precise. For example consider

$$\mathcal{I}^{\partial}_1 := \begin{cases} g(x) \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} \{x/B\} \cdot A(C) \\ f(x,y) \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} \{x/A(x'), y/D\} \cdot D \end{cases}$$

$$\mathcal{I}_2^\partial := \begin{cases} g(x) \mapsto & \varepsilon \cdot \varrho \xrightarrow{\varrho} \{x/B\} \cdot A(C) \\ & \varepsilon \cdot A(\varrho) \xrightarrow{\varrho} \{x/B\} \cdot A(C) \\ f(x,y) \mapsto & \varepsilon \cdot \varrho \xrightarrow{\varrho} \{x/A(x'), y/D\} \cdot D \end{cases} \qquad \sqsubseteq \beta^\gamma(\beta(\mathcal{I}_1^\partial))$$

For program $P := \{k(x) \to f(g(x), x)\}$ we have $\beta(\mathcal{P}^\partial[\![P]\!]_{\mathcal{I}_1^\partial}) = \{k(x) \mapsto \emptyset$ while $\beta(\mathcal{P}^\partial[\![P]\!]_{\mathcal{I}_2^\partial}) = \{k(x) \mapsto \{\varepsilon \cdot D\}$. Hence, by $\mathcal{P}^\partial$ monotonicity, $\mathcal{P}^\partial[\![P]\!]_{\mathcal{I}_1^\partial} \neq \mathcal{P}^\partial[\![P]\!]_{\beta^\gamma(\beta(\mathcal{I}_1^\partial))}$.

This is clearly an expected result as all the work of this section originated by the observation that the collection of computed results cannot give a correct semantics, which means that the corresponding optimal abstract immediate consequences operator has to be non precise (as we have just formally proved).

Note that having a correct abstraction, which means that we have a conservative approximation of concrete solutions, does not mean to have an abstract semantics which is correct w.r.t. a certain program equivalence. The approximation can indeed add extra solutions that render two program indistinguishable, while they should not be such.

3.1.3 Full abstraction w.r.t. $\approx_{cr}^{us}$

The two programs of Example 3.1.19 cannot be distinguished by $\approx_{cr}^{us}$, even if one program has different (longer) derivations. One of the two programs computes the solution "atomically", while the other does it in two more steps but without altering the computed answer in the extra steps. In general, small-step behaviors with extra steps that do not alter the computed answer are indistinguishable by $\approx_{cr}^{us}$. Counterintuitively, we can note that if we would have chosen to introduce constructors and variables one at a time, then the semantics of the two programs would have been equivalent. This is, besides, exactly what happens in the CRWL approach [49, 50, 64], where partial results "evolve" one constructor at a time, even if the small-step semantics would introduce them in coarser combinations.

Thus we can think of making a further abstraction of ∂ which shucks the introduction of constructors (introducing, consequentially, more edges in the denotation). However this naïve definition is not enough as the actual situation is quite more complex. So, to obtain the desired abstraction, we proceed with a typical abstract interpretation construction. To obtain an abstraction whose induced (abstract) semantics is fully abstract w.r.t. $\approx_{cr}^{us}$, we need to develop an abstraction over $\mathbb{I}_{\mathbb{ERT}}$ which maps to the same abstract element all denotations of $\approx_{cr}^{us}$-equivalent programs. So we start by defining an upper closure operator Ξ which maps any denotation $\mathcal{I}^\partial$ into the biggest denotation $\mathcal{I}_m^\partial$ amongst all the denotations of $\approx_{cr}^{us}$-equivalent programs. Such $\mathcal{I}_m^\partial = \Xi(\mathcal{I}^\partial)$ is called the closure of $\mathcal{I}^\partial$ w.r.t. Ξ and in the rest of the discussion we just call it "the closure" for brevity.

Having identified in this way the image of the concretization function within $\mathbb{I}_{\mathbb{ERT}}$ ("the semantics" of the new domain in terms of $\mathbb{I}_{\mathbb{ERT}}$ elements), we proceed by finding a suitable representation for the new domain, together with the searched abstraction.

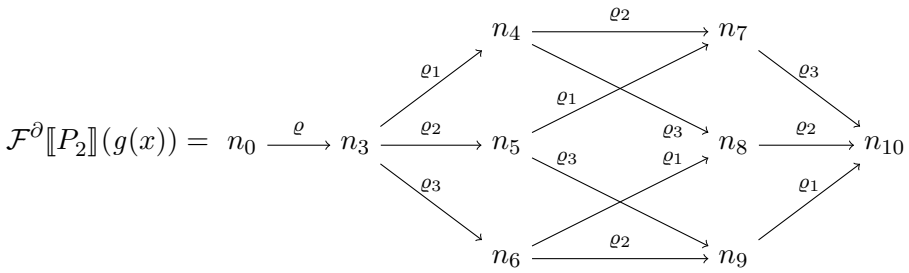
Let us explain which is the characterization of the closures by reasoning on a concrete example.

Example 3.1.20

Consider the program $P_1 := \{g(x) \to C(x, n : f(x), n),\ f(A) \to B\}$ which has the $\mathbb{I}_{\mathbb{ERT}}$ denotation

$$\mathcal{F}^\partial[\![P_1]\!](g(x)) = n_0 : \varepsilon \cdot \varrho \xrightarrow{\varrho} n_1 : \varepsilon \cdot C(x, \varrho_2, \varrho_2) \xrightarrow{\varrho_2} n_2 : \{x/A\} \cdot C(A, n\colon B, n)$$

Now consider the program $P_2 := \{g(x) \to id(C(id(x), id(f(x)), id(f(x)))),\ f(A) \to id(B)\}$ which is $\approx^{us}_{cr}$-equivalent to P_1. Its $\mathbb{I}_{\mathbb{ERT}}$ denotation is[4]



where, for $\vartheta := \{x/A\}$,

$$n_3 = \varepsilon \cdot C(\varrho_1, \varrho_2, \varrho_3) \qquad n_4 = \varepsilon \cdot C(x, \varrho_2, \varrho_3) \qquad n_5 = \vartheta \cdot C(\varrho_1, B, \varrho_3)$$
$$n_6 = \vartheta \cdot C(\varrho_1, \varrho_2, B) \qquad n_7 = \vartheta \cdot C(A, B, \varrho_3) \qquad n_8 = \vartheta \cdot C(A, \varrho_2, B)$$
$$n_9 = \vartheta \cdot C(\varrho_1, B, B) \qquad n_{10} = \vartheta \cdot C(A, B, B)$$

In the closure of $\mathcal{F}^{\partial}[\![P_1]\!]$ we have:

- All the nodes which can be obtained by replacing, in all possible combinations, into (the value of) $\varepsilon \cdot C(x, \varrho_2, \varrho_2)$ a bottom variable in each node. Namely n_0, n_1, n_3 and n_4.
- Moreover, all the nodes which can be obtained by replacing, in all possible combinations, into (the value of) $\{x/A\} \cdot C(A, n\colon B, n)$ a bottom variable in each node, *unless* we obtain one of the former values. Namely n_2 and n_5–n_{10}, but not (for instance) $\{x/A\} \cdot C(\varrho_1, \varrho_2, \varrho_3)$ because we already have n_3.

Moreover all these nodes are connected with an edge if one can be an evolution of another (an instance of substitution and value that changes at least a bottom variable).

In general

Property 3.1.21 *In a closure we have all possible introductions of one constructor (or variable) at a time between any two nodes which are connected in the original tree. More precisely we have all terms which can be obtained by removing the constructors from each original node n (by replacing, in all possible combinations, a bottom variable in each subterm), but without the elements which can also be generated by an ancestor of n.*

Moreover all these nodes are connected with an edge if one can be an evolution of another.

The latter fact implies that the tree structure is actually irrelevant, because the presence of an edge is determined solely by the content of nodes. Thus we can use just the set of the nodes of a closure to represent a closure.

Furthermore, something really interesting happens about shared subterms. If we have a path involving shared bottom variables (like n_0, n_1, n_2) then we have all its non-shared

[4] For the sake of conciseness we share common nodes in the following pictures. The real evolving result trees can be obtained by unraveling the depicted DAGs.

variants $(n_0, n_3, \ldots, n_{10})$. Thus only terms which share only data variables are relevant. But then we can forget also the variable sharing and use in the new denotation just traditional terms instead of term graphs.

In the following we denote by $\mathbb{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho)$ the term graphs of $\mathcal{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho)$ which are actually trees. We will call ψ-term any term in $\mathbb{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho)$. To model Property 3.1.21 we will use ψ-terms and order them with the *approximation ordering* $\lesssim$, defined as the least partial ordering satisfying

1. $\varrho \lesssim t$ for all $t \in \mathbb{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho)$ and $\varrho \in \mathcal{V}_\varrho$,
2. $x \lesssim x$ for all $x \in \mathcal{V}$ and
3. $\bigwedge_{i=1}^n t_i \lesssim s_i \Rightarrow c(t_1, \ldots, t_n) \lesssim c(s_1, \ldots, s_n)$ for all $c_{/n} \in \mathcal{C}$.

This order is essentially the approximation ordering of [63] using variables in $\mathcal{V}_\varrho$ instead of $\bot$.

In case we have data variables in the value of nodes of a denotation there is also another very impacting phenomenon.

Example 3.1.22

Consider again the program P_1 of Example 3.1.20. Its extension $P_3 := P_1 \cup \{g(C(x_1, x_2, x_3)) \to C(C(undf, x_2, x_3), undf, undf)\} \approx^{us}_{cr} P_1$ has

$$\mathcal{F}^\partial[\![P_3]\!](g(x)) = \; n_0 \begin{array}{l} \overset{\varrho}{\nearrow} n_{11} : \{x/C(x_1, x_2, x_3)\} \cdot C(C(\varrho_4, x_2, x_3), \varrho_2, \varrho_3) \\ \underset{\varrho}{\searrow} n_1 \xrightarrow{\varrho_2} n_2 \end{array}$$

We can add another rule

$$g(C(x_1, C(x_2, x_3, x_4), x_5)) \to C(C(x_1, C(x_2, undf, undf), undf), undf, undf)$$

and obtain another $\approx^{us}_{cr}$-equivalent program. In the closure of $\mathcal{F}^\partial[\![P_1]\!]$ this corresponds to instantiate the variable x of node n_4 with $C(undf, x_2, x_3)$ and $C(x_1, C(x_2, undf, undf)$ in its partial value, and, at the same time, composing $\{x/C(x_1, x_2, x_3)\}$ and $\{x/C(x_1, C(x_2, x_3, x_4), x_5)\}$ in its substitution. The resulting nodes are n_{11} and

$$n_{12} = \{C(x_1, C(x_2, x_3, x_4), x_5)\} \cdot C(C(x_1, C(x_2, \varrho_4, \varrho_5), \varrho_6), \varrho_2, \varrho_3).$$

Moreover all approximations between these nodes are also in the closure of $\mathcal{F}^\partial[\![P_1]\!]$. In the closure of $\mathcal{F}^\partial[\![P_1]\!]$ we also have

- all unraveling of the mentioned nodes;
- other instances of variable x, like $C(E(\varrho_7), E(\varrho_8), x_3)$, $E(E(E(\varrho_9)))$, *etc.*;

Furthermore, as before, we have all approximations between all these nodes.

The instances of variables which cannot appear are those that are non-linear or contain a value subterm which is not a variable. Thus, in general, if we have in a closure a partial computed result with data variables, then we also have *all* its instances with *any* linear ψ-term t such that all its data subterms are just variables. Namely the terms of

$$ui\mathbb{T}(\mathcal{C}, \mathcal{V}, \mathcal{V}_\varrho) := \{t \in \mathbb{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho) \setminus \mathcal{V}_\varrho \mid t \text{ linear}, t|_p \in \mathbb{T}(\mathcal{C}, \mathcal{V}) \Leftrightarrow t|_p \in \mathcal{V}\}.$$

We call these terms *usage-invisible*, since they cannot alter the behaviour of usage-equivalent programs. To each usage-invisible term t we associate a term $\lceil t \rfloor$ which is the result of the replacement in t of each bottom variable with a fresh data variable. Note that $\lceil t \rfloor$ is linear by construction.

In the following $uiSubsts_V$ denotes all the linear substitutions from V to usage-invisible terms and, by an abuse of notation, $uiSubsts_{var(t)\cap\mathcal{V}}$ will be denoted by $uiSubsts_t$ for any given term t. Again, to each $\vartheta \in uiSubsts_V$ we associate a linear substitution $\lceil\vartheta\rfloor$ s.t. $dom(\lceil\vartheta\rfloor) = dom(\vartheta)$ and $x\lceil\vartheta\rfloor = \lceil x\vartheta\rfloor$ for $x \in dom(\lceil\vartheta\rfloor)$.

It is very important to note that, in presence of data variables in (the value of) nodes, even if we start with a finite tree the closure will be infinite.

To sum up, to obtain a closure we need to unravel all terms, instantiate data variables with usage-invisible terms and then add all approximations between these nodes, in the sense of Property 3.1.21. So, to define the abstraction of a big-step sequence in $\mathbb{ERT}_{\overrightarrow{x_n}}$, we need first a function $\psi(\bar{t})$ to unravel a τ-term $\bar{t}$ and replace every occurrence of a bottom variable with a fresh one. Formally, $\psi : \mathcal{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho) \to \mathbb{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho)$ [5]

$$\psi(x) := x \qquad \psi(\varrho) := \varrho' \text{ fresh} \qquad \psi(c(t_1, \ldots, t_n)) := c(\psi(t_1), \ldots, \psi(t_n)) \tag{3.1.36}$$

Now we define the abstraction of a big-step sequence $\bar{d} \in \mathbb{ERT}_{\overrightarrow{x_n}}$ by structural induction as

$$\tilde{\zeta}_{\{\overrightarrow{x_n}\}}(\sigma \cdot \bar{s}) := \langle \sigma \cdot \bar{s} \rangle_{\{\overrightarrow{x_n}\}} \tag{3.1.37a}$$

$$\tilde{\zeta}_{\{\overrightarrow{x_n}\}}(\sigma \cdot \bar{s} \xrightarrow{\varrho} \bar{d}') := \langle \sigma \cdot \bar{s} \rangle_{\{\overrightarrow{x_n}\}} \cup \left\{ \theta \cdot t \in \tilde{\zeta}_{\{\overrightarrow{x_n}\}}(\bar{d}') \,\middle|\, t|_p \notin \mathcal{V}_\varrho,\ \bar{s}|_p = \varrho \right\} \tag{3.1.37b}$$

where $\langle \sigma \cdot \bar{s} \rangle_V$ is the down-closure (w.r.t. $\lesssim$) of all the *usage-invisible instances* of $\sigma \cdot \bar{s}$ defined by

$$\langle \sigma \cdot \bar{s} \rangle_V := \left\{ (\sigma\lceil\vartheta\rfloor)\restriction_V \cdot s \;\middle|\; \begin{array}{l} t \lesssim \psi(\bar{s}),\ \vartheta \in uiSubsts_t \\ s' \in base(t, \vartheta),\ s' \lesssim s \lesssim \psi(t\vartheta) \end{array} \right\} \tag{3.1.38}$$

and $base(t, \theta)$ is the set of all the ψ-terms obtained by replacing in t every occurrence of a variable $x \in dom(\theta)$ except one, with the least approximation of $x\theta$ not belonging to $\mathcal{V}_\varrho$, then replacing the remaining occurrence with $x\theta$. For instance

$$base(C(y, y, z, w, A(z, \varrho_1)), \{y/A(v, v), z/B\}) =$$
$$\{C(A(v, v), A(\varrho_2, \varrho_3), B, w, A(B, \varrho_1)),\ C(A(\varrho_2, \varrho_3), A(v, v), B, w, A(B, \varrho_1))\}$$

$\tilde{\zeta}$ yields all approximations according to Property 3.1.21. This is done in this way:

- Equation (3.1.37a) just collects all the approximations of all the usage-equivalent instances of the head,
- Equation (3.1.37b) both collects every approximation for every usage-equivalent instance of the head, and the elements recursively collected from the tail which actually represents an evolvement of one occurrence of the bottom variable that labels the step. This prevents from collecting elements which are more approximated than any ancestor encountered in the same path.

[5] Note that in $\psi(\bar{t})$ nothing is shared.

Definition 3.1.37 is extended to evolving result tree in $\mathbb{ERT}_{\overrightarrow{x_n}}$ as

$$\tilde{\zeta}_{\{\overrightarrow{x_n}\}}(T) := \bigcup_{\bar{d}\in T} \tilde{\zeta}_{\{\overrightarrow{x_n}\}}(\bar{d}) \tag{3.1.39}$$

The sets built by $\tilde{\zeta}$ contains all possible approximations which are coherent with program behaviour, thus we will call them *approximated result sets*.

We can define the new abstraction by composition as

$$\tilde{\nu}_V := \tilde{\zeta}_V \circ \partial_V \qquad \tilde{\zeta}(\mathcal{I}^\partial) := \lambda\pi.\ \tilde{\zeta}_{var(\pi)}(\mathcal{I}^\partial(\pi)) \qquad \tilde{\nu} := \tilde{\zeta} \circ \partial$$

and, analogously to what done for the evolving result tree case, derive systematically the domain (both its support and order), as well as the corresponding Galois insertion, from the abstraction $\tilde{\nu}$ (or $\tilde{\zeta}$).

First we define the (support of the) domain of approximated result set semantics as $\mathbb{I}_{\mathbb{ARS}} := \tilde{\zeta}(\mathbb{I}_{\mathbb{ERT}}) = \tilde{\nu}(\mathbb{I}) = [\mathbb{MGC} \to \mathbb{ARS}]$ ($\mathbb{ARS} \subset \wp(\mathcal{C}Substs \times \mathbb{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho))$). Moreover let $\mathbb{ARS}_{\overrightarrow{x_n}} := \tilde{\zeta}_{\{\overrightarrow{x_n}\}}(\mathbb{ERT}_{\overrightarrow{x_n}})$.

Note that, by construction, any $S \in \mathbb{ARS}$ contains $\varepsilon \cdot \varrho$ (for some $\varrho \in \mathcal{V}_\varrho$).

The order induced by $\tilde{\zeta}_{\{\overrightarrow{x_n}\}}$ is set inclusion on $\mathbb{ARS}_{\overrightarrow{x_n}}$ and the pointwise lifting of set inclusion on $\mathbb{I}_{\mathbb{ARS}}$. Hence $\mathbb{I}_{\mathbb{ARS}}$ and $\mathbb{ARS}_{\overrightarrow{x_n}}$ are (trivially) complete lattices.

Example 3.1.23

Consider the program P_1 of Example 3.1.20. The $\tilde{\zeta}_{\{x\}}$ abstraction of $\mathcal{F}^\partial[\![P_1]\!](g(x))$ is

$$\left\{\begin{array}{l}\varepsilon\cdot\varrho_0,\ \varepsilon\cdot C(\varrho_1,\varrho_2,\varrho_3),\ \varepsilon\cdot C(x,\varrho_2,\varrho_3),\ \{x/A\}\cdot C(\varrho_1,B,\varrho_3),\\ \{x/A\}\cdot C(\varrho_1,\varrho_2,B),\ \{x/A\}\cdot C(A,B,\varrho_3),\ \{x/A\}\cdot C(A,\varrho_2,B),\\ \{x/A\}\cdot C(\varrho_1,B,B),\ \{x/A\}\cdot C(A,B,B)\end{array}\right\}\cup$$
$$\left\{\{x/\lceil t\rfloor\}\cdot C(t,\varrho_2,\varrho_3)\,\middle|\, t\in ui\mathbb{T}(\mathcal{C},\mathcal{V},\mathcal{V}_\varrho)\right\}$$

Note that although $\{x/A\}\cdot C(A,\varrho_0,\varrho_1)$ belongs to $\tilde{\zeta}_{\{x\}}(\{x/A\}\cdot C(A,n\colon B,n))$, it is not collected in the final abstraction. This is justified by the fact that there is no way to change variable x in $C(x,\varrho,\varrho)$ without "evolving" ϱ and thus it is impossible to have $\{x/A\}\cdot C(A,\varrho,\varrho)$ in $\mathcal{F}^\partial[\![P_1]\!](g(x))$ and then it must not be generated by $\tilde{\zeta}_{\{x\}}$ abstraction.

Example 3.1.24

The $\tilde{\zeta}_{\{x\}}$ abstraction of both $\mathcal{F}^\partial[\![P_1]\!](f(x))$ and $\mathcal{F}^\partial[\![P_2]\!](f(x))$ of Example 3.1.19 is $\{\varepsilon\cdot\varrho, \varepsilon\cdot A(\varrho), \varepsilon\cdot A(x)\}\cup\{\{x/t\}\cdot A(t)\,|\,t\in ui\mathbb{T}(\mathcal{C},\mathcal{V},\mathcal{V}_\varrho)\}$.

This is not a coincidence because of Theorem 3.1.45.

Example 3.1.25

The $\tilde{\zeta}_\emptyset$ abstractions of the evolving result trees of Example 3.1.2 are

$$\{\varepsilon\cdot\varrho,\ \varepsilon\cdot Head,\ \varepsilon\cdot Tail\} \qquad \left\{\varepsilon\cdot t\,\middle|\, t\lesssim \underbrace{Z:\ldots:Z}_{n}:\varrho,\ n\geq 0\right\}$$

Note that the abstraction of *zeros* is "exponentially bigger" because we have all possible replacements of a Z with a bottom variable.

Example 3.1.26

The $\tilde{\zeta}_{\{x,y\}}$ abstraction of the evolving result tree of Example 3.1.3 is

$$\big\{\{x/S^n(x')\}\cdot S^n(\varrho)\,\big|\,n\geq 0\big\}\cup\big\{\{x/S^n(Z)\}\cdot S^n(y)\,\big|\,n\geq 0\big\}\cup \big\{(\{x/S^n(Z),y/\lceil t\rfloor\})\cdot S^n(t)\,\big|\,n\geq 0,\,t\in ui\mathbb{T}(\mathcal{C},\mathcal{V},\mathcal{V}_\varrho)\big\}$$

Example 3.1.27

The $\tilde{\zeta}_{\{x,y\}}$ abstraction of the evolving result tree of Example 3.1.4 is

$$\{\varepsilon\cdot\varrho\}\cup\big\{\{x/S^n(Z),y/S^n(y')\}\cdot \mathit{True}\,\big|\,n\geq 0\big\}\cup \big\{\{x/S^{n+1}(x'),y/S^n(Z)\}\cdot \mathit{False}\,\big|\,n\geq 0\big\}$$

Example 3.1.28

The $\tilde{\zeta}$ abstraction of the Example 3.1.5 is

$$\begin{cases} f(x)\mapsto\{\varepsilon\cdot\varrho,\,\{x/A(x_1)\}\cdot B(\varrho_1)\}\\ \mathit{loop}(x)\mapsto\{\varepsilon\cdot\varrho\}\\ g(x)\mapsto\{\varepsilon\cdot\varrho,\,\{x/A(x_1)\}\cdot B(\varrho_1)\}\\ h(x)\mapsto\{\varepsilon\cdot\varrho,\,\{x/B\}\cdot C\}\end{cases}$$

The previous examples show how the $\mathbb{I}_{\mathbb{ARS}}$ denotations may explode in size, in particular, in Examples 3.1.23 and 3.1.24 they have infinite size even if their ∂ counterparts are finite. This is an expected outcome, since the direct representation of an abstraction defined by a closure operator is not condensed by construction. Thus, now it remains to tackle the pragmatical and theoretical problem of determining a condensed version of $\mathbb{I}_{\mathbb{ARS}}$ denotations.

A condensed representation is very important in view of further abstractions and, of course, the implementation of the abstract fixpoint computations. For example applying a $\mathit{depth}(k)$ cut to a condensed version of $\mathcal{F}^{\tilde{\nu}}$ will certainly give better results than using $\mathcal{F}^{\tilde{\nu}}$ directly.

Weak evolving result sets

So far, $\mathcal{F}^{\tilde{\nu}}$ has been obtained as a closure of the denotations of $\mathcal{F}^{\partial}$, namely, adding all the usage-equivalent instances of the nodes and all the possible approximations between nodes. We have seen that $\tilde{\nu}$ is not injective, therefore the program equivalence induced by $\mathcal{F}^{\tilde{\nu}}$ is coarser than that of $\mathcal{F}^{\partial}$. We can have an isomorphic representation of $\mathcal{F}^{\tilde{\nu}}$, which solves the size explosion issue. The underlying idea is somehow similar to the representation of intervals $[a,b]\subseteq\mathbb{Z}$, indeed, what is needed to retrieve the entire subset are just the two extreme points a and b. The same approach can be applied to $\mathbb{ARS}_{\overrightarrow{x_n}}$

obtaining a smarter representation for denotations. The counterparts of the extremes are partial computed results $\sigma \cdot s_1$ and $\sigma \cdot s_2$ with the same substitution which define the set of all their usage-equivalent instances and all the possible approximations in-between. We denote such intervals by $\sigma \centerdot s_1\text{–}s_2$, whereas, by an abuse of notation $\sigma \centerdot s\text{–}s$ will be denoted simply by $\sigma \centerdot s$.

Every approximated result set in $\mathbb{ARS}$ can be described by means of unions of such intervals. Although, for the same approximated result set we may have different coverings, but for technical reasons we prefer to chose coverings which contain only and all the maximal intervals w.r.t. subset inclusion.

Note that, infinite evolving result trees paths with the same substitution collapse in a single interval where the upper extreme is an abstract partial computed result of an *infinite value*. For instance, the derivation of *zeros* of Examples 3.1.2 and 3.1.25 is represented by $\varepsilon \centerdot \varrho\text{–}(Z : Z : Z : \dots)$. In the following we admit also infinite terms in $\mathbb{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho)$.

Formally, an interval $\sigma \centerdot s_1\text{–}s_2$ represents the following set of partial computed results

$$unfold_V(\sigma \centerdot s_1\text{–}s_2) := \left\{ (\sigma\lceil\vartheta\rfloor)\restriction_{\overrightarrow{x_n}} \cdot s \;\middle|\; \begin{array}{l} s_1 \lesssim t \lesssim s_2,\ \vartheta \in uiSubsts_t \\ s' \in base(t,\vartheta),\ s' \lesssim s \lesssim \psi(t\vartheta) \end{array} \right\} \tag{3.1.40}$$

Clearly for $t_1 \lesssim s_1$ and $s_2 \lesssim t_2$, $unfold_V(\sigma \centerdot s_1\text{–}s_2) \subseteq unfold_V(\sigma \centerdot t_1\text{–}t_2)$.

An approximated result set $\tilde{S}$ is represented by means of a covering the following set of intervals

$$fold_V(\tilde{S}) := \{\sigma \centerdot s_1\text{–}s_2 \,|\, \text{maximal } unfold_V(\sigma \centerdot s_1\text{–}s_2) \subseteq \tilde{S}\} \tag{3.1.41}$$

Let us define $\mathbb{WERS}_{\overrightarrow{x_n}} := fold_{\{\overrightarrow{x_n}\}}(\mathbb{ARS}_{\overrightarrow{x_n}})$ (and $\mathbb{WERS} := \bigcup_{\overrightarrow{x_n} \in \mathcal{V}} \mathbb{WERS}_{\{\overrightarrow{x_n}\}}$). For any $S \in \mathbb{WERS}_{\overrightarrow{x_n}}$ we define

$$unfold_V(S) := \bigcup \{unfold_V(\sigma \centerdot s_1\text{–}s_2) \,|\, \sigma \centerdot s_1\text{–}s_2 \in S\} \tag{3.1.42}$$

We define order, *lub* and *glb* of $\mathbb{WERS}_{\overrightarrow{x_n}}$ as

$$S_1 \mathrel{\hat{\preccurlyeq}} S_2 :\Longleftrightarrow \hat{\bigvee}\{S_1, S_2\} = S_2 \tag{3.1.43}$$

$$\hat{\bigvee}\mathcal{S} := fold_{\{\overrightarrow{x_n}\}}(\bigcup \left\{ unfold_{\{\overrightarrow{x_n}\}}(S) \;\middle|\; S \in \mathcal{S} \right\}) \tag{3.1.44}$$

$$\hat{\bigwedge}\mathcal{S} := fold_{\{\overrightarrow{x_n}\}}(\bigcap \left\{ unfold_{\{\overrightarrow{x_n}\}}(S) \;\middle|\; S \in \mathcal{S} \right\}) \tag{3.1.45}$$

$\mathbb{WERS}_{\overrightarrow{x_n}}$ is, by construction, a complete lattice.

$\mathbb{WERS}_{\overrightarrow{x_n}}$ and $\mathbb{ARS}_{\overrightarrow{x_n}}$ are isomorphic, more specifically, there is an order preserving isomorphism between them.

Proposition 3.1.29 *For all* $\overrightarrow{x_n} \in \mathcal{V}$, $(\mathbb{ARS}_{\overrightarrow{x_n}}, \subseteq) \underset{fold_{\{\overrightarrow{x_n}\}}}{\overset{unfold_{\{\overrightarrow{x_n}\}}}{\twoheadleftarrow\!\!\twoheadrightarrow}} (\mathbb{WERS}_{\overrightarrow{x_n}}, \hat{\preccurlyeq})$

Note that, by construction, in any $S \in \mathbb{WERS}$ we have at least an interval starting from a bottom variable and the substitution of all intervals starting from a bottom variable is necessarily ε.

Example 3.1.30

Consider as A the approximated result set of Example 3.1.23. $fold_{\{x\}}(A)$ is

$$\{\varepsilon \centerdot \varrho_0\text{–}C(x, \varrho_2, \varrho_3),\ \{x/A\} \centerdot C(\varrho_1, B, \varrho_3)\text{–}C(A, B, B),\ \{x/A\} \centerdot C(\varrho_1, \varrho_2, B)\text{–}C(A, B, B)\}$$

We have two intervals starting from one B either in second or third position in C both reaching $C(A, B, B)$.

Example 3.1.31

The $fold_{\{x,y\}}$ abstraction of the evolving result tree of Example 3.1.26 is

$$\left\{\{x/S^n(x')\} \centerdot S^n(\varrho) \,\middle|\, n \geq 0\right\} \cup \left\{\{x/S^n(Z)\} \centerdot S^n(y) \,\middle|\, n \geq 0\right\}$$

This denotation is essentially the same of Example 3.1.3: both have the same nodes and edges in Example 3.1.3 correspond to edges of the Hasse diagram w.r.t. $\lesssim$.

Example 3.1.32

The $fold_{\{x,y\}}$ abstraction of the evolving result tree of Example 3.1.27 is

$$\{\varepsilon \centerdot \varrho\} \cup \left\{\{x/S^n(Z), y/S^n(y')\} \centerdot \mathit{True} \,\middle|\, n \geq 0\right\} \cup \left\{\{x/S^{n+1}(x'), y/S^n(Z)\} \centerdot \mathit{False} \,\middle|\, n \geq 0\right\}$$

Also this abstraction is essentially the same of Example 3.1.4.

Example 3.1.33

The $fold_{\emptyset}$ abstraction of the evolving result tree of Example 3.1.25 is

$$\begin{cases} \mathit{zeros} \mapsto \{\varepsilon \centerdot \varrho\text{–}Z : Z : Z : \ldots\} \\ \mathit{coin} \mapsto \{\varepsilon \centerdot \varrho\text{–}\mathit{Head},\ \varepsilon \centerdot \varrho\text{–}\mathit{Tail}\} \end{cases}$$

This denotation is essentially the same of Example 3.1.2 for *coin* but for *zeros* we started from an *infinite* derivation having an "exponentially bigger" closure which boils down to a *finite* denotation (containing an infinite value which can be finitely representable as well).

We can consider weak evolving result sets as trees where we have edges connecting intervals which have the same ending and starting extremes (i.e., $\sigma \centerdot t\text{–}s$ with $\vartheta \centerdot s\text{–}r$). The previous examples confirm that this new big-step trees can still be infinite in height and width as evolving result trees. Moreover now we can also have *finite* trees that contain *infinite* term values (also in image of answers). This also shows that in particular cases the "compression" made by ζ_V is remarkable.

The sets in $\mathbb{WERS}_{\overrightarrow{x_n}}$ are a compressed representation of the sets of $\mathbb{ARS}_{\overrightarrow{x_n}}$. Sets of $\mathbb{ARS}_{\overrightarrow{x_n}}$ that can be obtained by $\nu_{\{\overrightarrow{x_n}\}}$ abstraction describe how a computed result evolves constructor by constructor (potentially allowing all sorts of interleaving in presence of multiple op-rooted expressions) in contrast with $\partial_{\{\overrightarrow{x_n}\}}$ where several constructors are

introduced "atomically". Thus, as a matter of terminology, we call sets of $\mathbb{WERS}_{\overrightarrow{x_n}}$ *weak evolving computed result sets* or *weak evolving result sets* for brevity.

Since elements of $\mathbb{WERS}_{\overrightarrow{x_n}}$ are condensed, we can finally define our new fixpoint semantics based on interpretations that consist of weak evolving result sets, indexed by $\mathbb{MGC}$. The *weak evolving result set abstraction* is obtained by composing $fold_V$ with $\tilde{\nu}_V$ (and $\tilde{\zeta}_V$) as

$$\zeta_V := fold_V \circ \tilde{\zeta}_V \qquad \nu_V := fold_V \circ \tilde{\nu}_V = \zeta_V \circ \partial_V \tag{3.1.46}$$

$$\zeta := fold \circ \tilde{\zeta} \qquad \nu := fold \circ \tilde{\nu} = \zeta \circ \partial \tag{3.1.47}$$

$$fold(\mathcal{I}^{\tilde{\nu}}) := \lambda\pi.\ fold_{var(\pi)}(\mathcal{I}^{\tilde{\nu}}(\pi)) \qquad unfold(\mathcal{I}^{\nu}) := \lambda\pi.\ unfold_{var(\pi)}(\mathcal{I}^{\nu}(\pi))$$

Now let us define $\mathbb{I}_{\mathbb{WERS}} := fold(\mathbb{I}_{\mathbb{ARS}}) = \zeta(\mathbb{I}_{\mathbb{ERT}}) = \nu(\mathbb{I}) = [\mathbb{MGC} \to \mathbb{WERS}]$. We have that $\mathbb{I}_{\mathbb{WERS}}$ is, by construction, a complete lattice. Moreover, the order preserving isomorphism lifts straightforwardly to interpretations as well.

Corollary 3.1.34 $(\mathbb{I}_{\mathbb{WERS}}, \hat{\preccurlyeq}) \underset{unfold}{\overset{fold}{\twoheadleftarrow\!\!\twoheadrightarrow}} (\mathbb{I}_{\mathbb{ARS}}, \subseteq)$

Example 3.1.35

The ζ abstraction of Example 3.1.5 is

$$\begin{cases} f(x) \mapsto \{\varepsilon \centerdot \varrho,\ \{x/A(x_1)\} \centerdot B(\varrho_1)\} \\ loop(x) \mapsto \{\varepsilon \centerdot \varrho\} \\ g(x) \mapsto \{\varepsilon \centerdot \varrho,\ \{x/A(x_1)\} \centerdot B(\varrho_1)\} \\ h(x) \mapsto \{\varepsilon \centerdot \varrho,\ \{x/B\} \centerdot C\} \end{cases}$$

Example 3.1.36

The ζ abstraction of the program of Example 3.1.8 is

$$\begin{cases} main \mapsto \{\varepsilon \centerdot \varrho\text{–}C(B, A)\} \\ h(x) \mapsto \{\varepsilon \centerdot \varrho, \{x/B\} \centerdot A\} \\ g(x) \mapsto \left\{ \begin{array}{l} \varepsilon \centerdot \varrho\text{–}C(x, \varrho_1),\ \{x/B\} \centerdot C(B, \varrho_1)\text{–}C(B, A), \\ \{x/B\} \centerdot C(\varrho_2, A)\text{–}C(B, A) \end{array} \right\} \end{cases}$$

Example 3.1.37

Consider the programs P_1 and P_2

```
f x = B C (g x)
f A = B (g D) (g D)
g A = C
```

```
f x = B C (g x)
f A = B (g A) (g A)
g A = C
```

their ∂ program denotation are

$$\mathcal{F}^{\partial}[\![P_1]\!] = \begin{cases} f(x) \mapsto\ \varepsilon \cdot \varrho \overset{\varrho}{\underset{\varrho}{\rightrightarrows}} \begin{array}{l} \{x/A\} \cdot B(\varrho_1, \varrho_2) \\ \varepsilon \cdot B(C, \varrho_2) \overset{\varrho_2}{\longrightarrow} \{x/A\} \cdot B(C, C) \end{array} \\ g(x) \mapsto \varepsilon \cdot \varrho \overset{\varrho}{\to} \{x/A\} \cdot C \end{cases}$$

$$\mathcal{F}^{\partial}[\![P_2]\!] = \begin{cases} f(x) \mapsto \; \varepsilon \cdot \varrho \xrightarrow{\varrho} \{x/A\} \cdot B(\varrho_1, \varrho_2) \overset{\varrho_1}{\underset{\varrho_2}{\rightrightarrows}} \{x/A\} \cdot B(C, \varrho_2) \;\text{(}\xrightarrow{\varrho_2}\text{)} \\ \qquad\qquad\qquad\quad \{x/A\} \cdot B(\varrho_1, C) \;\text{(}\xrightarrow{\varrho_1}\text{)} \\ \qquad\quad \varepsilon \cdot \varrho \xrightarrow{\varrho} \varepsilon \cdot B(C, \varrho_2) \xrightarrow[\varrho_2]{} \{x/A\} \cdot B(C, C) \\ g(x) \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} \{x/A\} \cdot C \end{cases}$$

and
their $\tilde{\zeta}$ abstract versions are

$$\tilde{\zeta}(\mathcal{F}^{\partial}[\![P_1]\!]) = \begin{cases} f(x) \mapsto \left\{ \begin{array}{l} \varepsilon \cdot \varrho,\, \varepsilon \cdot B(\varrho_1, \varrho_2),\, \varepsilon \cdot B(C, \varrho_2),\, \{x/A\} \cdot B(\varrho_1, \varrho_2), \\ \{x/A\} \cdot B(\varrho_1, C),\, \{x/A\} \cdot B(C, C) \end{array} \right\} \\ g(x) \mapsto \{\varepsilon \cdot \varrho,\, \{x/A\} \cdot C\} \end{cases}$$

$$\tilde{\zeta}(\mathcal{F}^{\partial}[\![P_2]\!]) = \begin{cases} f(x) \mapsto \left\{ \begin{array}{l} \varepsilon \cdot \varrho,\, \varepsilon \cdot B(\varrho_1, \varrho_2),\, \varepsilon \cdot B(C, \varrho_2),\, \{x/A\} \cdot B(\varrho_1, \varrho_2), \\ \{x/A\} \cdot B(\varrho_1, C),\, \{x/A\} \cdot B(C, \varrho_2),\, \{x/A\} \cdot B(C, C) \end{array} \right\} \\ g(x) \mapsto \{\varepsilon \cdot \varrho,\, \{x/A\} \cdot C\} \end{cases}$$

while the ζ versions are

$$\zeta(\mathcal{F}^{\partial}[\![P_1]\!]) = \begin{cases} f(x) \mapsto \left\{ \begin{array}{l} \varepsilon \centerdot \varrho\text{–}B(C, \varrho_2),\, \{x/A\} \centerdot B(\varrho_1, \varrho_2)\text{–}B(\varrho_1, C), \\ \{x/A\} \centerdot B(\varrho_1, C)\text{–}B(C, C) \end{array} \right\} \\ g(x) \mapsto \{\varepsilon \centerdot \varrho,\, \{x/A\} \centerdot C\} \end{cases}$$

$$\zeta(\mathcal{F}^{\partial}[\![P_2]\!]) = \begin{cases} f(x) \mapsto \{\varepsilon \centerdot \varrho\text{–}B(C, \varrho_2),\, \{x/A\} \centerdot B(\varrho_1, \varrho_2)\text{–}B(C, C)\} \\ g(x) \mapsto \{\varepsilon \centerdot \varrho,\, \{x/A\} \centerdot C\} \end{cases}$$

Note that, for function f, the $\tilde{\zeta}$ version of P_1 has less elements than P_2, while for the ζ version is the opposite.

The element $\{x/A\} \cdot n : B(C, \varrho_2)$ which differentiates the two denotations can be used to produce a "most general testimony" that $P_1 \not\approx^{us}_{cr} P_2$. Indeed for the program Q

```
h (B C _) = E
```

where the pattern is obtained by replacing in node n bottom variables with anonymous variables, we have that $\mathcal{B}^{cr}[\![P_1 \cup Q]\!](h(f(x))) = \{\varepsilon \cdot E\}$ whereas $\mathcal{B}^{cr}[\![P_2 \cup Q]\!](h(f(x))) = \{\varepsilon \cdot E, \{x/A\} \cdot E\}$.

This example give us the possibility to outline an interesting fact, indeed, these programs are equivalent w.r.t. the CRWL semantics [49, 50, 64], because $[\![e]\!]^{P_1}_{\text{CRWL}} = [\![e]\!]^{P_2}_{\text{CRWL}}$ for all let-expressions e. These two programs compute the same results w.r.t. let-rewriting, but have different behaviour w.r.t. let-narrowing because P_2 computes $\{x/A\} \cdot E$ whereas P_1 does not. This is not surprising because of the well-known relations between let-rewriting and let-narrowing. For this same reason it can be proved that the CRWL semantics can be obtained by (further) abstraction of $\mathcal{F}^{\nu}$.

The concretization function $\nu^{\gamma} \colon [\mathbb{MGC} \to \mathbb{WERS}] \to \mathbb{I}$ can be provided by composition with the adjunction of ζ,

$$\zeta^{\gamma}(\mathcal{I}^{\nu}) = \lambda\pi.\; \zeta^{\gamma}_{\pi}(\mathcal{I}^{\nu}(\pi)) \tag{3.1.48}$$

$$\zeta^{\gamma}_{e}(S) := \bigvee\{\bar{T} \in \mathbb{ERT}_{var(e)} \mid \zeta_{var(e)}(\bar{T}) \mathrel{\hat{\preccurlyeq}} S\} \qquad \forall e \in \mathcal{T}(\Sigma, \mathcal{V}) \tag{3.1.49}$$

obtaining

$$\nu^{\gamma}(\mathcal{I}^{\nu}) = \lambda\pi.\bigsqcup\{T \in \mathbb{WSST}_{\pi} \mid \nu_{var(\pi)}(T) \mathrel{\hat{\preccurlyeq}} \mathcal{I}^{\nu}(\pi)\}.$$

The semantics induced by the weak evolving result set abstraction

The optimal abstract version of $\mathcal{P}[\![P]\!]$ w.r.t. ν, $\mathcal{P}^{\nu}[\![P]\!] := \nu \circ \mathcal{P}[\![P]\!] \circ \nu^{\gamma} = \zeta \circ \mathcal{P}^{\partial}[\![P]\!] \circ \zeta^{\gamma}$, is

$$\mathcal{P}^{\nu}[\![P]\!]_{\mathcal{I}^{\nu}} = \lambda f(\overrightarrow{x_n}).\ \hat{\bigvee}_{f(\overrightarrow{t_n}) \to r \ll P} zap^{\{\overrightarrow{x_n}/\overrightarrow{t_n}\}}_{\{\overrightarrow{x_n}\}}(\mathcal{E}^{\nu}[\![r]\!]_{\mathcal{I}^{\nu}}) \tag{3.1.50}$$

where zap is the extension over sets of intervals of

$$zap^{\vartheta}_{V}(\sigma \centerdot s_1 – s_2) := \begin{cases} \sigma \centerdot s_1 – s_2 & \text{if } s_1 \in \mathcal{V}_{\varrho} \\ (\vartheta\sigma)\restriction_V \centerdot s_1 – s_2 & \text{otherwise} \end{cases} \tag{3.1.51}$$

and the abstract evaluation of $e \in \mathcal{T}(\Sigma, \mathcal{V})$ w.r.t. an interpretation $\mathcal{I}^{\nu}$, namely $\mathcal{E}^{\nu}[\![e]\!]_{\mathcal{I}^{\nu}}$, is defined by induction on the size of $knots(e)$ as

$$\mathcal{E}^{\nu}[\![x]\!]_{\mathcal{I}^{\nu}} := \{\varepsilon \centerdot \varrho – x\} \tag{3.1.52a}$$

$$\mathcal{E}^{\nu}[\![\varphi(\overrightarrow{x_n})]\!]_{\mathcal{I}^{\nu}} := \mathcal{I}^{\nu}(\varphi(\overrightarrow{y_n}))[y_1/\,\mathcal{E}^{\nu}[\![x_1]\!]_{\mathcal{I}^{\nu}}]_{V_1} \ldots [y_n/\,\mathcal{E}^{\nu}[\![x_n]\!]_{\mathcal{I}^{\nu}}]_{V_n} \tag{3.1.52b}$$
$$\overrightarrow{y_n} \ll \mathcal{V} \text{ distinct}, V_i := \{x_1, \ldots, x_i, y_{i+1}, \ldots, y_n\}$$

$$\mathcal{E}^{\nu}[\![e]\!]_{\mathcal{I}^{\nu}} := \mathcal{E}^{\nu}[\![e[y]_p]\!]_{\mathcal{I}^{\nu}}[y/\,\mathcal{E}^{\nu}[\![e|_p]\!]_{\mathcal{I}^{\nu}}]_{var(e)} \tag{3.1.52c}$$
$$y \ll \mathcal{V}, \text{ leftmost } p \in baseknots(e)$$

provided that the abstract tree-embedding operation $G[x/B]_V$ over two weak evolving result sets is

$$G[x/B]_V := \hat{\bigvee}\{i \mid i_g \in G,\ i_b \in B,\ i_g[x/i_b]_V \vdash i\} \tag{3.1.53}$$

where $(\vartheta \centerdot g_1 – g_2)[x/(\eta \centerdot b_1 – b_2)]_V \vdash \sigma \centerdot s_1 – s_2$ if exist $g, b \in \mathbb{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_{\varrho})$ such that $g_1 \lesssim g \lesssim g_2$ and b is the least one such that $b_1 \lesssim b \lesssim b_2$ and

1. $\exists\,\delta := \vartheta \uparrow \eta\{x/b\}$ s.t. it is $\mathcal{V}_{\varrho}$-preserving and $\forall y \in \mathcal{V} \cap var(g).\ y\delta \notin \mathcal{V}_{\varrho}$,
2. for all p, if $b|_p \notin \mathcal{V}_{\varrho}$ then $(x\vartheta)|_p \notin var(x\vartheta) \setminus var(g)$

for $\sigma := \delta\restriction_V$, $s_1 \in base(g, \delta)$ and $s_2 := \psi(g_2(\vartheta \uparrow \eta\{x/b_2\}))$.

Broadly speaking, $G[x/B]_V$ is the *lub* of all the intervals obtained as the inlay of an interval of B into one of G. The inlaying of an interval $\eta \centerdot b_1 – b_2$ for a variable x into an interval $\vartheta \centerdot g_1 – g_2$, namely $\vartheta \centerdot g_1 – g_2[x/\eta \centerdot b_1 – b_2]_V \vdash \sigma \centerdot s_1 – s_2$, is done by determining the starting and the ending terms of the inlay (respectively s_1 and s_2), which are obtained by a sort of parameter passing handled as the application of two specific substitutions.

The conditions determine two elements from the given intervals, namely $\vartheta \cdot g$ and $\eta \cdot b$, which are suitable for the construction of the start s_1 of the resulting inlay. In particular, Point 1 constrains the parameter value b to be sufficiently evolved as the "parameter

pattern" $x\vartheta$ demands, while Point 2 forbids b to be evolved in positions which are not demanded. Once g and b have been determined, s_1 is obtained by updating, with $base(g, \delta)$, variables in g with the computed substitution δ but only on single occurrencies of variables. Namely, in case we have duplicated variables in g only one occurrence is bound, while the other occurrencies are approximated with a bottom variable.

The end s_2 can be simply obtained by "embedding" the maximal extreme b_2 into g_2, by using the $\mathcal{V}_\varrho$-preserving substitution $\vartheta \uparrow \eta\{x/b_2\}$. The existence of such a substitution is guaranteed by Point 1. In this case we don't care about demandness because non demanded "over computed" contributions are referred to variables not occurring in g_2.

Our weak evolving result set fixpoint denotation is defined as

$$\mathcal{F}^\nu[\![P]\!] := lfp(\mathcal{P}^\nu[\![P]\!]) \tag{3.1.54}$$

Example 3.1.38

The first two iterates for the program of Example 2.2.21 are

$$\mathcal{P}^\nu[\![P]\!]\uparrow 1 = \begin{cases} zeros \mapsto \{\varepsilon \centerdot \varrho_1\text{–}Z : \varrho\} \\ coin \mapsto \{\varepsilon \centerdot \varrho\text{–}Head,\ \varepsilon \centerdot \varrho\text{–}Tail\} \end{cases}$$

$$\mathcal{P}^\nu[\![P]\!]\uparrow 2 = \begin{cases} zeros \mapsto \{\varepsilon \centerdot \varrho_2\text{–}Z : Z : \varrho\} \\ coin \mapsto \mathcal{P}^\nu[\![P]\!]\uparrow 1(coin) \end{cases}$$

while its least fixed point is the same of Example 3.1.33. This is not a casualty, because of Corollary 3.1.44.

Example 3.1.39

The first two iterates for the program of Example 2.2.13 are

$$\mathcal{P}^\nu[\![P_+]\!]\uparrow 1 = \Big\{ x + y \mapsto \{\varepsilon \centerdot \varrho,\ \{x/Z\} \centerdot y,\ \{x/S(x_0)\} \centerdot S(\varrho)\}$$

$$\mathcal{P}^\nu[\![P_+]\!]\uparrow 2 = \left\{ x + y \mapsto \left\{ \begin{array}{l} \varepsilon \centerdot \varrho,\ \{x/Z\} \centerdot y,\ \{x/S(x_0)\} \centerdot S(\varrho), \\ \{x/S(Z)\} \centerdot S(y),\ \{x/S(S(x_0))\} \centerdot S(S(\varrho)) \end{array} \right\} \right.$$

while $\mathcal{F}^\nu[\![P_+]\!](x + y)$ is the same of Example 3.1.31.

Example 3.1.40

The first two iterates for the program of Example 3.1.4 are

$$\mathcal{P}^\nu[\![P_\leq]\!]\uparrow 1 = \Big\{ leq(x, y) \mapsto \{\varepsilon \centerdot \varrho,\ \{x/Z, y/S(y_0)\} \centerdot True,\ \{x/S(x_0), y/Z\} \centerdot False\}$$

$$\mathcal{P}^\nu[\![P_\leq]\!]\uparrow 2 = \left\{ leq(x, y) \mapsto \left\{ \begin{array}{l} \varepsilon \centerdot \varrho,\ \{x/Z, y/S(y_0)\} \centerdot True,\ \{x/S(x_0), y/Z\} \centerdot False \\ \{x/S(Z), y/S(S(y_0))\} \centerdot True,\ \{x/S(S(x_0)), y/S(Z)\} \centerdot False \end{array} \right\} \right.$$

while $\mathcal{F}^\nu[\![P_\leq]\!](leq(x, y))$ is the same of Example 3.1.32.

Example 3.1.41

Consider the program of Example 3.1.8, its weak evolving result set semantics is obtained in three iterations.

$$\mathcal{P}^\nu[\![P]\!]\uparrow 1 = \begin{cases} main \mapsto \{\varepsilon \centerdot \varrho\} \\ h(x) \mapsto \{\varepsilon \centerdot \varrho, \{x/B\} \centerdot A\} \\ g(x) \mapsto \{\varepsilon \centerdot \varrho\text{–}C(x, \varrho_1)\} \end{cases}$$

$$\mathcal{P}^\nu[\![P]\!]\uparrow 2 = \begin{cases} main \mapsto \{\varepsilon \centerdot \varrho\text{–}C(B, \varrho_1)\} \\ h(x) \mapsto \mathcal{P}^\nu[\![P]\!]\uparrow 1(h(x)) \\ g(x) \mapsto \left\{ \begin{array}{l} \varepsilon \centerdot \varrho\text{–}C(x, \varrho_1),\ \{x/B\} \centerdot C(B, \varrho_1)\text{–}C(B, A), \\ \{x/B\} \centerdot C(\varrho_2, A)\text{–}C(B, A) \end{array} \right\} \end{cases}$$

$$\mathcal{P}^\nu[\![P]\!]\uparrow 3 = \begin{cases} main \mapsto \{\varepsilon \centerdot \varrho\text{–}C(B, A)\} \\ h(x) \mapsto \mathcal{P}^\nu[\![P]\!]\uparrow 1(h(x)) \\ g(x) \mapsto \mathcal{P}^\nu[\![P]\!]\uparrow 2(g(x)) \end{cases}$$

and finally $\mathcal{P}^\nu[\![P]\!]\uparrow 4 = \mathcal{P}^\nu[\![P]\!]\uparrow 3 = \mathcal{F}^\nu[\![P]\!]$.

Example 3.1.42

The iterates for the program of Example 3.1.12 are

$$\mathcal{P}^\nu[\![P]\!]\uparrow 1 = \begin{cases} val(x) \mapsto \{\varepsilon \centerdot \varrho_1\text{–}(x, \varrho_0)\} \\ isleaf(x) \mapsto \{\varepsilon \centerdot \varrho_1, \{x/N(v_1, V, V)\} \centerdot True\} \\ h(x) \mapsto \{\varepsilon \centerdot \varrho_1\text{–}N(1, \varrho_0, V)\} \\ hson(x) \mapsto \{\varepsilon \centerdot \varrho, \{x/0\} \centerdot V, \{x/1\} \centerdot U\} \\ main(x) \mapsto \{\varepsilon \centerdot \varrho_1\} \end{cases}$$

$$\mathcal{P}^\nu[\![P]\!]\uparrow 2 = \begin{cases} val(x) \mapsto \{\varepsilon \centerdot \varrho_1\text{–}(x, \varrho_0), \{x/N(v_1, V, V)\} \centerdot (N(v_1, V, V), True)\} \\ isleaf(x) \mapsto \mathcal{P}^\nu[\![P]\!]\uparrow 1(isleaf(x)) \\ h(x) \mapsto \{\varepsilon \centerdot \varrho_1\text{–}N(1, \varrho_0, V), \{x/0\} \centerdot N(1, V, V), \{x/1\} \centerdot N(1, U, V)\} \\ hson(x) \mapsto \mathcal{P}^\nu[\![P]\!]\uparrow 1(hson(x)) \\ main(x) \mapsto \{\varepsilon \centerdot \varrho_1\text{–}(N(1, \varrho_0', V), \varrho_0)\} \end{cases}$$

$$\mathcal{P}^\nu[\![P]\!]\uparrow 3 = \begin{cases} val(x) \mapsto \mathcal{P}^\nu[\![P]\!]\uparrow 2(val(x)) \\ isleaf(x) \mapsto \mathcal{P}^\nu[\![P]\!]\uparrow 1(isleaf(x)) \\ h(x) \mapsto \mathcal{P}^\nu[\![P]\!]\uparrow 2(h(x)) \\ hson(x) \mapsto \mathcal{P}^\nu[\![P]\!]\uparrow 1(hson(x)) \\ main(x) \mapsto \left\{ \begin{array}{l} \varepsilon \centerdot \varrho_1\text{–}(N(1, \varrho_0', V), \varrho_0),\ \{x/1\} \centerdot (N(1, U, V), \varrho_2) \\ \{x/0\} \centerdot (N(1, V, V), \varrho_2)\text{–}(N(1, V, V), True) \end{array} \right\} \end{cases}$$

and finally $\mathcal{P}^\nu[\![P]\!]\uparrow 4 = \mathcal{P}^\nu[\![P]\!]\uparrow 3 = \mathcal{F}^\nu[\![P]\!]$. Note how the various paths leading to the same computed result in Example 3.1.12 have been compacted into the same interval.

Properties of the weak evolving result set semantics

The abstract evaluation operator $\mathcal{E}^\nu$ is precise.

Theorem 3.1.43 *For all* $e \in \mathcal{T}(\Sigma, \mathcal{V})$ *and all* $\mathcal{I} \in \mathbb{I}$, $\nu_{var(e)}(\mathcal{E}[\![e]\!]_{\mathcal{I}}) = \zeta_{var(e)}(\mathcal{E}^{\partial}[\![e]\!]_{\partial(\mathcal{I})}) = \mathcal{E}^{\nu}[\![e]\!]_{\nu(\mathcal{I})}$.

With this result we can prove that $\mathcal{P}^{\nu}[\![P]\!]$ is precise as well.

Corollary 3.1.44 *For all* $P \in \mathbb{P}_{\Sigma}$,

1. $\mathcal{P}^{\nu}[\![P]\!] \circ \zeta = \zeta \circ \mathcal{P}^{\partial}[\![P]\!]$, $\mathcal{P}^{\nu}[\![P]\!] \circ \nu = \nu \circ \mathcal{P}[\![P]\!]$,
2. $\mathcal{P}^{\nu}[\![P]\!]$ *is continuous,*
3. $\mathcal{F}^{\nu}[\![P]\!] = \mathcal{P}^{\nu}[\![P]\!]\uparrow\omega$,
4. $\mathcal{F}^{\nu}[\![P]\!] = \zeta(\mathcal{F}^{\partial}[\![P]\!]) = \nu(\mathcal{F}[\![P]\!])$.
5. $\{\sigma \cdot \lfloor v \rfloor \mid \sigma \cdot v \in \mathcal{F}^{\partial}[\![P]\!], v \in \mathcal{T}(\mathcal{C}, \mathcal{V})\} = \{\sigma \cdot v \mid \sigma \centerdot t\text{–}v \in \mathcal{F}^{\nu}[\![P]\!], v \in \mathbb{T}(\mathcal{C}, \mathcal{V})\}$

Theorem 3.1.45 (Full abstraction of $\mathcal{F}^{\nu}$ w.r.t. $\approx_{cr}^{us}$) *For all* $P_1, P_2 \in \mathbb{P}_{\Sigma}$, $\mathcal{F}^{\nu}[\![P_1]\!] = \mathcal{F}^{\nu}[\![P_2]\!]$ *if and only if* $P_1 \approx_{cr}^{us} P_2$.

This completes the story about *the* big-step semantics, since we have indeed found a fully abstract semantics w.r.t. *the* reasonable Curry program equivalence $\approx_{cr}^{us}$!

3.2 Modeling the Big-Step Operational Behaviour of Haskell programs

In this section we present a systematic way to convert a first-order Haskell program into a first-order Curry one which will "preserve small-step semantics", in the sense that from the small-step operational (Curry) semantics (defined in Subsection 2.1.1) of the converted program we can retrieve the small-step operational (Haskell) semantics (defined in Subsection 2.1.2) of the original Haskell program. This will permit us to easily reuse all the results given in this thesis for first-order Curry programs to first-order Haskell programs.

3.2.1 Conversion algorithm

In the following, we assume that all rules are unconditional (it is obvious how to extend it to conditional rules since only the left-hand sides of the rules are relevant for the definitional trees). To specify the construction algorithm, we define a partial function $IP(\pi, l \to r)$ which returns the pair $\langle p, \tau \rangle$ such that

- p is the leftmost position of a variable in π such that $l|_p$ is constructor rooted, and
- τ is the type of the constructor symbols which labels the node p in l

If there is no position satisfying the first condition, $IP(\pi, l \to r)$ is left undefined.

The generation of a definitional tree for a call pattern π and a non-empty list of rules P (where l unifies π for each $l \to r$ occurring in P) is described by the function $gt(\pi, P)$. Let R be the first rule of P, we distinguish the following cases for gt:

- if $IP(\pi, R) = \langle p, \tau \rangle$, let $c_1, \ldots, c_k$ be all the constructors of type τ with arities $n_1, \ldots, n_k$ respectively and $P_i = [l \to r \in P \mid l \text{ and } \pi \text{ unify}]$, then

$$gt(\pi, P) := branch(\pi, p, T_1, \ldots, T_h)$$

  where $\{T_1, \ldots, T_h\} = \{gt(\pi[c_i(x_1, \ldots, x_{n_i})]_p, P_i) \mid 1 \leq i \leq k,\ P_i \text{ not empty}\}$

- if $IP(\pi, R)$ is undefined, let $R = l \to r$ and $\sigma = mgu(l, \pi)$, then

$$gt(\pi, P) := rule(l\sigma \to r\sigma)$$

If P_f is the list of rules defining the function $f_{/n}$ in the Haskell program P, then, the definitional tree generated by computing $gt(f(x_1, \ldots, x_n), P)$ is the definitional tree for f in the converted Curry program $Cnv(P)$.

In summary, given a Haskell program P, its conversion $Cnv(P)$ (into a Curry program) is:

$$Cnv(P) := \{gt(f(x_1, \ldots, x_n), P_f) \mid P_f \text{ subprogram of } P \text{ defining the function } f_{/n}\} \tag{3.2.1}$$

Remark 3.2.1 The conversion algorithm always terminates computing a definitional tree for each Haskell function. Indeed, the number of constructors in each Haskell rule is supposed to be finite, and in each recursive call $gt(\pi, R)$ the number of constructors in π increases. Thus $IP(\pi, R)$ will be eventually undefined leading to the base case of gt. ■

Observation 3.2.2 *It is worth noting that, by construction, the given conversion produces only definitional trees with either branch nodes or rule nodes, thus any Curry program computed by the algorithm leads to an* inductively sequential *TRS. It is well known that this class of TRS is confluent w.r.t. rewriting.*

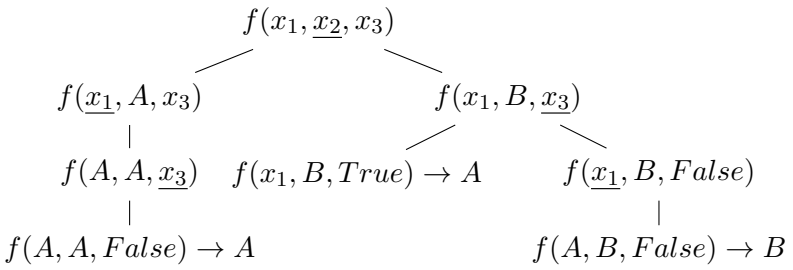
Example 3.2.3

Consider the program

```
data T = A | B
data Bool = True | False

f x B True = A
f A y False = y
```

Its the conversion produces the following definitional tree



The conversion process may yield to a Curry program with more rules than the original Haskell one. For instance, in Example 3.2.3, the equation `f A y False = y` produces two contributes, namely $f(A, B, False) \to B$ and $f(A, A, False) \to A$.

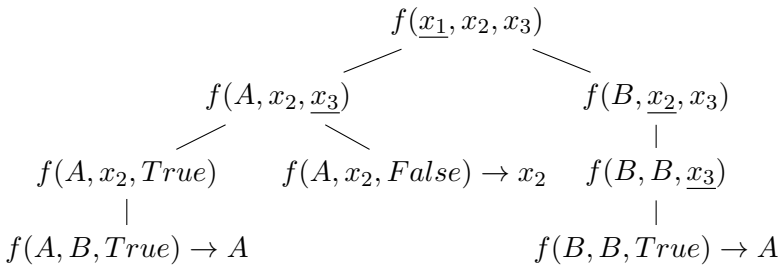
As the reader could expect, a different permutation of the same rules may lead to a different converted program.

Example 3.2.4

Inverting the rule order in the program of Example 3.2.3, that is

```
f A y False = y
f x B True = A
```

the the conversion yields to the following definitional tree



Although the programs of Examples 3.2.3 and 3.2.4 have the same set of rules, the relative order of the rules may change the program behavior. Consider for instance a looping expression $\perp$ and the expression $f(\perp, B, True)$. Then, $f(\perp, B, True)$ is evaluated to A with the program of Example 3.2.3, while it does not terminate with that of Example 3.2.4.

The converted Curry program is not always bigger than the original one, as the following example shows.

Example 3.2.5

Consider the program

```
g x = A
g B = B
```

The algorithm produces the definitional tree $rule(g(x) \to A)$, discarding the equation `g B = B`. This happens whenever a rule R occurs after another rule whose LHS subsumes that of R, making R useless in the program[7].

The generation process for a definitional tree of $Cnv(P)$ preserves this invariant: every call to gt is performed with a call pattern π and a subprogram Q of P which contains all the rules of P whose LHS unifies with π. Moreover, whenever a call $gt(\pi, P)$ is responsible for a recursive call $gt(\pi_i, P_i)$, π_i is equal to $\pi[c_i(x_1, \ldots, x_{n_i})]_p$ for some constructor c_i, where $IP(\pi, l \to r) = \langle p, \tau \rangle$ and $l \to r$ is the first rule occurring in P. Thus, for any ground expression e which unifies π, we have that $l \to r$ is the first feasible rule for e in P if and only if either

[7] This kind of situations are reported as a waring by the Glasgow Haskell Compiler.

- e unifies one of the call patterns π_i, or
- $e|_p$ is operation rooted.

Starting from the former considerations, we can relate the (small-step) operational behavior of a Haskell program with its conversion into a Curry one.

Proposition 3.2.6 *Given the Haskell program P, and two ground expressions e_1 and e_2, then, $HEval[\![e_1]\!] \overset{p}{\Rightarrow} e_2$ holds for P if and only if $Eval[\![e_1]\!] \overset{p,\varepsilon}{\Longrightarrow} e_2$ holds for $Cnv(P)$.*

3.2.2 Curry small-step behavior vs. Haskell small-step behavior

Now we define the small-step (operational) behavior $\mathcal{B}^{fss}_H[\![P]\!]$ of a Haskell program P, w.r.t. the rewriting strategy introduced in Subsection 2.1.2. As done for Curry it will be defined as the collection all the sequences of computations steps for all possible initial expressions.

Definition 3.2.7 (small-step behavior of Haskell programs) *For P a Haskell program, let the small-step behavior of P be defined by*

$$\mathcal{B}^{H}[\![P]\!] := \left\{ e_0 \underset{p_1}{\Rightarrow} e_1 \underset{p_2}{\Rightarrow} \dots \;\middle|\; e_0 \text{ ground term}, \forall i.\ HEval[\![e_i]\!] \overset{p_i}{\Rightarrow} e_{i+1} \right\}$$

We indicate with $\approx_H$ the equivalence relation induced by $\mathcal{B}^H$, namely $P_1 \approx_H P_2$ if and only if $\mathcal{B}^H[\![P_1]\!] = \mathcal{B}^H[\![P_2]\!]$.

Note that, Definition 3.2.7 considers as initial expressions only ground terms, instead of (possibly non ground) terms as in Definition 2.2.2.

An immediate consequence of Proposition 3.2.6 is

Corollary 3.2.8 *For any Haskell program P, $\mathcal{B}^H[\![P]\!] = \mathcal{B}^{fss}_{gr}[\![Cnv(P)]\!]$.*

Where, $\mathcal{B}^{fss}_{gr}[\![Q]\!] := \{d \in \mathcal{B}^{fss}[\![Q]\!] \mid \text{the head of } d \text{ is ground}\}$ for any Curry program Q.

Theorem 3.2.9 *Given P_1 and P_2 two Haskell programs, then $P_1 \approx_H P_2$ if and only if $Cnv(P_1) \approx_{fss} Cnv(P_2)$.*

Definition 3.2.10 (computed result behaviour for Haskell) *Given a Haskell program P, its computed result behaviour is*

$$\mathcal{B}^{cr}_H[\![P]\!] := \lambda e. \left\{ \wr v \int \;\middle|\; e \Rightarrow^* v \in \mathcal{B}^{ss}_H[\![P]\!],\ v \text{ ground data term} \right\} \tag{3.2.2}$$

We indicate with $\overset{H}{\approx}_{cr}$ the program equivalence relation induced by $\mathcal{B}^{cr}_H$, namely $P_1 \overset{H}{\approx}_{cr} P_2 \Leftrightarrow \mathcal{B}^{cr}_H[\![P_1]\!] = \mathcal{B}^{cr}_H[\![P_2]\!]$.

The deterministic nature of Haskell permits to formalize the computed result behavior as a partial function from ground terms to ground data terms. We preferred this alternative version just for symmetry w.r.t. Equation (3.1.1).

Proposition 3.2.11 *Let P a Haskell program, then $\mathcal{B}^{cr}_H[\![P]\!](e) = \mathcal{B}^{cr}[\![Cnv(P)]\!](e)$ for every ground expression e.*

3.3 Related Works

The weak evolving result set (evolving result tree) semantics which we present here is the first *condensed* goal-independent semantics which is fully abstract (correct) w.r.t. $\approx_{cr}^{us}$. In the literature there are several proposals that are correct but not fully abstract, or are fully abstract but not goal-independent, or fully abstract and goal-independent but not condensed. The issue is the simultaneous presence of *all* these characteristics, which we do not consider an option, because we are mainly interested in an application of the semantics for the construction of (semantics-based) program treatment tools, and our experience has shown how the use of a condensed goal-independent semantics dramatically effects the results.

As happened for other paradigms, the condensed goal-independent proposal has to "pay the fee" for the extra "expressive power" and the technical definition is more complex than other (non-condensed) proposals.

Even if there is no strictly comparable proposal, given the "long way" approach that we followed to reach our goal, in this work we are actually proposing several intermediate semantics which can be compared to other proposal by considering only *some* of the features. We try to illustrate in detail the several possibilities.

Goal-dependent approaches.

- The small-step (operational) semantics of [1] is essentially isomorphic to our small-step operational semantics. We have chosen to develop our version because:
  - its definition is (essentially a refinement of) the official small-step operational semantics [51, App. D.4] and so, as one can easily see, it is much closer (and more faithful) to the original version than the proposal of [1];
  - it does not "destroy" sharing on constructor nodes as [1] does, and thus it is much closer to what happens in implementations;
  - most importantly, with this form it is easier to define the big-step abstraction.
- The big-step (operational) semantics of [1] is just the collection of the value terminating small-step computations. Even if formally the big-step relation relates goals with computed results, its definition is inherently based on the construction of a small-step computation in order to determine the computed results. In particular it is not suited for developing analyses methods because it eventually needs to compute small-step sequences and one has to invent from scratch an ad hoc algorithm which statically terminates.

  It is completely different w.r.t. our big-step proposal, which computes just the needed part of computed results within denotations, without having any information at all about intermediate reducts. Analyses methods can be (quite) easily constructed on top of it (as shown in Subsection 4.2.1).
- There are several aspects of our proposal similar to those of CRWL semantics [49, 50, 64], from which we have taken several inspirations. The use of bottom variables in $\mathcal{F}^{\nu}$ instead of a unique bottom symbol like in CRWL is, limiting to this paper, mostly cosmetic. We are using bottom variables just in view

of future applications because it can be essential (or at least helpful) in the definition of abstractions.

The main difference is that of moving from a logical (denotational) approach (whose outcome is a non-condensed semantics, closed under instance) to a condensed (with a more operational flavor, but still denotational) approach. The result, as also pointed out in Example 3.1.37, is a semantics fully abstract w.r.t. let-narrowing outcomes ($\approx^{us}_{cr}$), instead of a semantics fully abstract w.r.t. let-rewriting [63] (but at a higher cost).

Goal-independent non-condensed approaches. [68], based on the CRWL semantics, shows a goal-independent semantics which is fully abstract w.r.t. let-rewriting outcomes. This proposal is not condensed, being closed under instance. It could be possible to provide a condensed representation by keeping just the crown of denotations, but it would nevertheless be fully abstract w.r.t. let-rewriting and not w.r.t. let-narrowing outcomes ($\approx^{us}_{cr}$).

Furthermore we are convinced that the result of such reformulation would lead to a proposal as complex as ours.

It can be formally shown that interpretations of [68] can be obtained by (further) abstraction of weak evolving result set semantics (with a very redundant actual representation).

Goal-independent condensed approaches. [3] proposed a condensed goal-independent fixpoint semantics for Term Rewriting Systems. It is defined for some (important) subclasses of TRS (almost orthogonal, right linear, topmost) while we consider left linear construction based TRS with free variables (which is the complete class of first order functional logic languages). Our proposal certainly originates from the experience made in [3] but the target languages are substantially different and thus a detailed comparison is not very relevant.

Anyway, the major drawback of the proposal of [3], in view of employing that semantics for abstract diagnosis, is that buggy programs cannot reasonably satisfy *a priori* any condition at all, so one cannot guarantee to have an applicable methodology. Our proposal on the contrary poses no conditions and thus is always applicable (as shown in [9]).

Abstract approaches. In the literature there are some abstract semantics which can be compared with some of our abstract proposals. For instance, the $depth(k)$ abstract semantics proposed in [9] (abstraction of evolving result tree semantics) encompasses some limitations of previous works on the same subject. Namely,

- Since we use a "truly" goal-independent concrete semantics we obtain a much more condensed abstract semantics than [4]. This is exactly the reason why we started our research on condensed semantics for generic TRS ([3]) and functional logic languages (this work).
- For $k = 1$ if we consider a TRS which is admissible to apply the technique of [11] we obtain the same results. However the abstract rewriting methodology of [11] requires canonicity, stratification, constructor discipline, and complete definedness for the analyses. This class of TRS is very restricted (even for

functional programming) and certainly cannot cover functional logic programs. On the contrary we require only left linearity and construction basedness.

- Also [52] works with the domain $depth(k)$. However it uses another abstraction of the terms (accounting sharing in a different way) and, most importantly, the baseline semantics is much more concrete than evolving result tree semantics, since it models call patterns. We are very interested in the future in defining an abstraction to model call patterns, as it will be the base to define a framework for Abstract Verification for Curry. In that occasion we will also compare that proposal to [52].

3.4 Discussion on the results

In this chapter three bottom-up fixpoint semantics for first order fragment of Curry are given. We started by giving a concrete bottom-up fixpoint semantics which we showed to be fully abstract w.r.t. the small-step behavior. Then, using abstract interpretation techniques, we obtained a hierarchy of fixpoint semantics by successive abstractions. This methodology has served both to design the semantics and to relate them with the small-step operational semantics in a formal way.

The first semantics models the small-step behaviour of programs by means of a collection of trees, small-step trees, describing the small-step computations of each operator.

Small-step trees have been abstracted to evolving result trees. These trees describe the evolution of data constructors by means of paths of partial computed results. In this way we obtained a fixpoint goal-independent semantics which is correct w.r.t. the computed result behaviour.

A further abstraction on evolving result trees has been performed, obtaining what we called weak evolving result set semantics, where denotations now consist in a collection of weak evolving result sets which describe how partial computed results evolve one constructor symbol at a time. The weak evolving result set semantics fulfills all (our) desired requirements:

- is fully abstract w.r.t. the $\approx_{cr}^{us}$ behaviour,
- has a goal-independent definition,
- is the fixpoint of a bottom-up construction, and
- is as condensed as possible.

The program equivalence $\approx_{cr}^{us}$ is good for reasoning in terms of module composition. Indeed, whenever $P_1 \approx_{cr}^{us} P_2$ holds we can consider the two programs as different implementations of the same module in the sense that, one can choose to use in a program Q either the operations defined in P_1 or those in P_2 without affecting the computed result behavior of Q. In contrast to similar program equivalence notions [63, 68] based on (Higher Order) Construction-based Conditional Rewriting Logic (HO-CRWL), we have stronger results. In particular, our semantics models precisely the computed result behaviour w.r.t. usage.

Moreover, among all other proposals, the evolving result tree semantics is the first one which achieves, at the same time, goal-independency and condensedness. The experience

of the senior members of our research group shows that these properties are essential for the development of efficacious semantics-based program manipulation tools, e.g. automatic program analyzers, debuggers, *etc.*.

Furthermore, we introduced a method to automatically convert a well typed first order Haskell program into a semantically equivalent inductively sequential Curry program. We investigated on the adequateness of this transformation both w.r.t. the small-step and the big-step behaviour. As a consequence, our semantics can be used to handle well typed first order Haskell programs as well.

In the future we are interested in extending our results to all the features of Curry: equational constraints (i.e., strict equality), residuation (in order to tackle strict primitive arithmetic operations) and higher-order. To do so, we think it would be profitable to formalize the partial computed results as elements of a cylindric constraint system ([86]) and, once the semantics has been reformulated in such terms, then extend the constraint system with strict equality and arithmetic primitives. Furthermore, cylindric constraint systems go particularly well with abstract interpretation techniques, as good abstractions can be obtained by abstracting only the constraint system while keeping the overlying structure.

Another interesting possibility, along the lines of this construction, is to develop (by abstraction of small-step trees) a semantics which can model "functional dependencies". Such a semantics is needed to tackle pre-post conditions and could be employed to define the homologous of abstract verification for logic programs [24] in the functional logic paradigm.

3.A Proofs

Proof of Proposition 3.1.7.
Point 1) Immediate by Equation (3.1.4).

Point 2) ∂ is surjective, therefore for any path $\sigma_0 \cdot \bar{s}_0 \xrightarrow{\varrho_1} \ldots \xrightarrow{\varrho_n} \sigma_n \cdot \bar{s}_n$ in $\bar{T}$ there exists a path $e_0 \overset{\vartheta_1}{\underset{p_1}{\hookrightarrow}} \ldots \overset{\vartheta}{\underset{p_n}{\hookrightarrow}} e_n$ such that $\bar{s}_i = \tau(e_i)$, $\sigma_0 = \emptyset$, $\sigma_{i+1} = (\vartheta_1 \cdots \vartheta_{i+1})\restriction_{e_0}$, and $\varrho_{i+1} = \bar{s}_i|_{p_{i+1}}$ for every $0 \leq i \leq n$. Thus $\varrho_{i+1} \in var(\bar{s}_i)$ for any $0 \leq i \leq n$. For every $0 \leq i \leq n$ we have that

$$
\begin{aligned}
&\tau(e_{i+1}) = && [\text{Definition 2.2.1 for some } s]\\
&= \tau((e_i[s]_{p_{i+1}})\vartheta_{i+1}) && [\text{by by Equation (3.1.2), since } \vartheta \in \mathcal{C}Substs]\\
&= \tau(e_i[s]_{p_{i+1}})\vartheta_{i+1} && [\text{by Equation (3.1.2)}]\\
&= \tau(e_i)[\tau(s)]_{p_{i+1}} && [\text{since } e_i|_{p_{i+1}} \text{ is } \mathcal{D}\text{-rooted, for some } \hat{\rho} \in \varrho Substs]\\
&= \tau(e_i)\hat{\rho}\vartheta_{i+1}
\end{aligned}
$$

Hence, by Definition 3.1.6, $\vartheta_i \cdot \bar{s}_i \twoheadrightarrow_{\varrho_{i+1}} \vartheta_{i+1} \cdot \bar{s}_{i+1}$ for every $0 \leq i \leq n$.

Point 3) Immediate by Equation (3.1.4) and definition of *prfxtree*.

Point 4) Since ∂ is surjective, there exists $T \in \mathbb{WSST}_e$ such that $\partial_{var(e)}(T) = \bar{T}$. The proof follows by Proposition 2.2.4.

Proof of Proposition 3.1.9.

$\bar{T}_1 \preccurlyeq \bar{T}_2 \iff$

[by Equation (3.1.10)]

$\bar{T}_1 \curlyvee \bar{T}_2 = \bar{T}_2 \iff$

[by Equation (3.1.11) and ∂_V surjectivity, given $V := var(e)$]

$\partial_V(\{T \in \mathbb{WSST}_e \mid \partial_V(T) = \bar{T}_1\} \sqcup \{T \in \mathbb{WSST}_e \mid \partial_V(T) = \bar{T}_2\}) =$
$\partial_V(\{T \in \mathbb{WSST}_e \mid \partial_V(T) = \bar{T}_2\}) \iff$

[by Equation (3.1.4) and $\bigsqcup$ definition]

$prfxtree(\bigcup\{\partial^{\varepsilon}_V(d) \mid d \in paths(\{T \in \mathbb{WSST}_e \mid \partial_V(T) = \bar{T}_1 \text{ or } \partial_V(T) = \bar{T}_2\})\}) =$
$prfxtree(\bigcup\{\partial^{\varepsilon}_V(d) \mid d \in paths(\{T \in \mathbb{WSST}_e \mid \partial_V(T) = \bar{T}_2\})\}) \iff$

[by ∂_V surjectivity]

$prfxtree(\bigcup\{\bar{d} \mid \bar{d} \in paths(\bar{T}_1) \text{ or } \bar{d} \in paths(\bar{T}_2)\}) =$
$prfxtree(\bigcup\{\bar{d} \mid \bar{d} \in paths(\bar{T}_2)\}) \iff$

$paths(\bar{T}_1) \subseteq paths(\bar{T}_2)$

Lemma 3.A.1 *Given g, b an embeddable pair, d_g, d_b small-step sequences for g and b respectively and σ a $\mathcal{C}$-linear substitution. If $d_g[x/d_b]_\sigma \vdash t \overset{\theta}{\Rightarrow}^* t'$ then, there exists $g \overset{\vartheta}{\Rightarrow}^* g' \in paths(d_g)$ and $b \overset{\eta}{\Rightarrow}^* b' \in paths(d_b)$ such that $\bar{h} = \sigma \uparrow \vartheta \uparrow \eta\{x/\tau(b')\}$ exists and is $\mathcal{V}_\varrho$-preserving.*

Moreover, $\sigma_{\bar{h}}\restriction_{var(t)} = \theta\restriction_t$ and $\tau(t') = \bar{h}[\tau(g')]$.

Proof.

By structural induction on the proof tree of

$$d_g[x/d_b]_\sigma \vdash t \overset{\theta}{\Rightarrow}^* t' \tag{1}$$

Base Case: The proof tree is a single application of axiom (2.2.7a). In this case $t \overset{\theta}{\Rightarrow}^* t'$ coincides with the zero step sequence t, that is $t = t'$ and $\theta = \varepsilon$. We will show that $g[x/b]_\sigma \vdash t$. $h\colon x\sigma \to b$ exists by (1), and $\sigma \in \mathcal{C}Substs$ by hypothesis, hence $h'\colon x\sigma \to \tau(b)$ exists. Hence, $\bar{h} = \sigma \uparrow \{x/\tau(b)\} = \sigma \uparrow \vartheta \uparrow \eta\{x/\tau(b')\}$ exists and is $\mathcal{V}_\varrho$-preserving. By (1), $t' = h[g]$, therefore $\tau(t') = \tau(h[g]) = h'[\tau(g)]$, moreover $h'[\tau(g)] = \bar{h}[\tau(g)]$. Hence $\tau(t') = \bar{h}[\tau(g)]$. It remains to show that $\theta\restriction_{var(t)} = \sigma_{\bar{h}}\restriction_{var(t)}$. Since $\theta = \varepsilon$ is it sufficient to show that $t\sigma_{\bar{h}} = t$. $t\sigma_{\bar{h}} = h[g]\sigma_{\bar{h}} = h[g](\sigma \uparrow \{x/\tau(b)\}) = h[g]\sigma = h[g] = t$.

Inductive Step: by cases on the rule which ends the proof tree

(2.2.7b): the proof tree for (1) is of the form

$$\dfrac{\overbrace{d_g \vdash^{\eta'} d'}^{(a)} \qquad \overbrace{d'[x/d_{b''}]_{\sigma\eta'} \vdash t'' \overset{\theta'}{\Rightarrow}{}^* t'}^{(b)}}{d_g[x/b \underset{q}{\overset{\eta'}{\Rightarrow}} d_{b''}]_\sigma \vdash t \underset{q}{\overset{\eta'}{\Rightarrow}} t'' \overset{\theta'}{\Rightarrow}{}^* t'}\ (2.2.7b)$$

By inductive hypothesis on (b) there exists $s \overset{\theta''}{\Rightarrow}{}^* s'$ and $b'' \overset{\eta''}{\Rightarrow}{}^* b'$ respectively prefix of d' and $d_{b''}$ such that $\bar{h}_1 = \sigma\eta' \uparrow \theta'' \uparrow \eta''\{x/\tau(b')\}$ exists and is $\mathcal{V}_\varrho$-preserving, $\sigma_{\bar{h}_1}\restriction_{var(t'')} = \theta'\restriction_{var(t'')}$ and $\tau(t') = \bar{h}_1[\tau(s')]$.

By inductive hypothesis on (b) there exists $g \overset{\vartheta}{\Rightarrow}{}^* g'$ prefix of d_g such that $\bar{h}_2 = \vartheta \uparrow \eta'$ exists and $\mathcal{V}_\varrho$-preserving (actually no bottom variable occurs in $\bar{h}_2$), $\sigma_{\bar{h}_2}\restriction_{var(s)} = \theta''\restriction_{var(s)}$ and $\tau(s') = \bar{h}_2[\tau(g')]$.

The following equalities hold

$$\begin{aligned}
\bar{h}_1 &= \\
&= \sigma\eta' \uparrow \theta'' \uparrow \eta''\{x/\tau(b')\} && [\text{by } dom(\sigma) \cap var(\eta') = \emptyset] \\
&= \sigma \uparrow \eta' \uparrow \theta'' \uparrow \eta''\{x/\tau(b')\} && [\text{by } \theta'' \preceq \vartheta \uparrow \eta' \text{ and } \eta' \uparrow \eta' = \eta'] \\
&= \sigma \uparrow \eta' \uparrow \vartheta \uparrow \eta''\{x/\tau(b')\} && [\text{by } dom(\eta') \cap var(\eta'') = \emptyset] \\
&= \sigma \uparrow \vartheta \uparrow \eta'\eta''\{x/\tau(b')\} && [\eta = \eta'\eta''] \\
&= \sigma \uparrow \vartheta \uparrow \eta\{x/\tau(b')\} \\
&= \bar{h}
\end{aligned}$$

Therefore $\bar{h}$ exists and is $\mathcal{V}_\varrho$-preserving. Moreover, $\tau(t') = \bar{h}_1[\tau(s')] = \bar{h}[\tau(s')] = \bar{h}[\bar{h}_2[\tau(g')]] = \bar{h}[\tau(g')]$ since $\bar{h} \circ \bar{h}_2 = \bar{h}$.

To show that $t\sigma_{\bar{h}} = t\eta'\theta'$ we first consider the case where $\eta' \neq \varepsilon$: in this case $t\eta' = t''$, thus $t'\eta'\theta' = t''\theta' = t''\sigma_{\bar{h}_1}$, but $\eta'\sigma_{\bar{h}_1} = \sigma_{\bar{h}_1} = \sigma_{\bar{h}}$, hence $t'\eta'\theta' = t\sigma_{\bar{h}}$. On the the other hand, if $\eta' = \varepsilon$, we just have to note that $t''\theta' = t''\sigma_{\bar{h}_1}$ which immediately implies that $t\eta'\theta' = t\theta' = t\sigma_{\bar{h}_1} = t\sigma_{\bar{h}_1}$ since $\theta'\restriction_{var(t)} = \theta'\restriction_{var(t)\cap var(t'')}$.

(2.2.7c): the proof tree for (1) is of the form

$$\dfrac{\overbrace{d_{g''}[x/d_b]_\sigma \vdash t'' \overset{\theta'}{\Rightarrow}{}^* t'}^{(a)}}{(g \underset{p}{\overset{\varepsilon}{\Rightarrow}} d_{g''})[x/d_b]_\sigma \vdash t \underset{p}{\overset{\varepsilon}{\Rightarrow}} t'' \overset{\theta'}{\Rightarrow}{}^* t'}\ (2.2.7c)$$

By inductive hypothesis on (a) there exist $g'' \overset{\vartheta'}{\Rightarrow}{}^* g'$ and $b \overset{\eta}{\Rightarrow}{}^* b'$ respectively prefix of $d_{g''}$ and d_b such that $\bar{h}_1 = \sigma \uparrow \vartheta' \uparrow \eta\{x/\tau(b')\}$ exists and is $\mathcal{V}_\varrho$-preserving, $\tau(t') = \bar{h}_1[\tau(g')]$ and $\sigma_{\bar{h}_1}\restriction_{var(t'')} = \theta'\restriction_{var(t'')}$.

$\bar{h} = \sigma\uparrow\vartheta\uparrow\eta\{x/\tau(b')\} = \sigma\uparrow\varepsilon\vartheta'\uparrow\eta\{x/\tau(b')\} = \bar{h}_1$, hence $\bar{h}$ exists and is $\mathcal{V}_\varrho$-preserving and $\tau(t') = \bar{h}[\tau(g')]$. Moreover $t\sigma_{\bar{h}} = t\theta$ since $\vartheta = \varepsilon\theta'$ and $\theta'\restriction_{var(t)} = \theta'\restriction_{var(t)\cap var(t'')}$.

(2.2.7d): the proof tree for (1) is of the form

$$\frac{\overbrace{d_g \vdash^{\eta'} d'}^{(a)} \qquad \overbrace{d'[x/d_{b''}]_{\sigma\eta'} \vdash t'' \overset{\theta'}{\Rightarrow}{}^* t'}^{(b)}}{d_g[x/b \underset{q}{\overset{\eta'}{\Rightarrow}} d_{b''}]_\sigma \vdash t \underset{p}{\overset{\eta'}{\Rightarrow}} t'' \overset{\theta'}{\Rightarrow}{}^* t'} \text{(2.2.7d)}$$

where $(g \underset{p}{\overset{\{y/c(\overrightarrow{z_n})\}}{\Longrightarrow}} d_{g''})$, $h\colon x\sigma \to b$ and $h[y]$ is rooted in q. The thesis can be proved to hold for this case analogously to that of (2.2.7b).

(2.2.7e): the proof tree for (1) is of the form

$$\frac{\overbrace{d_{g''}[x/d_b]_{\sigma\{y/c(\overrightarrow{z_n})\}} \vdash t'' \overset{\theta'}{\Rightarrow}{}^* t'}^{(a)}}{(g \underset{p}{\overset{\{y/c(\overrightarrow{z_n})\}}{\Longrightarrow}} d_{g''})[x/d_b]_\sigma \vdash t'' \overset{\theta'}{\Rightarrow}{}^* t'} \text{(2.2.7c)}$$

where $h\colon x\sigma \to b$ and $h[y]$ is $\mathcal{C}$-rooted.

By inductive hypothesis on (a) there exist $g'' \overset{\vartheta'}{\Rightarrow}{}^* g'$ and $b \overset{\eta}{\Rightarrow}{}^* b'$ respectively prefix of $d_{g''}$ and d_b such that $\bar{h}_1 = \sigma\{y/c(\overrightarrow{z_n})\}\!\uparrow\vartheta'\!\uparrow\eta\{x/\,\tau(b')\}$ exists and is $\mathcal{V}_\varrho$-preserving, $\tau(t') = \bar{h}_1[\tau(g')]$ and $\sigma_{\bar{h}_1}\!\restriction_{var(t'')} = \theta'\!\restriction_{var(t'')}$.

It is immediate to see that $\bar{h} = \sigma\!\uparrow\{y/c(\overrightarrow{z_n})\}\vartheta'\!\uparrow\eta\{x/\,\tau(b')\} = \bar{h}_1$ since, by hypothesis $\{y\} \cap var(\vartheta') = \emptyset$ and $dom(\sigma) \cap \{y, \overrightarrow{z_n}\} = \emptyset$. The thesis straightforwardly holds by inductive hypothesis and $\bar{h} = \bar{h}_1$.

(2.2.7f): the proof tree for (1) is of the form

$$\frac{\overbrace{d_b[w/c(\overrightarrow{z_n})]_\varepsilon \vdash d'}^{(a)} \qquad \overbrace{d_{g''}[x/d']_{\sigma\{y/c(\overrightarrow{x_n})\}} \vdash t'' \overset{\theta'}{\Rightarrow}{}^* t'}^{(b)}}{g \underset{p}{\overset{\{y/c(\overrightarrow{x_n})\}}{\Longrightarrow}} d_{g''}[x/d_b]_\sigma \vdash t \underset{p}{\overset{\{w/c(\overrightarrow{x_n})\}}{\Longrightarrow}} t'' \overset{\theta'}{\Rightarrow}{}^* t'} \text{(2.2.7b)}$$

where $h\colon x\sigma \to b$ and $h[y] = w \in \mathcal{V}$. By $\sigma \in \mathcal{C}Substs$ we have that if $y \neq w$ then w is a critical variable of b.

By inductive hypothesis on (b) there exists $g'' \overset{\vartheta'}{\Rightarrow}{}^* g'$ and $s \overset{\theta''}{\Rightarrow}{}^* s'$ respectively prefix of $d_{g''}$ and d' such that $\bar{h}_1 = \sigma\{y/c(\overrightarrow{z_n})\}\!\uparrow\vartheta'\!\uparrow\theta''\{x/\,\tau(s')\}$ exists and is $\mathcal{V}_\varrho$-preserving, $\tau(t') = \bar{h}_1[\tau(g')]$, and $\bar{h}_1\!\restriction_{var(t'')} = \theta'\!\restriction_{var(t'')}$.

By inductive hypothesis on (a) there exists $b \overset{\eta}{\Rightarrow}{}^* b'$ such that $\bar{h}_2 = \eta \uparrow \{w/c(\overrightarrow{z_n})\}$ exists, $\tau(s') = \bar{h}_2[\tau(b')]$ and, $\bar{h}_2\!\restriction_{var(s)} = \theta''\!\restriction_{var(s)}$.

The following equalities hold

$$\begin{aligned} \bar{h}_1 &= \sigma\{y/c(\overrightarrow{z_n})\} \uparrow \vartheta' \uparrow \theta''\{x/\,\tau(s')\} \\ &\quad [\,\theta'' \preceq \bar{h}_2\,] \end{aligned}$$

$$
\begin{aligned}
&= \sigma\{y/c(\overrightarrow{z_n})\} \uparrow \vartheta' \uparrow (\eta \uparrow \{w/c(\overrightarrow{z_n})\})\{x/\,\tau(s')\} \\
&\quad [\,\tau(s') = \bar{h}_2[b']\text{ and if } y \neq w \text{ then } w \text{ is critical variable of } b\,] \\
&= \sigma\{y/c(\overrightarrow{z_n})\} \uparrow \vartheta' \uparrow \eta \uparrow \{x/\,\tau(b')\} \\
&\quad [\,\text{by } dom(\sigma) \cap \{y, \overrightarrow{z_n}\} \text{ and } \{y\} \cap var(\vartheta')\,] \\
&= \sigma \uparrow \{y/c(\overrightarrow{z_n})\}\vartheta' \uparrow \eta \uparrow \{x/\,\tau(b')\} \\
&\quad [\,\vartheta = \{y/c(\overrightarrow{z_n})\}\vartheta'\,] \\
&= \sigma \uparrow \vartheta \uparrow \eta \uparrow \{x/\,\tau(b')\} \\
&= \bar{h}
\end{aligned}
$$

Therefore $\tau(t') = \bar{h}_2[\tau(g')] = \bar{h}[\tau(g')]$. Moreover, since $t\{w/c(\overrightarrow{z_n})\} = t''$ and $t''\theta' = t''\sigma_{\bar{h}_1}$ we have that $t\{w/c(\overrightarrow{z_n})\}\theta' = t\{w/c(\overrightarrow{z_n})\}\sigma_{\bar{h}_1}$ but we showed that $\{w/c(\overrightarrow{z_n})\}\sigma_{\bar{h}_1} = \sigma_{\bar{h}_1} = \sigma_{\bar{h}}$, thus $t\{w/c(\overrightarrow{z_n})\}\theta' = t\sigma_{\bar{h}}$.

Lemma 3.A.2 *Given g, b an embeddable pair, $V_g = var(g)$, $V_b = var(b)$, $x \in V_g \setminus V_b$ and $V = var(g\{x/b\})$. For all $G, G' \in \mathbb{WSST}_g$ s.t. $\partial_{V_g}(G) = \partial_{V_g}(G')$ and for all $B, B' \in \mathbb{WSST}_b$ s.t. $\partial_{V_b}(B) = \partial_{V_b}(B')$, $\partial_V(G[x/B]) \preccurlyeq \partial_V(G'[x/B'])$.*

Proof.

In the following, to keep the notation clear, given a sequence of substitutions $\sigma_1, \ldots, \sigma_n$ we will write $\overrightarrow{\sigma_0}$ and $\overrightarrow{\sigma_{j+1}}$ respectively for ε and $\overrightarrow{\sigma_j}\sigma_{j+1}$ for every $0 \leq j \leq n$.

We will prove a stronger result: for any $d_g \in G$, $d_b \in B$ and d canonical small-step sequences such that $d_g[x/d_b]_\varepsilon \vdash d$, then there exist $d'_g \in G'$, $d'_b \in B'$ and d' s.t. d' is canonical, $d'_g[x/d'_b]_\varepsilon \vdash d'$, $\partial^\varepsilon_V(d_g) = \partial^\varepsilon_V(d'_g)$ and $\partial^\varepsilon_V(d) = \partial^\varepsilon_V(d')$.

Suppose that $d = t_0 \xrightarrow[p_1]{\sigma_1} \ldots \xrightarrow[p_n]{\sigma_n} t_n$ for some $n \geq 0$. We proceed by induction on n. The case $n = 0$ is immediate.

For $n > 0$, by Lemma 3.A.1 exist $g_0 \overset{\vartheta_1}{\Longrightarrow}{}^* \ldots \overset{\vartheta_n}{\Longrightarrow}{}^* g_n$ and $b_0 \overset{\eta_1}{\Longrightarrow}{}^* \ldots \overset{\eta_n}{\Longrightarrow}{}^* b_n$, respectively prefix of d_g and d_b, s.t. for any $0 \leq i \leq n$, exists $\bar{h}_i = \overrightarrow{\vartheta_i} \uparrow \overrightarrow{\eta_i}\{x/\,\tau(b_i)\}$ which is $\mathcal{V}_\varrho$-preserving, $\tau(t_i) = \bar{h}_i[\tau(g_i)]$ and $\overrightarrow{\sigma_i}\restriction_V = \sigma_{\bar{h}_i}\restriction_V$. The inductive hypothesis states that there exist $d''_g \in G'$, $d''_b \in B'$ and d'' s.t. d'' is canonical and

$$
\begin{aligned}
&d''_g[x/d''_b]_\varepsilon \vdash d'' \\
&\partial^\varepsilon_{V_g}(d''_g) = \partial^\varepsilon_{V_g}(g_0 \overset{\vartheta_1}{\Longrightarrow}{}^* \ldots \overset{\vartheta_{n-1}}{\Longrightarrow}{}^* g_{n-1}) \\
&\partial^\varepsilon_V(d'') = \partial^\varepsilon_V(t_0 \xrightarrow[p_1]{\sigma_1} \ldots \xrightarrow[p_{n-1}]{\sigma_{n-1}} t_{n-1})
\end{aligned}
$$

By Lemma 3.A.1, $d''_g = g'_0 \overset{\vartheta'_1}{\Longrightarrow}{}^* \ldots \overset{\vartheta'_{n-1}}{\Longrightarrow}{}^* g'_{n-1}$ and $d''_b = b'_0 \overset{\eta'_1}{\Longrightarrow}{}^* \ldots \overset{\eta'_{n-1}}{\Longrightarrow}{}^* b'_{n-1}$ where it exists $\bar{h}'_i = \overrightarrow{\vartheta'_i} \uparrow \overrightarrow{\eta'_i}\{x/\,\tau(b'_i)\}$ $\mathcal{V}_\varrho$-preserving, $\tau(t_i) = \bar{h}'_i[\tau(g'_i)]$ and $\overrightarrow{\sigma_i}\restriction_V = \sigma_{\bar{h}'_i}\restriction_V$ for $0 \leq i \leq n-1$. Therefore, we have that $\vartheta_i = \vartheta'_i$, $\tau(g_i) = \tau(g'_i)$, $\eta_i = \eta'_i$ and $\tau(b_i) = \tau(b'_i)$ for every $0 \leq i \leq n-1$.

By Equation (3.1.3) $\partial^\varepsilon_V(d)$ is of the the form $a_0 \xrightarrow{\varrho_1} \ldots \xrightarrow{\varrho_n} a_n$ where $a_i = \overrightarrow{\sigma_i}\restriction_V \cdot \tau(t_i)$ for $0 \leq i \leq n$. Thus $a_i = \sigma_{\bar{h}_i}\restriction_V \cdot \bar{h}_i[\tau(g_i)]$ for $0 \leq i \leq n$. Moreover $a_i = \sigma_{\bar{h}'_i}\restriction_V \cdot \bar{h}'_i[\tau(g'_i)]$ for $0 \leq i \leq n-1$. By construction $a_{n-1} \twoheadrightarrow_{\varrho_n} a_n$, thus there are two possible cases:

$\boldsymbol{\varrho_n \in var(\tau(b_{n-1}))}$) then $b_{n-1} \overset{\eta_n}{\Longrightarrow}{}^* b_n$ is of the form $b_{n-1} \overset{\eta_n}{\underset{p_n}{\hookrightarrow}} b_n$, $g_{n-1} = g_n$ and $\vartheta_n = \varepsilon$. In fact, the steps $t_{n-1} \overset{\sigma_n}{\underset{p_n}{\hookrightarrow}} t_n$ come form a sequence of applications of rule (2.2.7b). By $\partial_{V_b}(B) = \partial_{V_b}(B')$ it exists a derivation $\tilde{d}_b \in B'$ s.t. $\partial^{\varepsilon}_{V_b}(\tilde{d}_b) = \partial^{\varepsilon}_{V_b}(d_b)$. Therefore, by $B' \in \mathbb{WSST}_b$ and Proposition 2.2.4, there exists $d'_b \in B'$ of the form $b'_0 \overset{\eta'_1}{\Longrightarrow}{}^* \ \ldots \ \overset{\eta'_{n-1}}{\Longrightarrow}{}^* b'_{n-1} \overset{\eta_n}{\underset{p_n}{\hookrightarrow}} b'_n$ where $\tau(b'_n) = \tau(b_n)$ and $\tau(b'_{n-1})|_{p_n} = \varrho_n$. Thus, choosing $d'_g = d''_g$, there exists d' s.t. $\partial^{\varepsilon}_V(d) = \partial^{\varepsilon}_V(d')$, for which $d'_g[x/d'_b]_{\varepsilon} \vdash d'$ holds. In fact, its proof tree can be obtained extending that of $d''_g[x/d''_b]_{\varepsilon} \vdash d''$ with a sequence of application of rule (2.2.7b).

$\boldsymbol{\varrho_n \in var(\tau(g_{n-1}))}$) then $g_{n-1} \overset{\vartheta_n}{\Longrightarrow}{}^* g_n$ is of the form $g_{n-1} \overset{\vartheta_n}{\underset{p}{\hookrightarrow}} g_n$ where $\vartheta_n = \vartheta_n^1 \cdots \vartheta_n^m$ and ϑ_n^i, for $1 \leq i \leq m$, are the labels of the small-steps sequence composing the former "big-step".

By $\partial_{V_g}(G) = \partial_{V_g}(G')$, it exists $d'_g \in G'$ of the form $g'_0 \overset{\vartheta'_1}{\Longrightarrow}{}^* \ \ldots \ \overset{\vartheta'_{n-1}}{\Longrightarrow}{}^* g'_{n-1} \overset{\vartheta'_n}{\underset{q}{\hookrightarrow}} g'_n$ such that $\partial_{V_g}(d'_g) = \partial_{V_g}(d_g)$. By construction $\vartheta'_n = \vartheta_n$, but the labels of the small-steps sequence composing the former "big-step" can differs from those composing ϑ_n. Let ${\vartheta'}_n^1 \cdots {\vartheta'}_n^k$ be the decomposition of ϑ'_n.

In fact, exists $\bar{h} = \overrightarrow{\vartheta_n} \uparrow \overrightarrow{\eta_{n-1}}\{x/\tau(b_{n-1})\}$, since there exists $\bar{h}_i = \overrightarrow{\vartheta_n} \uparrow \overrightarrow{\eta_n}\{x/\tau(b_n)\}$, hence there are two possible sub-cases:

- $\boldsymbol{\bar{h}}$ **is** $\boldsymbol{\mathcal{V}_\varrho}$**-preserving**) if $\bar{h}[\tau(g_n)]|_{p_n} = \varrho \in \mathcal{V}_\varrho$, then $\tau(g_n)|_{p_n} \in \mathcal{V}$ and $\varrho \in var(\tau(b_{n-1}))$, that is $b_{n-1} \overset{\eta_n}{\Longrightarrow}{}^* b_n$ is of the form $b_{n-1} \overset{\eta_n}{\underset{q}{\hookrightarrow}} b_n$ where $\tau(b_{n-1})|_q = \varrho$. In fact, the steps of $t_{n-1} \overset{\sigma_n}{\underset{p_n}{\hookrightarrow}} t_n$ come form a sequence of m applications of rules selected within (2.2.7c),(2.2.7e) and (2.2.7f), followed by a sequence of applications of rule (2.2.7b).

  For the same reason of the former case, there exists $d'_b \in B'$ of the form $b'_0 \overset{\eta'_1}{\Longrightarrow}{}^* \ \ldots \ \overset{\eta'_{n-1}}{\Longrightarrow}{}^* b'_{n-1} \overset{\eta_n}{\underset{q}{\hookrightarrow}} b'_n$ where $\tau(b'_n) = \tau(b_n)$ and $\tau(b'_{n-1})|_q = \varrho$. It is straightforward to see that there exists d' s.t. $\partial^{\varepsilon}_V(d) = \partial^{\varepsilon}_V(d')$, for which $d'_g[x/d'_b]_{\varepsilon} \vdash d'$ holds. In fact, its proof tree can be obtained extending that of $d''_g[x/d''_b]_{\varepsilon} \vdash d''$ with a sequence of k applications of rules selected within (2.2.7c),(2.2.7e) and (2.2.7f), followed by a sequence of applications of rule (2.2.7b).

  On the other hand, if $\bar{h}[\tau(g_n)]|_{p_n} = \varrho \notin \mathcal{V}_\varrho$ then $\tau(g_n)|_{p_n} \notin \mathcal{V}$. Thus $b_{n-1} = b_n$ and $\eta_n = \varepsilon$. Indeed, the steps of $t_{n-1} \overset{\sigma_n}{\underset{p_n}{\hookrightarrow}} t_n$ come from a sequence of applications of rules selected within (2.2.7c),(2.2.7e) and (2.2.7f). Thus, choosing $d'_b = d''_b$, it is straightforward to see that there exists d' s.t. $\partial^{\varepsilon}_V(d) = \partial^{\varepsilon}_V(d')$, for which $d'_g[x/d'_b]_{\varepsilon} \vdash d'$ holds. In fact, its proof tree can be obtained extending that of $d''_g[x/d''_b]_{\varepsilon} \vdash d''$ with a sequence of k applications of rules selected within (2.2.7c),(2.2.7e) and (2.2.7f).
- $\boldsymbol{\bar{h}}$ **isn't** $\boldsymbol{\mathcal{V}_\varrho}$**-preserving**) $\bar{h} = \overrightarrow{\vartheta_{n-1}}\vartheta'_n \uparrow \overrightarrow{\eta_{n-1}}\{x/\tau(b'_{n-1})\}$, since $\vartheta'_n = \vartheta_n$ and $\tau(b_{n-1}) = \tau(b'_{n-1})$. Let $1 \leq j \leq k$ be the first index for which $\overrightarrow{\vartheta_{n-1}}{\vartheta'}_n^1 \cdots {\vartheta'}_n^j \uparrow \overrightarrow{\eta_{n-1}}\{x/\tau(b'_{n-1})\}$

is not $\mathcal{V}_\varrho$-preserving. By construction, it exists a unique $\varrho_1 \in var(\tau(b_{n-1})) \cap \mathcal{V}_\varrho$ s.t. $h[\varrho_1] \neq \varrho_1$. Actually $h[\varrho_1]$ is $\mathcal{C}$-rooted.
By $\partial_{V_b}(B) = \partial_{V_b}(B')$, $B' \in \mathbb{WSST}_b$, and Proposition 2.2.4, there exists $d'^1_b \in B'$ of the form $b'_0 \overset{\eta'_1}{\Rightarrow}{}^* \ldots \overset{\eta'_{n-1}}{\Longrightarrow}{}^* b'_{n-1} \overset{\eta^1_n}{\underset{q_1}{\hookrightarrow}} b^1_{n-1} \overset{\theta^1}{\Rightarrow}{}^* b'_n$ s.t. $\eta^1_n\theta^1 = \eta_n$. Moreover $\overrightarrow{\vartheta_{n-1}}\vartheta'^1_n \cdots \vartheta'^j_n \uparrow \overrightarrow{\eta_{n-1}}\eta^1_n\{x/\tau(b^1_{n-1})\}$ is $\mathcal{V}_\varrho$-preserving.
By repeated application of this construction we eventually obtain a derivation $d'^l_b \in B'$ of the form

$$b'_0 \overset{\eta'_1}{\Rightarrow}{}^* \ldots \overset{\eta'_{n-1}}{\Longrightarrow}{}^* b'_{n-1} \overset{\eta^1_n}{\underset{q_1}{\hookrightarrow}} \ldots \overset{\eta^l_n}{\underset{q_l}{\hookrightarrow}} b^l_{n-1} \overset{\theta^l}{\Rightarrow}{}^* b'_n,$$

for some $1 \leq l \leq k$ s.t. $\overrightarrow{\vartheta_{n-1}}\vartheta'_n \uparrow \overrightarrow{\eta_{n-1}}\eta^1_n \cdots \eta^1_n\theta^l\{x/\tau(b^1_{n-1})\}$ is $\mathcal{V}_\varrho$-preserving, thus holding the the hypothesis of the former sub-case. Therefore it is straightforward to see that there exists $d'_b \in B'$ and d' s.t. $\partial^\varepsilon_V(d) = \partial^\varepsilon_V(d')$, for which $d'_g[x/d'_b]_\varepsilon \vdash d'$ holds. Actually, its proof tree can be obtained extending that of $d''_g[x/d''_b]_\varepsilon \vdash d''$ with a sequence of rule applications obtained alternating l times

- a sequence of applications of rules selected within (2.2.7c),(2.2.7e) and (2.2.7f), and
- a sequence of applications of rule (2.2.7d). Then ending the proof as described in the former sub-case.

Corollary 3.A.3 *Given g, b an embeddable pair, $V_g = var(g)$, $V_b = var(b)$, $x \in V_g \setminus V_b$ and $V = var(g\{x/b\})$. For all $G \in \mathbb{WSST}_g$ and $B \in \mathbb{WSST}_b$*

$$\partial_V(G[x/B]) \overset{(1)}{=} \partial_V(\partial^\gamma_g \circ \partial_{V_g}(G)[x/\,\partial^\gamma_b \circ \partial_{V_b}(B)]) \overset{(2)}{=} \partial_{V_g}(G)[x/\,\partial_{V_b}(B)]_V$$

Proof.

(1) Since $\partial_{var(e)} \circ \partial^\gamma_e \circ \partial_{var(e)} = \partial_{var(e)}$ for all $e \in \mathcal{T}(\Sigma, \mathcal{V})$ and Lemma 3.A.2.

(2) Noting that the each rule in (3.1.21) corresponds to one of the cases listed in the proof of Lemma 3.A.2.

Proof of Theorem 3.1.13.
We proceed by induction on the cardinality of $baseknots(e)$

Base Case: There are two possible cases for e:

$\boldsymbol{e = x \in \mathcal{V}}$) the following equalities hold

$$\begin{aligned}
&\partial_{\{x\}}(\mathcal{E}[\![x]\!]_\mathcal{I}) = && [\text{by Equation (2.2.5a)}]\\
&= \partial_{\{x\}}(x) && [\text{by Equation (3.1.4)}]\\
&= \varepsilon \cdot x && [\text{by Equation (3.1.19a)}]\\
&= \mathcal{E}^\partial[\![x]\!]_{\partial(\mathcal{I})}
\end{aligned}$$

$e = \varphi(\overrightarrow{x_n})$) consider $\overrightarrow{y_n} \in \mathcal{V}$ fresh distinct, and $V_i := \{x_1, \ldots, x_i, y_{i+1}, \ldots, y_n\}$ for $i \in \{1, \ldots, n\}$, then the following equalities hold

$$
\begin{aligned}
&\partial_{\{\overrightarrow{x_n}\}}(\mathcal{E}[\![\varphi(\overrightarrow{x_n})]\!]_{\mathcal{I}}) = \\
&\qquad [\text{by Equation (2.2.5b)}] \\
&= \partial_{\{\overrightarrow{x_n}\}}(\mathcal{I}(\varphi(\overrightarrow{y_n}))[y_1/x_1]\ldots[y_n/x_n]) \\
&\qquad [\text{by straightforward induction on } n \text{ using Corollary 3.A.3}] \\
&= \partial_{\{\overrightarrow{y_n}\}}(\mathcal{I}(\varphi(\overrightarrow{y_n})))[y_1/\,\partial_{\{x_1\}}(x_1)]_{V_1}\ldots[y_n/\,\partial_{\{x_n\}}(x_n)]_{V_n} \\
&\qquad [\text{by Equation (3.1.4)}] \\
&= \partial_{\{\overrightarrow{y_n}\}}(\mathcal{I}(\varphi(\overrightarrow{y_n})))[y_1/\varepsilon \cdot x_1]_{V_1}\ldots[y_n/\varepsilon \cdot x_n]_{V_n} \\
&\qquad [\text{by Equation (3.1.5)}] \\
&= \partial(\mathcal{I})(\varphi(\overrightarrow{y_n}))[y_1/\varepsilon \cdot x_1]_{V}\ldots[y_n/\varepsilon \cdot x_n]_{V} \\
&\qquad [\text{by Equation (3.1.19b)}] \\
&= \mathcal{E}^{\partial}[\![\varphi(\overrightarrow{x_n})]\!]_{\partial(\mathcal{I})}
\end{aligned}
$$

Inductive Step: let p be the leftmost node in $baseknots(e)$, and y be a fresh data variable. We have that

$$
\begin{aligned}
&\partial_{var(e)}(\mathcal{E}[\![e]\!]_{\mathcal{I}}) = \\
&\qquad [\text{by Equation (2.2.5c), with } y \text{ fresh variable}] \\
&= \partial_{var(e)}(\mathcal{E}[\![e[y]_p]\!]_{\mathcal{I}}[y/\,\mathcal{E}[\![e|_p]\!]_{\mathcal{I}}]) \\
&\qquad [\text{by Corollary 3.A.3}] \\
&= \partial_{var(e[y]_p)}(\mathcal{E}[\![e[y]_p]\!]_{\mathcal{I}})[y/\,\partial_{var(e|_p)}(\mathcal{E}[\![e|_p]\!]_{\mathcal{I}})]_{var(e)} \\
&\qquad [\text{by inductive hypothesis}] \\
&= \mathcal{E}^{\partial}[\![e[y]_p]\!]_{\partial(\mathcal{I})}[y/\,\mathcal{E}^{\partial}[\![e|_p]\!]_{\partial(\mathcal{I})}]_{var(e)} \\
&\qquad [\text{by Equation (3.1.19c)}] \\
&= \mathcal{E}^{\partial}[\![e]\!]_{\partial(\mathcal{I})}
\end{aligned}
$$

Proof of Corollary 3.1.14.

Point 1) Given an op-rooted term e and $T \in \mathbb{WSST}_e$ of the form $e \overset{\sigma}{\Rightarrow}{}^{*}\ e\sigma \underset{p}{\overset{\varepsilon}{\Rightarrow}} T'$ where the root term of T' is r, we have that

$$
\begin{aligned}
&\partial_{var(e)}(T) = \\
&\qquad [\text{by Equations (3.1.3) and (3.1.4)}] \\
&= \begin{cases} zaproot(\curlyvee\left\{\partial^{\sigma}_{var(e)}(d) \;\middle|\; d \in paths(T')\right\}) & \text{if } r \text{ is } \mathcal{D}\text{-rooted} \\ \varepsilon \cdot \varrho \xrightarrow{\varrho} \curlyvee\left\{\partial^{\sigma}_{var(e)}(d) \;\middle|\; d \in paths(T')\right\} & \text{otherwise} \end{cases} \\
&\qquad [\text{by Equation (3.1.17)}] \\
&= \begin{cases} zaproot(\sigma\lfloor\partial_{var(r)}(T')\rfloor_{var(e)}) & \text{if } r \text{ is } \mathcal{D}\text{-rooted} \\ \varepsilon \cdot \varrho \xrightarrow{\varrho} \sigma\lfloor\partial_{var(r)}(T')\rfloor_{var(e)} & \text{otherwise} \end{cases}
\end{aligned}
\tag{1}
$$

Now we prove that $\mathcal{P}^{\partial}[\![P]\!] \circ \partial = \partial \circ \mathcal{P}[\![P]\!]$. Given $\mathcal{I} \in \mathbb{I}$, $f(\overrightarrow{x_n}) \in \mathbb{MGC}$, $V = \{\overrightarrow{x_n}\}$ and T_f a fresh variant of the definitional tree for f in P, for $pat(T_f) = f(\overrightarrow{x_n})$, the following equalities hold

$$
\begin{aligned}
&\partial(\mathcal{P}[\![P]\!]_{\mathcal{I}})(f(\overrightarrow{x_n})) = \\
&\qquad [\text{by Equation (3.1.5)}] \\
&= \partial_V(\mathcal{P}[\![P]\!]_{\mathcal{I}}(f(\overrightarrow{x_n}))) \\
&\qquad [\text{by Equation (2.2.9)}] \\
&= \partial_V(\xi[\![T_f]\!]_{\mathcal{I}}) \\
&\qquad [\text{by Definition 2.2.10}] \\
&= \partial_V(\bigsqcup\left\{f(\overrightarrow{x_n}) \overset{\sigma}{\Rightarrow}^* \mathcal{E}[\![r]\!]_{\mathcal{I}} \;\middle|\; rule(f(\overrightarrow{t_n}) \to r) \in T_f,\, \sigma = \{\overrightarrow{x_n}/\overrightarrow{t_n}\}\right\}) \\
&\qquad [\text{by Definition 1.4.1}] \\
&= \partial_V(\bigsqcup\left\{f(\overrightarrow{x_n}) \overset{\sigma}{\Rightarrow}^* \mathcal{E}[\![r]\!]_{\mathcal{I}} \;\middle|\; f(\overrightarrow{t_n}) \to r \ll P,\, \sigma = \{\overrightarrow{x_n}/\overrightarrow{t_n}\}\right\}) \\
&\qquad [\text{by } \partial \circ \bigsqcup = \bigvee \circ \partial] \\
&= \bigvee\left\{\partial_V(f(\overrightarrow{x_n}) \overset{\sigma}{\Rightarrow}^* \mathcal{E}[\![r]\!]_{\mathcal{I}}) \;\middle|\; f(\overrightarrow{t_n}) \to r \ll P,\, \sigma = \{\overrightarrow{x_n}/\overrightarrow{t_n}\}\right\} \\
&\qquad [\text{by Equation (3.1.16), (1) and Theorem 3.1.13}] \\
&= \bigvee\{\xi^{\partial}_{\{\overrightarrow{x_n}\}}[\![r]\!]^{\{\overrightarrow{x_n}/\overrightarrow{t_n}\}}_{\partial(\mathcal{I})} \mid f(\overrightarrow{t_n}) \to r \ll P\} \\
&\qquad [\text{by Equation (3.1.15)}] \\
&= (\mathcal{P}^{\partial}[\![P]\!]_{\partial(\mathcal{I})})(f(\overrightarrow{x_n}))
\end{aligned}
$$

Point 2) Consider any $\{A_i\} \subseteq \mathbb{I}_{\mathbb{ERT}}$. Since ∂ is surjective

$$\exists C_i \in \mathbb{I}.\, A_i = \partial(C_i) \tag{2}$$

Then

$$
\begin{aligned}
&\bigvee\{\mathcal{P}^{\partial}[\![P]\!]_{A_i}\} = && [\text{by (2)}] \\
&\bigvee\{\mathcal{P}^{\partial}[\![P]\!]_{\partial(C_i)}\} = && [\text{by Point 1 of Corollary 3.1.14}] \\
&\bigvee\{\partial(\mathcal{P}[\![P]\!]_{C_i})\} = && [\text{since } \partial \circ \bigsqcup = \bigvee \circ \partial] \\
&\partial(\bigsqcup\{\mathcal{P}[\![P]\!]_{C_i}\}) = && [\text{since } \mathcal{P}[\![P]\!] \text{ is continuous}] \\
&\partial(\mathcal{P}[\![P]\!]_{\bigsqcup\{C_i\}}) = && [\text{by Point 1 of Corollary 3.1.14}] \\
&\mathcal{P}^{\partial}[\![P]\!]_{\partial(\bigsqcup\{C_i\})} = && [\text{again since } \partial \circ \bigsqcup = \bigvee \circ \partial] \\
&\mathcal{P}^{\partial}[\![P]\!]_{\bigvee\{\partial(C_i)\}} = && [\text{by (2)}] \\
&\mathcal{P}^{\partial}[\![P]\!]_{\bigvee\{A_i\}}
\end{aligned}
$$

that is, $\mathcal{P}^{\partial}[\![P]\!]$ is continuous.

Point 3) $\mathcal{F}^{\partial}[\![P]\!] = \mathcal{P}^{\partial}[\![P]\!]\uparrow\omega$ is an immediate consequence of Point 2.

Point 4) The following equalities hold

$$\begin{aligned}
&\mathcal{F}^{\partial}[\![P]\!] = && [\text{by Point 3}]\\
&= \mathcal{P}^{\partial}[\![P]\!]{\uparrow}\omega && [\text{repeatedly applying Point 1}]\\
&= \partial(\mathcal{P}[\![P]\!]{\uparrow}\omega) && [\text{by Equation (2.2.11)}]\\
&= \partial(\mathcal{F}[\![P]\!])
\end{aligned}$$

Proof of Corollary 3.1.15.
First observe that for all $e \in \mathcal{T}(\Sigma, \mathcal{V})$, the following hold

$$\begin{aligned}
&\partial_{var(e)}(\mathcal{N}[\![e \textit{ in } P]\!]) = && [\text{by Theorems 2.2.19 and 2.2.25}]\\
&\partial_{var(e)}(\mathcal{E}[\![e]\!]_{\mathcal{F}[\![P]\!]}) = && [\text{by Theorem 3.1.13}]\\
&\mathcal{E}^{\partial}[\![e]\!]_{\partial(\mathcal{F}[\![P]\!])} = && [\text{by Point 4 of Corollary 3.1.14}]\\
&\mathcal{E}^{\partial}[\![e]\!]_{\mathcal{F}^{\partial}[\![P]\!]}.
\end{aligned}$$

This and Corollary 2.2.26 implies that, $\mathcal{F}^{\partial}[\![P_1]\!] = \mathcal{F}^{\partial}[\![P_2]\!]$ if and only if

$$\forall e \in \mathcal{T}(\Sigma, \mathcal{V}).\ \partial_{var(e)}(\mathcal{N}[\![e \textit{ in } P_1]\!]) = \partial_{var(e)}(\mathcal{N}[\![e \textit{ in } P_2]\!]) \tag{1}$$

By Equation (3.1.3), for any canonical small-step derivation d of the form $e_0 \xrightarrow[p_n]{\vartheta_1} \dots \xrightarrow[p_n]{\vartheta_n} e_n$, the last node of $\partial^{\varepsilon}_{var(e_0)}(d)$ is $(\vartheta_1 \cdots \vartheta_n){\restriction_{e_0}} \cdot \tau(e_n)$. In particular, by Equation (3.1.2), if $e_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ then $\tau(e_n) = e_n$.

Thus, by Definitions 3.1.1 and 2.2.12, we conclude that (1) implies $P_1 \approx_{cr} P_2$.

Proof of Theorem 3.1.18.
Let $Q \in \mathbb{UP}^{\Sigma'}_{\Sigma}$ for some Σ', $H_i(\mathcal{I}^{\partial}) := \mathcal{P}^{\partial}[\![Q]\!]_{\mathcal{F}^{\partial}[\![P_i]\!] \curlyvee \mathcal{I}^{\partial}}$, $\mathcal{X}_i := \mathit{lfp}(H_i)$ and $\mathcal{S}_i := \mathcal{X}_i \curlyvee \mathcal{F}^{\partial}[\![P_i]\!]$. Then

$$\begin{aligned}
&\mathcal{P}^{\partial}[\![Q \cup P_i]\!]_{\mathcal{S}_i} = && [\text{by Equation (3.1.15) and } \mathcal{S}_i \text{ definition}]\\
&\mathcal{P}^{\partial}[\![Q]\!]_{\mathcal{X}_i \curlyvee \mathcal{F}^{\partial}[\![P_i]\!]} \curlyvee \mathcal{P}^{\partial}[\![P_i]\!]_{\mathcal{X}_i \curlyvee \mathcal{F}^{\partial}[\![P_i]\!]} = && [\text{by } H_i \text{ definition}]\\
&H_i(\mathcal{X}_i) \curlyvee \mathcal{P}^{\partial}[\![P_i]\!]_{\mathcal{X}_i \curlyvee \mathcal{F}^{\partial}[\![P_i]\!]} = && [\text{since } \mathcal{P}^{\partial}[\![P_i]\!]_{\mathcal{X}_i \curlyvee \mathcal{F}^{\partial}[\![P_i]\!]} \text{ does not use } \mathcal{X}_i]\\
&H_i(\mathcal{X}_i) \curlyvee \mathcal{P}^{\partial}[\![P_i]\!]_{\mathcal{F}^{\partial}[\![P_i]\!]} = && [\text{since } \mathcal{X}_i \text{ and } \mathcal{F}^{\partial}[\![P_i]\!] \text{ are fixpoints}]\\
&\mathcal{X}_i \curlyvee \mathcal{F}^{\partial}[\![P_i]\!] = && [\text{by } \mathcal{S}_i \text{ definition}]\\
&\mathcal{S}_i
\end{aligned}$$

Thus $\mathcal{S}_i$ is a fixpoint. It has to be the least fixpoint, otherwise either $\mathcal{X}_i$ or $\mathcal{F}^{\partial}[\![P_i]\!]$ should not be the least fixpoint of H_i or $\mathcal{P}^{\partial}[\![P_i]\!]$, which is absurd.

Thus $\mathcal{F}^{\partial}[\![Q \cup P_i]\!] = \mathcal{X}_i \curlyvee \mathcal{F}^{\partial}[\![P_i]\!]$, where $\mathcal{X}_i$ depends uniquely on Q and $\mathcal{F}^{\partial}[\![P_i]\!]$. Hence $\mathcal{F}^{\partial}[\![P_1]\!] = \mathcal{F}^{\partial}[\![P_2]\!]$ if and only if $\mathcal{F}^{\partial}[\![Q \cup P_1]\!] = \mathcal{F}^{\partial}[\![Q \cup P_2]\!]$.

The thesis follows then by Definition 3.1.17 and Corollary 3.1.15.

Proof of Proposition 3.1.29.
By construction $(\mathbb{ARS}_{\overrightarrow{x_n}}, \subseteq) \underset{fold_{\{\overrightarrow{x_n}\}}}{\overset{unfold_{\{\overrightarrow{x_n}\}}}{\twoheadleftarrow\!\!\!\!\twoheadrightarrow}} (\mathbb{WERS}_{\overrightarrow{x_n}}, \hat{\preccurlyeq})$, thus it suffices to prove that $\forall \tilde{S} \in \mathbb{ARS}_{\overrightarrow{x_n}}.\ unfold_{\{\overrightarrow{x_n}\}} \circ fold_{\{\overrightarrow{x_n}\}}(\tilde{S}) = \tilde{S}$.

$$
\begin{aligned}
&unfold_{\{\overrightarrow{x_n}\}}(fold_{\{\overrightarrow{x_n}\}}(\tilde{S})) = \\
&\qquad [\text{by Equation (3.1.41)}] \\
&= unfold_{\{\overrightarrow{x_n}\}}(\{\sigma \centerdot s_1 \text{–} s_2 \,|\, \text{maximal } unfold_{\{\overrightarrow{x_n}\}}(\sigma \centerdot s_1 \text{–} s_2) \subseteq \tilde{S}\}) \\
&\qquad [\text{by Equation (3.1.42)}] \\
&= \bigcup \{unfold_{\{\overrightarrow{x_n}\}}(\sigma \centerdot s_1 \text{–} s_2) \,|\, \text{maximal } unfold_{\{\overrightarrow{x_n}\}}(\sigma \centerdot s_1 \text{–} s_2) \subseteq \tilde{S}\} \\
&= \tilde{S}
\end{aligned}
$$

In order to prove the precision w.r.t. the ζ_V abstraction of the embedding operator of Equation (3.1.53) (formally stated by Corollary 3.A.8) we proceed as follows:

- we define an auxiliary embedding operator over $\mathbb{ARS}$ (see Equation (3.A.1)),
- then in Lemma 3.A.6 we prove its precision w.r.t. the $\tilde{\zeta_V}$ abstraction,
- finally, in Lemma 3.A.7 we prove the operator of Equation (3.A.1) is equal (up to isomorphism of Proposition 3.1.29) to that of Equation (3.1.53).

Given g, b an embeddable pair, x a variable in $var(g) \setminus var(b)$, $G \in \mathbb{ARS}_g$, and $B \in \mathbb{ARS}_b$, the embedding of B into G w.r.t. x is defined by

$$
G[x/B]_V := \left\{ \sigma\restriction_V \cdot s \;\middle|\; \begin{array}{l} \vartheta \cdot g \in G,\ \eta \cdot b \in B, \\ \sigma = \vartheta \uparrow \eta\{x/b\}\ \mathcal{V}_\varrho\text{-preserving}, \\ \forall y \in \mathcal{V} \cap var(g).\ y\sigma \notin \mathcal{V}_\varrho, \\ b|_p \notin \mathcal{V}_\varrho \Rightarrow (x\vartheta)|_p \notin var(x\vartheta) \setminus var(g) \\ s' \in base(g, \sigma),\ s' \lesssim s \lesssim \psi(g\sigma) \end{array} \right\} \tag{3.A.1}
$$

Lemma 3.A.4 *Given a τ-term $\bar{g}$ and a substitution σ,*

$$
\{t \,|\, t \lesssim \psi(\bar{g}\sigma)\} = \{t \,|\, g \lesssim \psi(\bar{g}),\ s' \in base(g, \sigma),\ s' \lesssim t \lesssim \psi(g\sigma)\}
$$

Proof.

$\supseteq$) immediate by definition of *base*.

$\subseteq$) Let $t \lesssim \psi(\bar{g}\sigma)$. There are two possible cases: (1) $t \lesssim \psi(\bar{g})$; (2) $t \not\lesssim \psi(\bar{g})$. If (1) holds then $t \in \{t \,|\, g \lesssim \psi(\bar{g})\}$, therefore, since $\{t \,|\, g \lesssim \psi(\bar{g})\} \subseteq \{t \,|\, g \lesssim \psi(\bar{g}),\ s' \in base(g, \sigma),\ s' \lesssim t \lesssim \psi(g\sigma)\}$, we have that $t \in \{t \,|\, g \lesssim \psi(\bar{g}),\ s' \in base(g, \sigma),\ s' \lesssim t \lesssim \psi(g\sigma)\}$,

On the other hand, if (2) holds, then we can take the greatest $g \lesssim \psi(\bar{g})$ such that all the positions of g are also positions of t. Now it is immediate to see that $s' \lesssim t \lesssim \psi(g\sigma)$ for some $s' \in base(g, \sigma)$. Therefore $t \in \{t \,|\, g \lesssim \psi(\bar{g}),\ s' \in base(g, \sigma),\ s' \lesssim t \lesssim \psi(g\sigma)\}$.

Lemma 3.A.5 *Let $T \in \mathbb{ERT}_{\overrightarrow{x_n}}$, and $d = \sigma_0 \cdot \bar{s}_0 \xrightarrow{\varrho_1} \cdots \xrightarrow{\varrho_k} \sigma_k \cdot \bar{s}_k \in T$. Then, $\sigma \cdot s \in \tilde{\zeta}_{\{\overrightarrow{x_n}\}}(d)$ and $\forall d' \in paths(d) \setminus \{d\}. \sigma \cdot s \notin \tilde{\zeta}_{\{\overrightarrow{x_n}\}}(d')$ if and only if $\sigma \cdot s \in \langle \sigma_k \cdot \bar{s}_k \rangle_{\{\overrightarrow{x_n}\}}$ and $\forall 1 \leq i \leq k$ there exists p such that $\bar{s}_{i-1}|_p = \varrho_i$ and $s|_p \notin \mathcal{V}_\varrho$*

Proof.

⇒) By hypothesis $\sigma \cdot s \in \tilde{\zeta}_{\{\overrightarrow{x_n}\}}(d)$. Then, by a straightforward induction on the structure of d, using Point 2 of Proposition 3.1.7 and Equations (3.1.37), it can be proved that there exists $0 \leq i \leq k$ such that $\sigma \cdot s \in \langle \sigma_i \cdot \bar{s}_i \rangle_{\{\overrightarrow{x_n}\}}$ and for $\forall 1 \leq j \leq i$ there exists p such that $\bar{s}_{j-1}|_p = \varrho_j$ and $s|_p \notin \mathcal{V}_\varrho$. By hypothesis, for every proper prefix d' of d, $\sigma \cdot s \notin \tilde{\zeta}_{\{\overrightarrow{x_n}\}}(d')$. Therefore i must coincide to k.

⇐) By hypothesis and Definition 3.1.37 we have that $\sigma \cdot s \in \tilde{\zeta}_{\{\overrightarrow{x_n}\}}(d)$.

It remains to prove that, for all $d' \in paths(d) \setminus \{d\}$, $\sigma \cdot s \notin \tilde{\zeta}_{\{\overrightarrow{x_n}\}}(d')$. We proceed by contradiction: assume that there exists a proper prefix d' of d such that $\sigma \cdot s \in \tilde{\zeta}_{\{\overrightarrow{x_n}\}}(d')$. By Definition 3.1.37, $\sigma \cdot s \in \langle \sigma_i \cdot \bar{s}_i \rangle_{\{\overrightarrow{x_n}\}}$ for some $0 \leq i < k$. By Equation (3.1.38), for every position p such that $\bar{s}_i|_p = \varrho_i$, if p is a position of s then $s|_p = \varrho_i$. By Point 2 of Proposition 3.1.7 we have that for all $1 \leq j < k$ and for all p, if $\bar{s}_{j-1}|_p = \varrho_j$ than $\bar{s}_k|_p \in \mathcal{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho) \setminus \mathcal{V}_\varrho$. This is in contradiction with the hypothesis that $\forall 1 \leq i \leq k$ there exists p such that $\bar{s}_{i-1}|_p = \varrho_i$ and $s|_p \notin \mathcal{V}_\varrho$.

Lemma 3.A.6 *Given g, b an embeddable pair, $V_g = var(g)$, $V_b = var(b)$, $x \in V_g \setminus V_b$ and $V = var(g\{x/b\})$. For all $G \in \mathbb{ERT}_g$ and for all $B \in \mathbb{ERT}_b$, $\tilde{\zeta}_V(G[x/B]_V) = \tilde{\zeta}_{V_g}(G)[x/\tilde{\zeta}_{V_b}(B)]_V$*

Proof.

⊆) Let $\sigma \cdot s \in \tilde{\zeta}_V(G[x/B]_V)$. By Equations (3.1.20) and (3.1.39) there exist $d_g \in G$ and $d_b \in B$ such that $d_g[x/d_b]_V \vdash d$ and $\sigma \cdot s \in \tilde{\zeta}_V(d)$. Assume w.l.o.g. that

a) for every proper prefix d' of d, $\sigma \cdot s \notin \tilde{\zeta}_V(d')$, and

b) for every proper prefix d'_g of d_g and any proper prefix d'_b of d_b, $d'_g[x/d'_b]_V \vdash d$ does not hold.

Let $d = \sigma_0 \cdot \bar{s}_0 \xrightarrow{\varrho''_1} \cdots \xrightarrow{\varrho''_k} \sigma_k \cdot \bar{s}_k$, $d_g = \vartheta_0 \cdot \bar{g}_0 \xrightarrow{\varrho_1} \cdots \xrightarrow{\varrho_n} \vartheta_n \cdot \bar{g}_n$, and $d_b = \eta_0 \cdot \bar{b}_0 \xrightarrow{\varrho'_1} \cdots \xrightarrow{\varrho'_m} \eta_m \cdot \bar{b}_m$. By Point (b) and $d_g[x/d_b]_V \vdash d$, it holds that $\sigma_k = \sigma_{\bar{h}}\restriction_V$ and $\bar{s}_k = \bar{h}[\bar{g}_n]$ where $\bar{h} = \vartheta_n \uparrow \eta_m\{x/\bar{b}_m\}$ is $\mathcal{V}_\varrho$-preserving.

By Corollary 3.A.3, $d \in \mathbb{ERT}_{g\{x/b\}}$. Therefore, by Lemma 3.A.5, $\sigma \cdot s \in \langle \sigma_k \cdot \bar{s}_k \rangle_V$ and $\forall 1 \leq \ell \leq k$ $\exists p$ s.t. $\bar{s}_{\ell-1}|_p = \varrho''_\ell$ and $s|_p \notin \mathcal{V}_\varrho$.

Consider the set $\langle \sigma_k \cdot \bar{s}_k \rangle_V$, it can be over-approximated by (2)

$\langle \sigma_k \cdot \bar{s}_k \rangle_V =$

[by Equation (3.1.38)]

$$= \left\{ (\sigma_k \lceil \theta \rfloor) \restriction_V \cdot s'' \;\middle|\; \begin{array}{l} t \lesssim \psi(\bar{s}_k),\ \theta \in uiSubsts_t \\ s' \in base(t,\theta),\ s' \lesssim s'' \lesssim \psi(t\theta) \end{array} \right\}$$

$$[\,\forall t_1, t_2.\, t_1 \lesssim t_2 \Rightarrow uiSubsts_{t_1} \subseteq uiSubsts_{t_2}\,]$$

$$\subseteq \left\{ (\sigma_k \lceil \theta \rfloor) \restriction_V \cdot s \;\middle|\; \theta \in uiSubsts_{\psi(\bar{s}_k)},\ s'' \lesssim \psi(\bar{s}_k\theta) \right\}$$

$$[\,\text{by } \sigma_k = \sigma_{\bar{h}} \restriction_V,\ \bar{s}_k = \bar{h}[\bar{g}_n] \text{ and } \psi(\bar{h}[\bar{g}_n]) = \psi(\bar{g}_n\sigma_{\bar{h}})\,]$$

$$= \left\{ (\sigma_{\bar{h}} \lceil \theta \rfloor) \restriction_V \cdot s \;\middle|\; \theta \in uiSubsts_{\psi(\bar{g}_n\sigma_{\bar{h}})},\ s'' \lesssim \psi(\bar{g}_n\sigma_{\bar{h}}\theta) \right\}$$

$$[\,\text{by Lemma 3.A.4}\,]$$

$$= \left\{ (\sigma_{\bar{h}} \lceil \theta \rfloor) \restriction_V \cdot s \;\middle|\; \begin{array}{l} \theta \in uiSubsts_{\psi(\bar{g}_n\sigma_{\bar{h}})},\ t \lesssim \psi(\bar{g}_n), \\ s' \in base(t,\sigma_{\bar{h}}\theta),\ s' \lesssim s'' \lesssim \psi(\bar{g}_n\sigma_{\bar{h}}\theta) \end{array} \right\} \tag{1}$$

$$\subseteq \left\{ \delta \restriction_V \cdot s'' \;\middle|\; \begin{array}{l} \vartheta \cdot \tilde{g} \in \langle \vartheta_n \cdot \bar{g}_n \rangle_{V_g}\ \eta \cdot \tilde{b} \in \langle \eta_m \cdot \bar{b}_m \rangle_{V_b} \\ \delta = \vartheta \uparrow \eta\{x/\tilde{b}\}\ \mathcal{V}_\varrho\text{-preserving} \\ \forall y \in \mathcal{V} \cap var(\tilde{g}).\, y\delta \notin \mathcal{V}_\varrho \\ \tilde{b}|_p \notin \mathcal{V}_\varrho \Rightarrow (x\vartheta)|_p \notin var(x\vartheta) \setminus var(\tilde{g}) \\ s' \in base(\tilde{g},\delta),\ s' \lesssim s'' \lesssim \psi(g\delta) \end{array} \right\} \tag{2}$$

since for every partial computed result of (1) where $\theta = \varepsilon$ there exist two opportune partial computed results $\vartheta_n \cdot g' \in \langle \vartheta_n \cdot \bar{g}_n \rangle_{V_g}$ and $\eta_m \cdot b' \in \langle \eta_n \cdot \bar{b}_m \rangle_{V_b}$ which (2) can use to make it. Indeed, there exist two suitable approximations g' and b' of $\psi(\bar{g}_n)$ and $\psi(\bar{b}_m)$ respectively, such that $\delta = \vartheta_n \uparrow \eta_m\{x/b'\}$ is $\mathcal{V}_\varrho$-preserving, and $\forall y \in \mathcal{V} \cap var(g').\, y\delta \notin \mathcal{V}_\varrho$. Moreover, the requirement that $b'|_p \notin \mathcal{V}_\varrho \Rightarrow (x\vartheta_n)|_p \notin var(x\vartheta_n) \setminus var(g')$ for all positions p does not alter neither $base(g',\delta)$ nor $\psi(g'\delta)$. Similar arguments can be used also in case $\theta \neq \varepsilon$, using two opportune usage invisible instances of $\vartheta_n \cdot g'$ and $\eta_m \cdot b'$ respectively.

Actually, we are interested only on partial computed results $\sigma \cdot s \in \langle \sigma_k \cdot \bar{s}_k \rangle_V$ s.t. $\forall 1 \leq \ell \leq k$, $\exists p$ s.t. $\bar{s}_{\ell-1}|_p = \varrho''_\ell$ and $s|_p \notin \mathcal{V}_\varrho$.

By $d_g[x/d_b]_V \vdash d$ and Rules 3.1.21, for every $1 \leq j \leq m$ it holds that

- **c)** $\exists\, 1 \leq i \leq n$ such that $h = \vartheta_i \uparrow \eta_{j-1}\{x/\bar{b}_{j-1}\}$ and $h[\varrho_j]$ is $\mathcal{C}$-rooted (see Equation (3.1.21c)), or
- **d)** $\exists\, 1 \leq i \leq n$ such that $h = \vartheta_i \uparrow \eta_{j-1}\{x/\bar{b}_{j-1}\}$ is $\mathcal{V}_\varrho$-preserving and $\varrho_j \in var(h[\bar{g}_i])$ (see Equations (3.1.21b) and (3.1.21e)).

Therefore, since $\vartheta_i \preceq \vartheta_n$ for every $1 \leq i \leq n$, we have that any partial computed result $\eta \cdot \tilde{b}$ chosen in (2) satisfy $\forall 1 \leq j \leq m.\, \exists q'.\, \bar{b}_{j-1}|_{q'} = \varrho'_j$ and $\tilde{b}|_{q'} \notin \mathcal{V}_\varrho$.

Moreover, by $d_g[x/d_b]_V \vdash d$ and Rules 3.1.21, all the labels of d_g are labels of d (namely, $\{\varrho_1, \ldots, \varrho_n\} \subseteq \{\varrho''_1, \ldots, \varrho''_k\}$) and if $\varrho''_\ell = \varrho_i$ then $s_{\ell-1}|_{\varrho''_\ell} = g_{i-1}|_{\varrho_i}$. Thus, in order to satisfy $\forall 1 \leq \ell \leq k$, $\exists p$ s.t. $\bar{s}_{\ell-1}|_p = \varrho''_\ell$ and $s|_p \notin \mathcal{V}_\varrho$, in (2) it must be taken a partial computed result $\vartheta \cdot \tilde{g}$ which holds that $\forall 1 \leq i \leq n$, $\exists q$ s.t. $g_{i-1}|_q = \varrho_i$ and $\tilde{g}|_q \notin \mathcal{V}_\varrho$.

Hence, by Lemma 3.A.5, we have that, in order to make $\sigma \cdot s$, (2) can take $\vartheta \cdot \tilde{g}$ and $\eta \cdot \tilde{b}$ respectively form $\tilde{\zeta}_{V_b}(d_g)$ and $\tilde{\zeta}_{V_b}(d_b)$.

In conclusion, by Equation (3.A.1) we have that $\sigma \cdot s \in \tilde{\zeta}_{V_g}(G)[x/\,\tilde{\zeta}_{V_b}(B)]_V$.

$\supseteq$) Let $\sigma \cdot s \in \tilde{\zeta}_{V_g}(G)[x/\tilde{\zeta}_{V_b}(B)]_V$, by Equation (3.A.1) there exist $\vartheta \cdot \tilde{g} \in \tilde{\zeta}_{V_g}(G)$ and $\eta \cdot \tilde{b} \in \tilde{\zeta}_{V_b}(B)$ such that

1. $\exists \tilde{\sigma} = \vartheta \uparrow \eta\{x/\tilde{b}\}$ $\mathcal{V}_\varrho$-preserving s.t. $\forall y \in \mathcal{V} \cap var(\tilde{g}).\, y\tilde{\sigma} \notin \mathcal{V}_\varrho$,
2. for all p, $\tilde{b}|_p \notin \mathcal{V}_\varrho$ implies $(x\vartheta)|_p \notin var(x\vartheta) \setminus var(\tilde{g})$
3. $s' \lesssim s \lesssim \psi(\tilde{g}\tilde{\sigma})$ for some $s' \in base(\tilde{g}, \tilde{\sigma})$ and $\tilde{\sigma}\restriction_V = \sigma$.

By $\vartheta \cdot \tilde{g} \in \tilde{\zeta}_{V_g}(G)$, $\eta \cdot \tilde{b} \in \tilde{\zeta}_{V_b}(B)$, and Equation (3.1.39) we have that there exist $d_g \in G$ and $d_b \in B$ such that $\vartheta \cdot \tilde{g} \in \tilde{\zeta}_{V_g}(d_g)$ and $\eta \cdot \tilde{b} \in \tilde{\zeta}_{V_b}(d_b)$.

Suppose w.l.o.g. that for every proper prefix d'_g of d_g, $\vartheta \cdot \tilde{g} \notin \tilde{\zeta}_{V_g}(d'_g)$ and for every proper prefix d'_b of d_b $\eta \cdot \tilde{b} \notin \tilde{\zeta}_{V_b}(d'_b)$. Let

$$d_g = \vartheta_0 \cdot \bar{g}_0 \xrightarrow{\varrho_1} \cdots \xrightarrow{\varrho_n} \vartheta_n \cdot \bar{g}_n \text{ and } d_b = \eta_0 \cdot \bar{b}_0 \xrightarrow{\varrho'_1} \cdots \xrightarrow{\varrho'_m} \eta_m \cdot \bar{b}_m.$$

By Lemma 3.A.5 we have that $\vartheta \cdot \tilde{g} \in \langle \vartheta_n \cdot \bar{g}_n \rangle_{V_g}$ and for every $1 \leq i \leq n$, there exists a position q such that $\bar{g}_{i-1}|_q = \varrho_i$ and $\tilde{g}|_q \notin \mathcal{V}_\varrho$. Again, by Lemma 3.A.5 we have that $\eta \cdot \tilde{b} \in \langle \eta_n \cdot \bar{b}_m \rangle_{V_b}$ and for every $1 \leq j \leq m$, there exists a position q such that $\bar{b}_{j-1}|_q = \varrho'_j$ and $\tilde{b}|_q \notin \mathcal{V}_\varrho$.

By Equation (3.1.38), $(\vartheta_n \lceil \theta' \rfloor)\restriction_{V_g} = \vartheta$ and $(\eta_m \lceil \theta'' \rfloor)\restriction_{V_b} = \eta$, for some usage invisible substitutions θ,θ'. Therefore, by Point 1 we have that there exists $\bar{h} = \vartheta_n \uparrow \eta_m\{x/\bar{b}_m\}$ which is $\mathcal{V}_\varrho$-preserving and such that $(\sigma_{\bar{h}} \lceil \theta \rfloor)\restriction_V = \tilde{\sigma}\restriction_V$ for some usage invisible substitution θ. By Point 2 and since $\vartheta_i \preceq \vartheta_n$ for all $1 \leq i \leq n$, we have that none of the steps of d_b has been performed in a non demanding position. Namely, for every $1 \leq j \leq m$

- $\exists\, 1 \leq i \leq n$ such that $h = \vartheta_i \uparrow \eta_{j-1}\{x/\bar{b}_{j-1}\}$ and $h[\varrho_j]$ is $\mathcal{C}$-rooted (see rule (3.1.21c)), or
- $\exists\, 1 \leq i \leq n$ such that $h = \vartheta_i \uparrow \eta_{j-1}\{x/\bar{b}_{j-1}\}$ is $\mathcal{V}_\varrho$-preserving and $\varrho_j \in var(h[\bar{g}_i])$ (see rules (3.1.21b) and (3.1.21e)).

It can be proved that there exists a path $d = \sigma_0 \cdot \bar{s}_0 \xrightarrow{\varrho''_1} \cdots \xrightarrow{\varrho''_k} \sigma_k \cdot \bar{s}_k$ such that $d_g[x/d_b]_V \vdash d$ and $\sigma_k \cdot \bar{s}_k = \sigma_{\bar{h}}\restriction_V \cdot \bar{h}[\bar{g}_n]$. By this and Point 3, we have that $\sigma \cdot s \in \langle \sigma_{\bar{h}}\restriction_V \cdot \bar{h}[\bar{g}_n] \rangle_V$.

By $d_g[x/d_b]_V \vdash d$ and Rules 3.1.21, for every $1 \leq \ell \leq k$, ϱ''_ℓ is either ϱ_i for some $1 \leq i \leq n$ or a ϱ'_j for some $1 \leq j \leq m$. By Points 1 and 3, for every $1 \leq \ell \leq k$, there exists a position q such that $s_{\ell-1}|_q = \varrho''_\ell$ and $s|_q \neq \mathcal{V}_\varrho$. By Equations (3.1.20) and (3.1.39), $d \in G[x/B]_V$ and, by Corollary 3.A.3, $G[x/B]_V \in \mathbb{ERT}_{g\{x/b\}}$. Finally, by Lemma 3.A.5, we have that $\sigma \cdot s \in \tilde{\zeta}_V(d)$, hence $\sigma \cdot s \in \tilde{\zeta}_V(G[x/B]_V)$.

Lemma 3.A.7 *Given g, b an embeddable pair, $V_g = var(g)$, $V_b = var(b)$, $x \in V_g \setminus V_b$ and $V = var(g\{x/b\})$. For all $G \in \mathbb{ARS}_g$ and for all $B \in \mathbb{ARS}_b$, $fold_V(G[x/B]_V) = fold_{V_g}(G)[x/\, fold_{V_b}(B)]_V$*

Proof.
Straightforward, by proving that for all pair of intervals $\vartheta \centerdot g_1 – g_2$ and $\eta \centerdot b_1 – b_2$ such that $unfold_{V_g}(\vartheta \centerdot g_1 – g_2) \subseteq G$ and $unfold_{V_b}(\eta \centerdot b_1 – b_2) \subseteq B$ the following equality holds

$$unfold_{V_g}(\vartheta \centerdot g_1 – g_2)[x/\, unfold_{V_b}(\eta \centerdot b_1 – b_2)]_V = unfold_V(\{i \mid (\vartheta \centerdot g_1 – g_2)[x/(\eta \centerdot b_1 – b_2)]_V \vdash i\}) \quad (1)$$

Corollary 3.A.8 *Given g, b an embeddable pair, $V_g = var(g)$, $V_b = var(b)$, $x \in V_g \setminus V_b$ and $V = var(g\{x/b\})$. For all $G \in \mathbb{ERT}_g$ and for all $B \in \mathbb{ERT}_b$, $\zeta_V(G[x/B]_V) = \zeta_{V_g}(G)[x/\, \zeta_{V_b}(B)]_V$*

Proof.

$\zeta_V(G[x/B]_V) =$ [by $\forall V.\ \zeta_V = fold_V \circ \tilde{\zeta}_V$]
$= fold_V(\tilde{\zeta}_V(G[x/B]_V))$ [by Lemma 3.A.6]
$= fold_V(\tilde{\zeta}_{V_g}(G)[x/\, \tilde{\zeta}_{V_b}(B)]_V)$ [by Lemma 3.A.7]
$= fold_{V_g}(\tilde{\zeta}_{V_g}(G))[x/\, fold_{V_b}(\tilde{\zeta}_{V_b}(B))]_V$ [by $\forall V.\ \zeta_V = fold_V \circ \tilde{\zeta}_V$]
$= \zeta_{V_g}(G)[x/\, \zeta_{V_b}(B)]_V$

Proof of Theorem 3.1.43.
By $\nu_V = \zeta_V \circ \partial_V$ and Theorem 3.1.13, $\nu_{var(e)}(\mathcal{E}[\![e]\!]_{\mathcal{I}}) = \zeta_{var(e)}(\mathcal{E}^{\partial}[\![e]\!]_{\partial(\mathcal{I})})$. It remains to prove that $\zeta_{var(e)}(\mathcal{E}^{\partial}[\![e]\!]_{\partial(\mathcal{I})}) = \mathcal{E}^{\nu}[\![e]\!]_{\zeta(\partial(\mathcal{I}))}$. Let $\mathcal{I}^{\partial} := \partial(\mathcal{I})$, we procede by induction on the cardinality of $baseknots(e)$

Base Case: there are to possible cases: if $e = x \in \mathcal{V}$ is immediate.

If $e = \varphi(\overrightarrow{x_n})$, let $\overrightarrow{y_n} \in \mathcal{V}$ fresh distinct, and let $V_i := \{x_1, \ldots, x_i, y_{i+1}, \ldots, y_n\}$ for $i \in \{1, \ldots, n\}$. Then

$\zeta_{\{\overrightarrow{x_n}\}}(\mathcal{E}^{\partial}[\![\varphi(\overrightarrow{x_n})]\!]_{\mathcal{I}^{\partial}}) =$
[by Equation (3.1.19b)]
$= \zeta_{\{\overrightarrow{x_n}\}}(\mathcal{I}^{\partial}(\varphi(\overrightarrow{y_n}))[y_1/\varepsilon \cdot x_1]_{V_1} \ldots [y_n/\varepsilon \cdot x_n]_{V_n})$
[repeatedly applying Corollary 3.A.8]
$= \zeta_{\{\overrightarrow{y_n}\}}(\mathcal{I}^{\partial}(\varphi(\overrightarrow{y_n})))[y_1/\, \zeta_{\{x_1\}}(\varepsilon \cdot x_1)]_{V_1} \ldots [y_n/\, \zeta_{\{x_n\}}(\varepsilon \cdot x_n)]_{V_n}$
[by $\forall \pi.\ \zeta_{\pi}(\mathcal{I}^{\partial}(\pi)) = \zeta(\mathcal{I}^{\partial})(\pi)$ and the former case]
$= \zeta(\mathcal{I}^{\partial})(\varphi(\overrightarrow{y_n}))[y_1/\, \mathcal{E}^{\nu}[\![x_1]\!]_{\zeta(\mathcal{I}^{\partial})}]_{V_1} \ldots [y_n/\, \mathcal{E}^{\nu}[\![x_n]\!]_{\zeta(\mathcal{I}^{\partial})}]_{V_n}$
[by Equation (3.1.52b) and construction of $\mathcal{I}^{\partial}$]
$= \mathcal{E}^{\nu}[\![\varphi(\overrightarrow{x_n})]\!]_{\zeta(\mathcal{I}^{\partial})} = \mathcal{E}^{\nu}[\![\varphi(\overrightarrow{x_n})]\!]_{\zeta(\partial(\mathcal{I}))}$

Inductive Step: Let p be the leftmost position in $baseknots(e)$ and y a fresh data variable, then

$$\begin{aligned}
&\zeta_{var(e)}(\mathcal{E}^{\partial}[\![e]\!]_{\mathcal{I}^{\partial}}) = && [\text{by Equation (3.1.19c)}]\\
&= \zeta_{var(e)}(\mathcal{E}^{\partial}[\![e[y]_p]\!]_{\mathcal{I}^{\partial}}[y/\,\mathcal{E}^{\partial}[\![e|_p]\!]_{\mathcal{I}^{\partial}}]_{var(e)}) && [\text{by Corollary 3.A.8}]\\
&= \zeta_{var(e[y]_p)}(\mathcal{E}^{\partial}[\![e[y]_p]\!]_{\mathcal{I}^{\partial}})[y/\,\zeta_{var(e|_p)}(\mathcal{E}^{\partial}[\![e|_p]\!]_{\mathcal{I}^{\partial}})]_{var(e)} && [\text{by ind. hypothesis}]\\
&= \mathcal{E}^{\nu}[\![e[y]_p]\!]_{\zeta(\mathcal{I}^{\partial})}[y/\,\mathcal{E}^{\nu}[\![e|_p]\!]_{\zeta(\mathcal{I}^{\partial})}]_{var(e)} && [\text{by Equation (3.1.52c)}]\\
&= \mathcal{E}^{\nu}[\![e]\!]_{\zeta(\mathcal{I}^{\partial})} = \mathcal{E}^{\nu}[\![e]\!]_{\zeta(\partial(\mathcal{I}))}
\end{aligned}$$

Proof of Corollary 3.1.44.

Point 1) We first show that, given $\mathcal{I}^{\partial} \in \mathbb{I}_{\mathbb{ERT}}$, $f(\overrightarrow{t_n}) \to r \in P$, and $\overrightarrow{x_n} \in \mathcal{V}$ fresh distinct, it holds that

$$\zeta_{\{\overrightarrow{x_n}\}}(\xi^{\partial}_{\overrightarrow{x_n}}[\![r]\!]^{\{\overrightarrow{x_n}/\overrightarrow{t_n}\}}_{\mathcal{I}^{\partial}}) = zap^{\{\overrightarrow{x_n}/\overrightarrow{t_n}\}}_{\{\overrightarrow{x_n}\}}(\zeta_{var(r)}(\mathcal{E}^{\partial}[\![r]\!]_{\mathcal{I}^{\partial}})) \tag{1}$$

We proceed by cases as Equation (3.1.16)

- if r is op-rooted, $\xi^{\partial}_{\overrightarrow{x_n}}[\![r]\!]^{\{\overrightarrow{x_n}/\overrightarrow{t_n}\}}_{\mathcal{I}^{\partial}} = zaproot(\{\overrightarrow{x_n}/\overrightarrow{t_n}\}\lfloor\mathcal{E}^{\partial}[\![r]\!]_{\mathcal{I}^{\partial}}\rfloor_{\overrightarrow{x_n}})$. By Definition 3.1.52 the head of $\mathcal{E}^{\partial}[\![r]\!]_{\mathcal{I}^{\partial}}$ is of the form $\varepsilon \cdot \varrho$ for some $\varrho \in \mathcal{V}_{\varrho}$. By definition of *zaproot* and Equation (3.1.17) $\varepsilon \cdot \varrho$ is the root node of $\xi^{\partial}_{\overrightarrow{x_n}}[\![r]\!]^{\{\overrightarrow{x_n}/\overrightarrow{t_n}\}}_{\mathcal{I}^{\partial}}$. Moreover, for every node $\sigma \cdot s$ of $\mathcal{E}^{\partial}[\![r]\!]_{\mathcal{I}^{\partial}}$ such that $s \notin \mathcal{V}_{\varrho}$ there is a node $(\{\overrightarrow{x_n}/\overrightarrow{t_n}\}\sigma)\restriction_{\{\overrightarrow{x_n}\}} \cdot s$ in $\xi^{\partial}_{\overrightarrow{x_n}}[\![r]\!]^{\{\overrightarrow{x_n}/\overrightarrow{t_n}\}}_{\mathcal{I}^{\partial}}$ corresponding to it and vice versa. Thus, by definition of *zap* and ζ_V we have that
- if r is not op-rooted, $\xi^{\partial}_{\overrightarrow{x_n}}[\![r]\!]^{\{\overrightarrow{x_n}/\overrightarrow{t_n}\}}_{\mathcal{I}^{\partial}} = \varepsilon \cdot \varrho \xrightarrow{\varrho} \{\overrightarrow{x_n}/\overrightarrow{t_n}\}\lfloor\mathcal{E}^{\partial}[\![r]\!]_{\mathcal{I}^{\partial}}\rfloor_{\overrightarrow{x_n}}$. By Definition 3.1.52 the head of $\mathcal{E}^{\partial}[\![r]\!]_{\mathcal{I}^{\partial}}$ is $\varepsilon \cdot \tau(r)$ where $\tau(r)$ is not op-rooted. It is straightforward to see that

  $$\varepsilon \cdot \varrho \xrightarrow{\varrho} \{\overrightarrow{x_n}/\overrightarrow{t_n}\}\lfloor\mathcal{E}^{\partial}[\![r]\!]_{\mathcal{I}^{\partial}}\rfloor_{\overrightarrow{x_n}} = zaproot(\{\overrightarrow{x_n}/\overrightarrow{t_n}\}\lfloor\varepsilon \cdot \varrho \xrightarrow{\varrho} \mathcal{E}^{\partial}[\![r]\!]_{\mathcal{I}^{\partial}}\rfloor_{\overrightarrow{x_n}})$$

  Thus, similarly to the former case we have that (1) holds.

Now we prove that $\mathcal{P}^{\nu}[\![P]\!] \circ \zeta = \zeta \circ \mathcal{P}^{\partial}[\![P]\!]$. Given $\mathcal{I}^{\partial} \in \mathbb{I}_{\mathbb{ERT}}$, and $f(\overrightarrow{x_n}) \in \mathbb{MGC}$ we have that

$$\begin{aligned}
&\zeta_{\{x\}}(\mathcal{P}^{\partial}[\![P]\!]_{\mathcal{I}^{\partial}}(f(\overrightarrow{x_n}))) = && [\text{by Equation (3.1.15)}]\\
&= \zeta_{\{x\}}(\bigvee\{\xi^{\partial}_{\overrightarrow{x_n}}[\![r]\!]^{\{\overrightarrow{x_n}/\overrightarrow{t_n}\}}_{\mathcal{I}^{\partial}} \mid f(\overrightarrow{t_n}) \to r \ll P\}) && [\zeta_V \circ \bigvee = \hat{\bigvee} \circ \zeta_V]\\
&= \hat{\bigvee}\left\{\zeta_{\{x\}}(\xi^{\partial}_{\overrightarrow{x_n}}[\![r]\!]^{\{\overrightarrow{x_n}/\overrightarrow{t_n}\}}_{\mathcal{I}^{\partial}}) \,\middle|\, f(\overrightarrow{t_n}) \to r \ll P\right\} && [\text{by (1)}]\\
&= \hat{\bigvee}\left\{zap^{\{\overrightarrow{x_n}/\overrightarrow{t_n}\}}_{\{\overrightarrow{x_n}\}}(\zeta_{var(r)}(\mathcal{E}^{\partial}[\![r]\!]_{\mathcal{I}^{\partial}})) \,\middle|\, f(\overrightarrow{t_n}) \to r \ll P\right\} && [\text{by Theorem 3.1.13}]\\
&= \hat{\bigvee}\left\{zap^{\{\overrightarrow{x_n}/\overrightarrow{t_n}\}}_{\{\overrightarrow{x_n}\}}(\mathcal{E}^{\partial}[\![r]\!]_{\zeta(\mathcal{I}^{\partial})}) \,\middle|\, f(\overrightarrow{t_n}) \to r \ll P\right\} && [\text{by Equation (3.1.50)}]\\
&= \mathcal{P}^{\nu}[\![P]\!]_{\zeta(\mathcal{I}^{\partial})}(f(\overrightarrow{x_n}))
\end{aligned}$$

Finally, we have that

$$\begin{aligned}
&\mathcal{P}^{\nu}[\![P]\!] \circ \nu && [\text{since } \nu = \zeta \circ \partial] \\
&= \mathcal{P}^{\nu}[\![P]\!] \circ \zeta \circ \partial && [\text{since } \mathcal{P}^{\nu}[\![P]\!] \circ \zeta = \zeta \circ \mathcal{P}^{\partial}[\![P]\!]] \\
&= \zeta \circ \mathcal{P}^{\partial}[\![P]\!] \circ \partial && [\text{by Point 1 of Corollary 3.1.14}] \\
&= \zeta \circ \partial \circ \mathcal{P}[\![P]\!] && [\text{again, since } \nu = \zeta \circ \partial] \\
&= \nu \circ \mathcal{P}[\![P]\!]
\end{aligned}$$

Point 2) Identical to proof of Point 2 of Corollary 3.1.14 with ν instead of ∂ and Point 1 of Corollary 3.1.44 instead of Point 1 of Corollary 3.1.14

Point 3) $\mathcal{F}^{\nu}[\![P]\!] = \mathcal{P}^{\nu}[\![P]\!]\uparrow\omega$ is an immediate consequence of Point 2.

Point 4) Identical to proof of Point 4 of Corollary 3.1.14 with Point 3 instead of Point 3 of Corollary 3.1.14 and Point 1 instead of Point 1 of Corollary 3.1.44.

Point 5) Immediate by Point 4 and Definition 3.1.37.

Corollary 3.A.9 *For all* $P_1, P_2 \in \mathbb{P}_{\Sigma}$, $\mathcal{F}^{\nu}[\![P_1]\!] = \mathcal{F}^{\nu}[\![P_2]\!]$ *implies* $P_1 \approx_{cr} P_2$.

Proof.

For all $e \in \mathcal{T}(\Sigma, \mathcal{V})$ we have that

$$\begin{aligned}
&\nu_{var(e)}(\mathcal{N}[\![e \text{ in } P]\!]) = && [\text{by Theorems 2.2.19 and 2.2.25}] \\
&\nu_{var(e)}(\mathcal{E}[\![e]\!]_{\mathcal{F}[\![P]\!]}) = && [\text{by Theorem 3.1.43}] \\
&\mathcal{E}^{\nu}[\![e]\!]_{\nu(\mathcal{F}[\![P]\!])} = && [\text{by Point 4 of Corollary 3.1.44}] \\
&\mathcal{E}^{\nu}[\![e]\!]_{\mathcal{F}^{\nu}[\![P]\!]}
\end{aligned}$$

Therefore,

$$\begin{aligned}
&\mathcal{F}^{\nu}[\![P_1]\!] = \mathcal{F}^{\nu}[\![P_2]\!] \\
&\quad [\text{by the former equalities and Corollary 2.2.26}] \\
&\iff \forall e \in \mathcal{T}(\Sigma, \mathcal{V}).\ \nu_{var(e)}(\mathcal{N}[\![e \text{ in } P_1]\!]) = \nu_{var(e)}(\mathcal{N}[\![e \text{ in } P_2]\!]) \\
&\quad [\nu_V = \zeta_V \circ \partial_V] \\
&\iff \forall e \in \mathcal{T}(\Sigma, \mathcal{V}).\ \zeta_{var(e)}(\partial_{var(e)}(\mathcal{N}[\![e \text{ in } P_1]\!])) = \zeta_{var(e)}(\partial_{var(e)}(\mathcal{N}[\![e \text{ in } P_2]\!]))
\end{aligned}$$

As stated in the proof of Corollary 3.1.15 for every canonical small-step derivation $e \stackrel{\vartheta}{\Rightarrow}^{*} v$ with $v \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, the last node of $\partial^{\varepsilon}_{var(e)}(e \stackrel{\vartheta}{\Rightarrow}^{*} v)$ is $\vartheta \cdot v$. By the maximality of v w.r.t. $\lesssim$ and Equation (3.1.46) in $\zeta_{var(e)}(\partial^{\varepsilon}_{var(e)}(e \stackrel{\vartheta}{\Rightarrow}^{*} v))$ there is an interval of the form $\vartheta \centerdot s\text{–}v'$ for some $s \in \mathbb{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_{\varrho})$ and $v' = \psi(v)$. Since $v \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, by Equation (3.1.36) $v' = \wr v \int$. Thus, by Definitions 3.1.1 and 2.2.12, we conclude that $\mathcal{F}^{\nu}[\![P_1]\!] = \mathcal{F}^{\nu}[\![P_2]\!]$ implies $P_1 \approx_{cr} P_2$.

Lemma 3.A.10 *For any given $t \in \mathbb{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho)$ there exist $t' \in ui\mathbb{T}(\mathcal{C}, \mathcal{V}, \mathcal{V}_\varrho)$ and $\theta \colon \mathcal{V} \to \mathbb{T}(\mathcal{C}, \mathcal{V})$ such that $t = t'\theta$*

Proof.

If $t \in ui\mathbb{T}(\mathcal{C}, \mathcal{V}, \mathcal{V}_\varrho)$ then it suffices to take $t' = t$ and $\theta = \varepsilon$. Otherwise, there exists a non variable position p such that $t|_p \in \mathbb{T}(\mathcal{C}, \mathcal{V})$. Let $s_0 := t[y]_p$ and $\sigma_0 = \{y/t|_p\}$ with $y \ll \mathcal{V}$. If $s_0 \in ui\mathbb{T}(\mathcal{C}, \mathcal{V}, \mathcal{V}_\varrho)$ we are done, otherwise, we repeat this construction on s_0 obtaining s_1 and σ_1 going on until a s_n such that $s_n \in ui\mathbb{T}(\mathcal{C}, \mathcal{V}, \mathcal{V}_\varrho)$ is eventually reached.

Then, we conclude by taking taking $t' = s_n$ and $\theta = \sigma_0 \cdots \sigma_n$.

Proof of Theorem 3.1.45.

$\Rightarrow$) Identical to proof of Theorem 3.1.18 with $\mathcal{P}^\nu[\![P]\!]$ instead of $\mathcal{P}^\partial[\![P]\!]$, Equation (3.1.50) instead of Equation (3.1.15) and Corollary 3.A.9 instead of Corollary 3.1.15.

$\Leftarrow$) Before starting the proof we need to define some auxiliary suffs: consider a signature $\hat{\Sigma} = \hat{\mathcal{C}} \cup \hat{\mathcal{D}}$ disjoint from Σ and the transformation $\widehat{\cdot} \colon \mathbb{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho) \to \mathbb{T}(\hat{\mathcal{C}}, \mathcal{V})$ defined by

$$\widehat{t} := \begin{cases} x & \text{if } t = x \in \mathcal{V} \\ \bot & \text{if } t \in \mathcal{V}_\varrho \\ c(\widehat{t_1}, \ldots, \widehat{t_n}) & \text{if } t = c(t_1, \ldots, t_n) \end{cases}$$

where $\bot \in \hat{\mathcal{C}}$.

To any ψ-term t we associate a program $\mathcal{Q}[\![t]\!] := \{g_t(a) \to b\}$ where $g_t \in \hat{\mathcal{D}}$ and $\langle a, b\rangle := toRule(t)$

$$toRule(t) := \begin{cases} \langle y, y\rangle & \text{if } t \in \mathcal{V} \text{ and } y \ll \mathcal{V} \\ \langle y, \bot\rangle & \text{if } t \in \mathcal{V}_\varrho \text{ and } y \ll \mathcal{V} \\ \langle c(\overrightarrow{a_n}), \hat{c}(\overrightarrow{b_n})\rangle & \text{if } t = c(\overrightarrow{t_n}) \text{ and } \forall 1 \le i \le n.\, \widetilde{t_i} = \langle a_i, b_i\rangle \end{cases}$$

By Definition 3.1.17, $\mathcal{Q}[\![t]\!] \in \mathbb{UP}^{\hat{\Sigma}}_{\Sigma}$ for any given $t \in \mathbb{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho)$.

By Proposition 3.1.29 we can reason in $\mathbb{ARS}_{\overrightarrow{x_n}}$ instead of $\mathbb{WERS}_{\overrightarrow{x_n}}$, thus in the rest of the proof, by abuse of notation, we will write $\sigma \cdot s \in A$ to indicate $\sigma \cdot s \in unfold_{\{\overrightarrow{x_n}\}}(A)$ for any given $A \in \mathbb{WERS}_{\overrightarrow{x_n}}$.

Now we prove the thesis by showing that $\mathcal{F}^\nu[\![P_1]\!] \neq \mathcal{F}^\nu[\![P_2]\!]$ implies $P_1 \not\approx^{us}_{cr} P_2$. To this end we proceed by contradiction. Assume that $\mathcal{F}^\nu[\![P_1]\!] \neq \mathcal{F}^\nu[\![P_2]\!]$ but $P_1 \approx^{us}_{cr} P_2$, then there exists a $f_{/n} \in \mathcal{D}$ such that $\mathcal{F}^\nu[\![P_1]\!](f(\overrightarrow{x_n})) \neq \mathcal{F}^\nu[\![P_2]\!](f(\overrightarrow{x_n}))$. Assume, w.l.o.g., that $\sigma \cdot s \in \mathcal{F}^\nu[\![P_1]\!](f(\overrightarrow{x_n}))$ but $\sigma \cdot s \notin \mathcal{F}^\nu[\![P_2]\!](f(\overrightarrow{x_n}))$ and that $\sigma \cdot s$ is the most general partial computed result holding the previous condition.

Let $\mathcal{R}_i := P_i \cup \mathcal{Q}[\![s]\!]$ for $i \in \{1, 2\}$ where $\mathcal{Q}[\![s]\!] = \{g_s(\pi_s) \to \hat{\pi}_s\}$.

As proven in Corollary 3.A.9, we have that all the (totally) computed result belonging to $\mathcal{E}^\nu[\![g_s(f(\overrightarrow{x_n}))]\!]_{\mathcal{F}^\nu[\![\mathcal{R}_1]\!]}$ belongs to $\mathcal{B}^{cr}[\![\mathcal{R}_1]\!](g_s(f(\overrightarrow{x_n})))$. Thus, it is straightforward to see that $\sigma \cdot \widehat{s} \in \mathcal{B}^{cr}[\![\mathcal{R}_1]\!](g_s(f(\overrightarrow{x_n})))$. Hence, by $\sigma \cdot s \notin \mathcal{F}^\nu[\![P_2]\!](f(\overrightarrow{x_n}))$, we also have

$\sigma \cdot \widehat{s} \in \mathcal{B}^{cr}[\![\mathcal{R}_2]\!](g_s(f(\overrightarrow{x_n})))$. Again, this implies that $\sigma \cdot \widehat{s} \in \mathcal{E}^{\nu}[\![g_s(f(\overrightarrow{x_n}))]\!]_{\mathcal{F}^{\nu}[\![\mathcal{R}_2]\!]}$. Therefore

$$
\begin{aligned}
&\sigma \cdot \widehat{s} \in \mathcal{E}^{\nu}[\![g_s(f(\overrightarrow{x_n}))]\!]_{\mathcal{F}^{\nu}[\![\mathcal{R}_2]\!]} \\
&\quad [\text{by Definition 3.1.52 with } y \ll \mathcal{V}] \\
&\iff \sigma \cdot \widehat{s} \in \mathcal{F}^{\nu}[\![\mathcal{R}_2]\!](g_s(y))[y/\mathcal{F}^{\nu}[\![\mathcal{R}_2]\!](f(\overrightarrow{x_n}))]_{\{\overrightarrow{x_n}\}} \\
&\quad [\text{by } \mathcal{F}^{\nu}[\![\mathcal{R}_2]\!](f(\overrightarrow{x_n})) = \mathcal{F}^{\nu}[\![P_2]\!](f(\overrightarrow{x_n}))] \\
&\iff \sigma \cdot \widehat{s} \in \mathcal{F}^{\nu}[\![\mathcal{R}_2]\!](g_s(y))[y/\mathcal{F}^{\nu}[\![P_2]\!](f(\overrightarrow{x_n}))]_{\{\overrightarrow{x_n}\}}. \qquad (1)
\end{aligned}
$$

By Equation (3.A.1), $\sigma \cdot \widehat{s}$ can be retrieved in (1) only from the (totally) computed result $\{y/\pi_s\} \cdot \hat{\pi}_s \in \mathcal{F}^{\nu}[\![\mathcal{R}_2]\!](g_s(y))$ and a partial computed result $\sigma' \cdot s' \in \mathcal{F}^{\nu}[\![P_2]\!](f(\overrightarrow{x_n}))$ satisfying the following conditions

1. $\exists \delta = \{y/\pi_s\} \uparrow \sigma'\{y/s'\}$ $\mathcal{V}_{\varrho}$-preserving s.t. $\forall y \in var(\hat{\pi}_s).\, y\delta \notin \mathcal{V}_{\varrho}$
2. $\forall p.\, s'|_p \notin \mathcal{V}_{\varrho} \Rightarrow \pi_s|_p \notin var(\pi_s) \setminus var(\hat{\pi}_s)$
3. $\sigma \cdot \widehat{s} = \delta\restriction_{\overrightarrow{x_n}} \cdot \hat{\pi}_s\delta$.

Let $\theta := \sigma'\, mgu(\pi_s, s')$, by the previous conditions we have that $z\delta = z\theta$ for every $z \in \mathcal{V} \setminus \{y\}$.

By Point 2, for any position p s.t. $\pi_s|_p \in var(\pi_s) \setminus var(\hat{\pi}_s)$ either p is not a position of s' (and, by Point 1, there exists an outer position p' s.t. $s'|_{p'} \in \mathcal{V}$), or $s'|_p \in \mathcal{V}_{\varrho}$. By construction $\pi_s\rho\rho' = s$ for some $\rho\colon\ var(\pi_s) \setminus var(\hat{\pi}_s) \to \mathcal{V}_{\varrho}$ and $\rho'\colon\ var(\hat{\pi}_s) \to \mathcal{V}$, and $\widehat{s} = \hat{\pi}_s\rho'$. Thus, by Point 3, we have that $\hat{\pi}_s\rho' = \hat{\pi}_s\delta$ and $mgu(s, s')$ exists and is $\mathcal{V}_{\varrho}$-preserving (up-to renaming on bottom variables).

In particular we have that $s'\theta' = s$ for some substitution $\theta'\colon \mathcal{V} \to \mathbb{T}(\mathcal{C}, \mathcal{V}_{\varrho} \cup \mathcal{V})$. There are to possible cases

$\theta' \in uiSubsts_{s'}$) in this case we have that $\sigma \cdot s$ is a usage invisible instance of $\sigma' \cdot s'$, which implies that $\sigma \cdot s \in \mathcal{F}^{\nu}[\![P_2]\!](f(\overrightarrow{x_n}))$, since $unfold_{\{\overrightarrow{x_n}\}}(\mathcal{F}^{\nu}[\![P_2]\!](f(\overrightarrow{x_n})))$ is closed under invisible instantiations. This contradicts the hypothesis that $\sigma \cdot s \notin \mathcal{F}^{\nu}[\![P_2]\!](f(\overrightarrow{x_n}))$.

$\theta' \notin uiSubsts_{s'}$) By decomposing every ψ-term in $img(\theta')$ as in Lemma 3.A.10 we can decompose θ' as $\vartheta\vartheta'$ where $\vartheta \in uiSubsts_{s'}$ and $\vartheta'\colon\ var(s'\theta'') \to \mathbb{T}(\mathcal{C}, \mathcal{V})$. Let $\mathcal{R}'_i = P_i \cup \mathcal{Q}[\![s'\vartheta]\!]$ for $i \in \{1, 2\}$.

Since $\sigma \cdot s \in \mathcal{F}^{\nu}[\![P_1]\!](f(\overrightarrow{x_n}))$ but $\sigma \cdot s \notin \mathcal{F}^{\nu}[\![P_2]\!](f(\overrightarrow{x_n}))$, it can be shown that $\sigma \cdot \widehat{s} \in \mathcal{E}^{\nu}[\![g_{(s'\theta)}(f(\overrightarrow{x_n}))]\!]_{\mathcal{F}^{\nu}[\![\mathcal{R}'_1]\!]}$ but $\sigma \cdot \widehat{s} \notin \mathcal{E}^{\nu}[\![g_{(s'\theta)}(f(\overrightarrow{x_n}))]\!]_{\mathcal{F}^{\nu}[\![\mathcal{R}'_2]\!]}$. Hence, $\sigma \cdot \widehat{s} \in \mathcal{B}^{cr}[\![\mathcal{R}'_1]\!](g_{(s'\vartheta)}(f(\overrightarrow{x_n})))$ but $\sigma \cdot \widehat{s} \notin \mathcal{B}^{cr}[\![\mathcal{R}'_2]\!](g_{(s'\vartheta)}(f(\overrightarrow{x_n})))$. This contradicts the assumption $P_1 \approx^{us}_{cr} P_2$.

In conclusion, we proved that $\mathcal{F}^{\nu}[\![P_1]\!] \neq \mathcal{F}^{\nu}[\![P_2]\!]$ implies $P_1 \not\approx^{us}_{cr} P_2$, that is to say, $P_1 \approx^{us}_{cr} P_2$ implies $\mathcal{F}^{\nu}[\![P_1]\!] = \mathcal{F}^{\nu}[\![P_2]\!]$.

Proof of Proposition 3.2.6.
By structural induction on e_1.

Proof of Corollary 3.2.8.
Immediate by Definitions 2.2.2 and 3.2.7, and Proposition 3.2.6.

Proof of Theorem 3.2.9.

$\Leftarrow$) Immediate by Corollary 3.2.8.

$\Rightarrow$) Suppose that $Cnv(P_1) \not\approx_{fss} Cnv(P_2)$, that is $\mathcal{B}^{fss}[\![Cnv(P_1)]\!] \neq \mathcal{B}^{fss}[\![Cnv(P_1)]\!]$. We proceed by showing that there exists a ground term t such that the Curry small-step behavior for t w.r.t. $Cnv(P_1)$ differs from that w.r.t. $Cnv(P_2)$. If such a term exists it is a witness for $\mathcal{B}^{fss}_{gr}[\![Cnv(P_1)]\!] \neq \mathcal{B}^{fss}_{gr}[\![Cnv(P_2)]\!]$. By Corollary 3.2.8, we have that $\mathcal{B}^{H}[\![P_1]\!] \neq \mathcal{B}^{H}[\![P_2]\!]$ therefore, by Definition 3.2.7, $P_1 \not\approx_H P_2$ which concludes the proof.

It remains to show how to construct t. To this aim, we recall the notion of (syntactic) program equivalence $\overset{pt}{=}$ defined in the proof of Lemma 2.2.3 where its has been proved that the equivalence $\overset{pt}{=}$ is exactly $\approx_{fss}$.

Let T_1 and T_2 two definitional tree for some operator f respectively belonging $Cnv(P_1)$ and $Cnv(P_2)$ such that $T_1 \overset{pt}{\neq} T_2$ (such trees exist by hypothesis). By construction $pat(T_1) = pat(T_2)$, let denote with π that pattern. Let also denote with $\bot$ a looping expression. As observed in Observation 3.2.2, neither T_1 nor T_2 has or-nodes thus we can describe a constructive method for generating t by cases as follows:

1. if both T_1 and T_2 are brach nodes. Let p_1 and p_2 be the inductive position of T_1 and T_2 respectively, then there are two possible cases:
   $\boldsymbol{p_1 \neq p_2}$) We take t as a ground instance of π such that $t|_{p_1} = \bot$ and $t|_{p_2} = \bot$.
   $\boldsymbol{p_1 = p_2}$) Let $T_1^1, \ldots, T_1^{k_1}$ be the subtrees of T_1 and $T_2^1, \ldots, T_2^{k_2}$ be those of T_2.
   If there is a T_1^i for some $1 \leq i \leq k_1$, such that $pat(T_1^i) \neq pat(T_2^j)$ for all $1 \leq j \leq k_2$, we take as t a ground instance of a pattern of a rule node reachable from T_1^i (we make the same construction if the former condition holds replacing T_1 with T_2)
   Otherwise, we chose two subtrees T_1^i and T_2^j for some $1 \leq i \leq k_1$ and $1 \leq j \leq k_2$ such that $pat(T_1^i) = pat(T_2^j)$ and $T_1^i \overset{pt}{\neq} T_2^j$ and we repeat the construction of t starting from T_1^i and T_2^j.
2. if T_1 or T_2 are both rule nodes t will be any ground instance of π. It is worth noting that the RHS of the rule of T_1 is must be different from that of T_2.
3. otherwise, w.l.o.g. we have that T_1 is a branch node whereas T_2 is a rule node (the case). Then we take t as a ground instance of π such that $t|_p = \bot$, where p is the inductive position of T_1.

It can be shown that the term t generated by the former construction is such that the Curry small-step behavior for t w.r.t. $Cnv(P_1)$ differs from that of t w.r.t. $Cnv(P_2)$.

Proof of Proposition 3.2.11.
Immediate by Corollary 3.2.8 and Definition 3.2.10.

4

Abstraction Framework

Abstract

In this chapter, starting from the fixpoint semantics in Subsection 3.1.3, we describe how to develop an abstract semantics which approximates the observable behavior of a Curry program. We focus on particular program properties modeled as Galois Insertions between the domain $\mathbb{I}_{\mathbb{WERS}}$ and the abstract domain chosen to model the property.

Then, we present two motivating instances of this abstraction framework by means of two well studied abstract domains, namely the *depth*(k) and the $\mathcal{POS}$ domains. The first one will be used to model the computed result behavior up to a fixed depth, and the second one will be used to model the computed result groundness behavior.

4.1 Abstraction Scheme

In this section, starting from the fixpoint semantics in Subsection 3.1.3, we develop an abstract semantics which approximates the observable behavior of the program. Program properties which can be of interest are Galois Insertions between the domain $\mathbb{I}_{\mathbb{WERS}}$ and the abstract domain chosen to model the property.

We will restrict our attention to a special class of abstract interpretations which are obtained from what we call a *weak evolving result set abstraction* that is a Galois Insertion $(\mathbb{WERS}, \hat{\preccurlyeq}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathbb{A}, \leq)$. This abstraction can be systematically lifted to a Galois Insertion $\mathbb{I}_{\mathbb{WERS}} \underset{\bar{\alpha}}{\overset{\bar{\gamma}}{\leftrightarrows}} [\mathbb{MGC} \to \mathbb{A}]$ by function composition (i.e., $\bar{\alpha}(f) = \alpha \circ f$).

Then we can derive the optimal abstract version of $\mathcal{P}^\alpha[\![P]\!]$ simply as $\mathcal{P}^\alpha[\![P]\!] := \bar{\alpha} \circ \mathcal{P}^\nu[\![P]\!] \circ \bar{\gamma}$ guaranteeing that $\mathcal{F}^\alpha[\![P]\!] := \mathcal{P}^\alpha[\![P]\!]{\uparrow}\omega$ is the best correct approximation of $\mathcal{F}^\nu[\![P]\!]$ by construction. We recall that correct means $\alpha(\mathcal{F}^\nu[\![P]\!]) \leq \mathcal{F}^\alpha[\![P]\!]$ and best means that it is the minimum (w.r.t. $\leq$) of all correct approximations. In particular, if $\mathbb{A}$ is Noetherian the abstract fixpoint is reached in a finite number of steps, that is, there exists a finite natural number h such that $\mathcal{P}^\alpha[\![P]\!]{\uparrow}\omega = \mathcal{P}^\alpha[\![P]\!]{\uparrow}h$.

4.2 Case study

In this section, following the abstraction scheme of Section 4.1, we present two instances of our framework, namely by using the *depth*(k) and the $\mathcal{POS}$ domains. *depth*(k) will be used for modeling respectively the computed result behavior up to a fixed depth of the results, while $\mathcal{POS}$ will be used for modeling the groundness behavior of the computed result.

The case studies proposed in this section aim at showing the applicative potentiality of the abstraction framework even if applied to (conceptually) simple abstract properties. Indeed, in Chapters 5 and 6 we will exploit weak evolving result set abstractions for the development of semantic-based tools.

4.2.1 Modeling the Groundness Behaviour of Curry

We now outline a simple example of analysis obtained by abstraction of the weak evolving result set semantics, namely Groundness Analysis of computed results in Curry.

In order to define the abstract domain we have to do several small steps. We will use propositional formulas to represent the groundness dependencies of variables. In particular, we will use the domain $\mathcal{POS}$ [7] of positive propositional formula classes modulo logical equivalence, built using $\leftrightarrow$, $\wedge$ and $\vee$ over variable names[1], and ordered by logical implication.

First of all (following the lines of [23, 25]) we have to define the abstraction $\Gamma(t)$ of a term $t \in \mathbb{T}(\mathcal{C}, \mathcal{V})$ as

$$\Gamma(t) := \bigwedge_{x \in var(t)} x \tag{4.2.1}$$

The formula intuitively suggests that in order for t to be ground, all its variables must be ground.

We can lift Γ to substitutions to obtain abstract substitutions as[2]

$$\Gamma(\vartheta) := \bigwedge_{x/t \in \vartheta} (x \leftrightarrow \Gamma(t))$$

Abstract substitutions are propositional formulas which express the *groundness dependencies* between the variables of the domain and the ones of the range of the concrete substitution. We must define the abstract notion of restriction of an abstract substitution w.r.t. a set of variables. Namely, [3]

$$F|_{\overrightarrow{x_n}} := \begin{cases} F & \text{if } var(F) \subseteq \{\overrightarrow{x_n}\} \\ (F[y \mapsto true] \vee F[y \mapsto false])|_{\overrightarrow{x_n}} & \text{for some } y \in var(F) \setminus \{\overrightarrow{x_n}\} \end{cases}$$

where the formula $F[y \mapsto E]$ is obtained by replacing each occurrence of the variable y in F by E.

Now we would conceptually define the abstraction of a computed result $\vartheta \cdot v$ representing a possible result of a function call. When the computed answer ϑ shares variables with t we have a dependency between function arguments and the resulting value. Thus, w.r.t. groundness, the result value plays the same role as data variables.

In order to have an uniform treatment of arguments and results, which besides makes definitions much clearer, we augment the "signature" of $\mathbb{MGC}$ by a bottom variable. That is $\mathbb{MGC} := \{f(\overrightarrow{x_n}) \triangleright \varrho \mid f/n \in \mathcal{D},\ \overrightarrow{x_n} \in \mathcal{V} \text{ are distinct variables},\ \varrho \in \mathcal{V}_\varrho\}$. This can be seen, with a more logical eye, as explicitly considering n-ary (multi-valued) functions as

[1] As usual we also add the bottom formula *false* to obtain a complete lattice.
[2] Note that $\Gamma(\varepsilon) = true$.
[3] by using *Schröder's elimination principle*

$n+1$-ary predicates, where the result variable is a special variable that has to be treated differently from the argument variables. We prefer to use the notation $f(\overrightarrow{x_n}) \triangleright \varrho$ instead of $f(x_1, \ldots, x_n, \varrho)$ to make more evident that ϱ plays the role of the result of the call $f(\overrightarrow{x_n})$.

Actually all definitions of the previous sections are isomorphic to their counterpart with the augmented version of $\mathbb{MGC}$, thus in the following we will implicitly abuse all notations.

We can obtain the desired *groundness dependencies of computed results* abstraction α_Γ by further abstraction of ν. Namely[4]

$$\Gamma_\varrho(S) := \bigvee \{\Gamma(\sigma\{\varrho/t\}) \mid \sigma \centerdot t\text{–}v \in S,\, v \in \mathbb{T}(\mathcal{C}, \mathcal{V})\}$$

and

$$\alpha_\Gamma(\mathcal{I}^\nu) := \lambda f(\overrightarrow{x_n}) \triangleright \varrho.\, \Gamma_\varrho(\mathcal{I}^\nu(f(\overrightarrow{x_n})))|_{\overrightarrow{x_n},\varrho}$$

Analogously to what done previously we define $\mathbb{I}_{\mathbb{GR}} := \alpha_\Gamma(\mathbb{I}_{\mathbb{WERS}}) = [\mathbb{MGC} \to \mathbb{GR}] \subseteq [\mathbb{MGC} \to \mathcal{POS}]$ and $\mathbb{GR}_{\overrightarrow{x_n} \triangleright \varrho} := \Gamma_\varrho(\mathbb{ERT}_{\overrightarrow{x_n}})$. The abstraction is not surjective on $\mathcal{POS}$ because bottom variables are not bounded in the same way as program variables are.

Let γ_Γ be α_Γ adjoint. The optimal abstract version of $\mathcal{P}^\partial$, $\mathcal{P}^{gr}[\![P]\!] := \alpha_\Gamma \circ \mathcal{P}^\partial \circ \gamma_\Gamma$, is

$$\mathcal{P}^{gr}[\![P]\!]_{\mathcal{I}^{gr}} = \lambda f(\overrightarrow{x_n}) \triangleright \varrho. \bigvee_{f(\overrightarrow{t_n}) \to r \in P} (\Gamma(\{\overrightarrow{x_n}/\overrightarrow{t_n}\}) \wedge \mathcal{E}^{gr}[\![r \triangleright \varrho]\!]_{\mathcal{I}^{gr}})|_{\overrightarrow{x_n},\varrho} \tag{4.2.2}$$

where the abstract evaluation function $\mathcal{E}^{gr}$ is defined as

$$\mathcal{E}^{gr}[\![x \triangleright \varrho]\!]_{\mathcal{I}^{gr}} := \varrho \leftrightarrow x \tag{4.2.3}$$

$$\mathcal{E}^{gr}[\![\varphi(\overrightarrow{t_n}) \triangleright \varrho]\!]_{\mathcal{I}^{gr}} := \mathcal{I}^{gr}(\varphi(\overrightarrow{\varrho_n}) \triangleright \varrho) \wedge \bigwedge_{i=1}^{n} \Phi_i \qquad \overrightarrow{\varrho_n} \text{ fresh} \tag{4.2.4}$$

where

$$\Phi_i := \begin{cases} \mathcal{E}^{gr}[\![t_i \triangleright \varrho_i]\!]_{\mathcal{I}^{gr}} & \text{if } \mathcal{I}^{gr}(\varphi(\overrightarrow{\varrho_n}) \triangleright \varrho) \leq (\varrho \to \varrho_i) \text{ or} \\ & \quad t_i \in \mathcal{T}(\mathcal{C}, \mathcal{V}) \\ \mathit{true} & \text{otherwise} \end{cases}$$

Note that, by the assumptions on interpretations, for a constructor c Equation (4.2.4) boils down to

$$\mathcal{E}^{gr}[\![c(\overrightarrow{t_n}) \triangleright \varrho]\!]_{\mathcal{I}^{gr}} = (\varrho \leftrightarrow \bigwedge_{i=1}^{n} \varrho_i) \wedge \bigwedge_{i=1}^{n} \mathcal{E}^{gr}[\![t_i \triangleright \varrho_i]\!]_{\mathcal{I}^{gr}} \qquad \overrightarrow{\varrho_n} \text{ fresh}$$

[4]Note that if S has no interval ending in a value then $\Gamma_\varrho(S) = \mathit{false}$.

Example 4.2.1

Consider the prelude's ++ definition. Then

$$\mathcal{P}^{gr}[\![\texttt{++}]\!]\uparrow 1 = \left\{ xs \texttt{ ++ } ys \triangleright \varrho \mapsto xs \wedge (\varrho \leftrightarrow ys) \right.$$
$$\mathcal{P}^{gr}[\![\texttt{++}]\!]\uparrow 2 = \left\{ xs \texttt{ ++ } ys \triangleright \varrho \mapsto \varrho \leftrightarrow (xs \wedge ys) \right.$$
$$\mathcal{P}^{gr}[\![\texttt{++}]\!]\uparrow 3 = \mathcal{P}^{gr}[\![\texttt{++}]\!]\uparrow 2 = \mathcal{P}^{gr}[\![\texttt{++}]\!]\uparrow \omega$$

Thus the abstract semantics states that the result of ++ is ground if and only if both its arguments are ground.

Example 4.2.2

Consider the union of the programs of Examples 2.2.13, 2.2.21 and 3.1.4.

$$\mathcal{P}^{gr}[\![P]\!]\uparrow 1 = \mathcal{P}^{gr}[\![P]\!]\uparrow \omega = \begin{cases} x + y \triangleright \varrho \mapsto x \wedge (\varrho \leftrightarrow y) \\ leq(x, y) \triangleright \varrho \mapsto \varrho \wedge (x \vee y) \\ coin \triangleright \varrho \mapsto \varrho \\ zeros \triangleright \varrho \mapsto false \end{cases}$$

Thus the abstract semantics states that

- + will always bind its first argument to a ground term, while its result will be ground if and only if its second argument will be;
- *leq* will always return a ground result and will bind one of its two arguments to a ground term;
- *coin* will always return a ground result;
- *zeros* will not give finite results (either no results or infinite ones).

Example 4.2.3

Consider the program

```
find k ((k',v):_) = k =:= k' &> v
find k (_:xs) = find k xs
```

with the predefined interpretation

$$\begin{cases} x \texttt{ =:= } y \triangleright \varrho \mapsto \varrho \wedge x \leftrightarrow y \\ x \texttt{ &> } y \triangleright \varrho \mapsto x \wedge (\varrho \leftrightarrow y) \end{cases}$$

we have

$$\mathcal{P}^{gr}[\![P]\!]\uparrow 1 = \mathcal{P}^{gr}[\![P]\!]\uparrow \omega = \left\{ find(x, y) \triangleright \varrho \mapsto y \rightarrow (x \wedge \varrho) \right.$$

Thus the abstract semantics states that whenever the second argument of *find* is ground then the result and its first argument will become ground.

4.2.2 Modeling the computed result behavior up to a fixed depth

An interesting finite abstraction of an infinite set of constructor terms are sets of terms up to a particular depth k, which has already been used in call-pattern analysis for functional logic programs [52], the abstract diagnosis of functional programs [4] and logic programs [27] or in the abstraction of term rewriting systems [11] (with $k = 1$).

Now we show how to approximate a weak evolving result set by means of a *depth*(k) cut $\lightning_k$ which cuts terms having a depth greater than k. Terms are cut by replacing each subterm placed at depth k with a fresh variable, called cut variable and denoted with a $\hat{\bullet}$, taken from the set $\widehat{\mathcal{V}}$ (disjoint from $\mathcal{V}$). *depth*(k) terms represent each term obtained by instantiating the variables of $\widehat{\mathcal{V}}$ with ψ-terms.

We extend $\lightning_k$ to intervals of the form $\{x_1/t_1, \ldots, x_n/t_n\} \centerdot s_1\text{–}s_2$ essentially by cutting s_1, s_2 and all t_i. However, same positions in s_1 and s_2 have to be cut with same cut variables. For instance, given the interval i

$$\{x/S(x'), y/A(z, S(S(y)))\} \centerdot B(\varrho_1, S(\varrho_2), S(S(\varrho_3)), x')\text{–}B(y, S(S(z)), S(S(z)), x')$$

we have

$$i\lightning_3 = \{x/S(x'), y/A(z, S(S(\hat{x}_1)))\} \centerdot B(\varrho_1, S(\varrho_2), S(S(\hat{x}_2)), x')\text{–}B(y, S(S(\hat{x}_3)), S(S(\hat{x}_2)), x')$$
$$i\lightning_2 = \{x/S(x'), y/A(z, S(\hat{x}_1))\} \centerdot B(\varrho_1, S(\hat{x}_2), S(\hat{x}_3), x')\text{–}B(y, S(\hat{x}_2), S(\hat{x}_3), x')$$
$$i\lightning_1 = \{x/S(\hat{x}_1), y/A(\hat{x}_2, \hat{x}_3)\} \centerdot B(\hat{x}_4, \hat{x}_5, \hat{x}_6, \hat{x}_7)$$

We define the order $\leq$ for this domain by successive lifting of the order $\hat{x} \leq t$ for every ψ-term $t \in \mathbb{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho)$ and variable $\hat{x} \in \widehat{\mathcal{V}}$. First we extend $\leq$ by structural induction on the structure over terms, then we extend it to substitutions by pointwise extension and then over *depth*(k) partial computed result by pairwise extension.

Finally given two *depth*(k) weak evolving result sets $\hat{S}_1$ and $\hat{S}_2$, $\hat{S}_1 \leq \hat{S}_2$ in and only if for any $i_1 \in \hat{S}_1$ there exists a $i_2 \in \hat{S}_2$ such that $i_1 \leq i_2$. The set of all *depth*(k) weak evolving result sets ordered by $\leq$ is a complete lattice. Let $\bigvee$ be its join.

The *depth*(k) cut of a weak evolving result set S is

$$\kappa(S) := \bigvee \{\beta\lightning_k \mid \beta \in S\} \tag{4.2.5}$$

Example 4.2.4

Given the *depth*(k) weak evolving result set $S = \{\varepsilon \centerdot \varrho,\ i_1,\ i_2\}$ where

$$i_1 = \{x/S(S(y))\} \centerdot S(S(\varrho_1))\text{–}S(S(y)), \qquad i_2 = \{x/S(S(Z))\} \centerdot S(\varrho_1)\text{–}S(S(Z))$$

for $k = 2$, $\kappa(S) = \{\varepsilon \centerdot \varrho,\ \{x/S(S(\hat{x}_1))\} \centerdot S(\varrho_1)\text{–}S(S(\hat{x}_2))\}$.

It is worth noting that the join operator in Equation (4.2.5) may cause some *depth*(k) intervals to "collapse" within a greater *depth*(k) interval. Indeed, in Example 4.2.4 the *depth*(k) abstractions of α and β collapse together because $\alpha\lightning_2 \leq \beta\lightning_2$.

Supposing a $k \geq 1$, the resulting (optimal) abstract immediate consequence operator is

$$\mathcal{P}^\kappa[\![P]\!]_{\mathcal{I}^\kappa} = \lambda f(\overrightarrow{x_n}).\ \bigvee_{f(\overrightarrow{t_n}) \to r \ll P} \kappa zap^{\{\overrightarrow{x_n}/\overrightarrow{t_n}\}}_{\{\overrightarrow{x_n}\}}(\mathcal{E}^\kappa[\![r]\!]_{\mathcal{I}^\kappa})$$

where κzap is the extension on sets of $depth(k)$ intervals of

$$\kappa zap^{\vartheta}_{V}(\sigma \centerdot s_1 – s_2) := \begin{cases} (\sigma \centerdot s_1 – s_2)\!\downarrow_k & \text{if } s_1 \in \mathcal{V}_\varrho \\ ((\vartheta\sigma)\restriction_V \centerdot s_1 – s_2)\!\downarrow_k & \text{otherwise} \end{cases} \tag{4.2.6}$$

and the abstract evaluation of $e \in \mathcal{T}(\Sigma, \mathcal{V})$ w.r.t. an interpretation $\mathcal{I}^\kappa$, namely $\mathcal{E}^\kappa[\![e]\!]_{\mathcal{I}^\kappa}$, has the same definition of Equation (3.1.52)[5].

Example 4.2.5

Consider the program P defined by

```
from n = n : from (S n)
take 0 _ = []
take (S n) (x:xs) = x : take n xs
```

and take $k = 3$. According to the previous definition, the abstract semantics of P is reached at the third iteration:

$$\mathcal{P}^\kappa[\![P]\!]\uparrow 1 = \begin{cases} from(n) \mapsto \{\varepsilon \centerdot \varrho – n{:}\varrho_1\} \\ take(n, xs) \mapsto \{\varepsilon \centerdot \varrho, \{n/0\} \centerdot [], \{n/S(n_1), xs/x_1{:}xs'\} \centerdot \varrho_1{:}\varrho_2 – x{:}\varrho_2\} \end{cases}$$

$$\mathcal{P}^\kappa[\![P]\!]\uparrow 2 = \begin{cases} from(n) \mapsto \{\varepsilon \centerdot \varrho – n{:}S(\hat{x}){:}\varrho_1\} \\ take(n, xs) \mapsto \left\{ \begin{array}{l} \varepsilon \centerdot \varrho, \{n/0\} \centerdot [], \{n/S(n'), xs/x_1{:}xs'\} \centerdot \varrho_1{:}\varrho_2 – x_1{:}\varrho_2 \\ \{n/S(0), xs/x_1{:}xs'\} \centerdot [\varrho_1] – [x_1], \\ \{n/S(S(n')), xs/x_1{:}x_2{:}xs'\} \centerdot \varrho_1{:}\varrho_2{:}\varrho_3 – x_1{:}x_2{:}\varrho_3 \end{array} \right\} \end{cases}$$

$$\mathcal{P}^\kappa[\![P]\!]\uparrow 3 = \begin{cases} from(n) \mapsto \{\varepsilon \centerdot \varrho – n{:}S(\hat{x}_1){:}\hat{x}_2{:}\hat{x}_3\} \\ take(n, xs) \mapsto \left\{ \begin{array}{l} \varepsilon \centerdot \varrho, \{n/0\} \centerdot [], \{n/S(n'), xs/x_1{:}xs'\} \centerdot \varrho_1{:}\varrho_2 – x_1{:}\varrho_2 \\ \{n/S(0), xs/x_1{:}xs'\} \centerdot [\varrho_1] – [x_1], \\ \{n/S(S(n')), xs/x_1{:}x_2{:}xs'\} \centerdot \varrho_1{:}\varrho_2{:}\varrho_3 – x_1{:}x_2{:}\varrho_3 \\ \{n/S(S(0)), xs/x_1{:}x_2{:}xs'\} \centerdot \varrho_1{:}\varrho_2{:}[] – x_1{:}x_2{:}[] \\ \{n/S(S(S(\hat{n}))), xs/x_1{:}x_2{:}\hat{x}_3{:}\hat{xs}'\} \centerdot \varrho_1{:}\varrho_2{:}\hat{y}_1{:}\hat{y}_2 – x_1{:}x_2{:}\hat{y}_3{:}\hat{y}_4 \end{array} \right\} \end{cases}$$

$$\mathcal{P}^\kappa[\![P]\!]\uparrow 4 = \mathcal{P}^\kappa[\![P]\!]\uparrow 3 = \mathcal{F}^\kappa[\![P]\!]$$

Example 4.2.6

Consider the program P_+ of Example 2.2.13. For $k = 1$

$$\mathcal{F}^\kappa[\![P_+]\!] = \Big\{ x + y \mapsto \{\varepsilon \centerdot \varrho,\ \{x/Z\} \centerdot y,\ \{x/S(\hat{x}_1)\} \centerdot S(\hat{x}_2)\}$$

This is essentially the same result that can be obtained by [11], with the *head* upper-closure operator, that derives the following abstract TRS:

$$Z^a +^a 0^a \to Z^a \qquad Z^a +^a S^a(\top_{nat}) \to S^a(\top_{nat})$$
$$S^a(\top_{nat}) +^a Z^a \to S^a(\top_{nat}) \qquad S^a(\top_{nat}) +^a S^a(\top_{nat}) \to S^a(\top_{nat})$$

Note that in our semantics the first two rules of the abstract TRS are subsumed by $\{x/Z\}.y$ while the remaining rules are subsumed by $\{x/S(\hat{x}_1)\} \centerdot S(\hat{x}_2)$.

[5] Note that the definition implicitly treats cut variables as data variables.

Example 4.2.7

Consider the program P defined by

```
main = diff (pair (sub2 (S (Z ? S Z))))
diff (Z, S x) = True
pair x = (x,x)
sub2 (S (S x)) = x
```

For $k = 2$, the abstract analysis of P reaches the fixpoint in 3 steps, giving

$$\begin{cases} sub2(x) \mapsto \{\varepsilon \centerdot \varrho,\ \{x/S(S(\hat{y}))\} \centerdot x'\} \\ pair(x) \mapsto \{\varepsilon \centerdot \varrho\text{–}(x,x)\} \\ diff(x) \mapsto \{\varepsilon \centerdot \varrho,\ \{x/(Z,S(\hat{x}))\} \centerdot True\} \\ main \mapsto \{\varepsilon \centerdot \varrho\} \end{cases}$$

For the same program and same k [52] reaches the call pattern $disj((\top,\top)) \doteq True$ causing $main \doteq True$ to be observed, which does not correspond to a concrete call pattern. However, for $k \geq 3$, this false call pattern is no more observed.

It is worth noting that the resulting abstract semantics encompasses some limitations of previous works

- Since we use a "truly" goal-independent concrete semantics we obtain a much more compact abstract semantics than [4].
- For $k = 1$ if we consider TRS admissible to apply the technique of [11] we obtain the same results. However the abstract rewriting methodology of [11] requires canonicity, stratification, constructor discipline, and complete definedness for the analyses. This class of TRS is very restricted (even for functional programming) and certainly cannot cover functional logic programs. On the contrary we require only left linearity and construction basedness.
- Since we use a careful definition of the abstraction function that uses cut variables instead of just a single $\top$ symbol like does [52] we have some slightly better results. For the same k we do not produce all false answers which are produced by [52]. These answers won't be generated by [52] for $k + 1$, but due to the quickly growing size of $depth(k)$ our improvement can be worthy.

4.3 Discussion of the results

We formulated a parametric abstraction scheme for systematically deriving goal-independent bottom-up fixpoint (approximate) abstract semantics from the weak evolving result set semantics introduced in Subsection 3.1.3. In particular we have shown two specific instances of this framework which model interesting properties of the computed result behavior of Curry programs, such as the groundness behavior of computed result, and the computed result behavior w.r.t. usage up to a fixed depth of the results. We showed the potential of these specific instances by means of examples.

To the best of our knowledge, the abstract semantics of Subsection 4.2.1 is the first attempt of groundness analysis of computed results for lazy functional logic languages. We are aware that this first proposal is pretty rough, but the theoretical environment which we have developed provides good foundations to improve such proposal in the future.

We also compared the $depth(k)$ instance of our framework with [4, 11, 52] showing how it encompasses some limitations of these works.

Clearly in this framework we cannot handle properties like termination or observations over call patterns, namely properties which are not an abstraction of weak evolving result sets. To this end we need more concrete semantics, which could be developed by (more concrete than ν) abstractions of the small-step semantics introduced in Section 2.2. Note that if we would have opted for a direct construction of $\mathcal{F}^{\nu}$ then a different base semantics would have to be constructed from scratch.

5

Abstract Diagnosis

Abstract

We present a generic scheme for the abstract debugging of functional logic programs. We associate to programs a semantics based on a (continuous) immediate consequence operator, $\mathcal{P}[\![P]\!]$, which models correctly the powerful features of modern functional logic languages (non-deterministic, non-strict functions defined by non-confluent programs and call-time choice behaviour). Then, we develop an effective debugging methodology which is based on abstract interpretation: by approximating the intended specification of the semantics of P we derive a finitely terminating bottom-up diagnosis method, which can be used statically. Our debugging framework does not require the user to provide error symptoms in advance and is applicable with partial specifications and even partial programs.

Finding program bugs is a long-standing problem in software development, even for highly expressive declarative languages. In the field of lazy functional programming, the problem is well known since the mid eighties because, as pointed out by [71, 18], the execution flow is, in general, hard to predict from the program code because of laziness. This makes these languages problematic to debug using conventional debugging tools such as breakpoints, tracing and variable watching.

There has been a lot of work on declarative debugging for functional logic languages, amongst all [2, 19, 14, 17, 72, 18], mostly following the declarative debugging approach.

Declarative debugging is a semi-automatic debugging technique where the debugger tries to locate the node in a computation tree which is ultimately responsible for a visible bug symptom. This is done by asking questions on correctness of solutions to the user, which assumes the role of the oracle. As noted by [2, 75], when debugging real code, the questions are often textually large and may be difficult to answer.

Abstract diagnosis for Logic Programs [27, 26, 28] is a framework parametric w.r.t. an abstract program property which can be considered as an extension of declarative debugging since there are instances of the framework that deliver the same results. It is based on the use of an immediate consequence operator T_P to identify bugs in logic programs. The framework is goal independent and does not require the determination of symptoms in advance.

In this chapter, we develop an abstract diagnosis method for functional logic programs using the ideas of [27]. The technique is *inherently* based on the use of an immediate consequence operator, therefore the (top-down) operational semantics for functional logic languages in the literature are not suited for this purpose. Thus we make use of the

abstraction framework of Chapter 4 for producing an "abstract immediate consequence operator" $\mathcal{P}^\alpha[\![P]\!]$ by approximation the concrete $\mathcal{P}^\nu[\![P]\!]$ operator of Subsection 3.1.3.

We show that, given the abstract intended specification $\mathcal{S}^\alpha$ of the semantics of a program P, we can check the correctness of P by a single application of $\mathcal{P}^\alpha[\![P]\!]$ and thus, by a simple static test, we can determine all the rules which are wrong w.r.t. the considered abstract property.

The diagnosis is based on the detection of *incorrect rules* and *uncovered elements*, which both have been defined in terms of one application of $\mathcal{P}^\alpha[\![P]\!]$ to the abstract specification. It is worth noting that no fixpoint computation is required, since the abstract semantics does not need to be computed. The key issue of this approach is the goal-independence of the concrete semantics, meaning that the semantics is defined by collecting the observable properties about "most general" calls, while still providing a complete characterization of the program behavior.

Among other valuable facilities, this debugging approach supports the development of efficacious diagnostic tools that detect program errors without having to determine symptoms in advance. By using suitable abstract domains several details of the computation can be hidden and thus the information that is required to the user about the (abstract) intended behaviour can be dramatically reduced. Obviously if we use more abstract domains we can detect less errors: for example an erroneous integer value cannot be detected by looking at groundness information. The choice of an abstract domain is thus a trade-off between the precision of errors that can be detected and the effort in providing the specification.

5.1 Declarative debugging of functional logic programs

Declarative debugging algorithms were first developed for diagnosing wrong and missing answers in Prolog [89]. More recently the ideas have been adapted to diagnosing errors in lazy functional languages [74, 70, 72] and functional logic languages [19, 14, 17, 18].

In [71], Lee Naish proposed a simple but very general scheme for declarative debugging which can be applied to multiple classes of bugs in a variety of languages. The debugging scheme assumes that any terminated computation can be represented as a finite tree, called *computation tree*. The root of this tree corresponds to the result of the main computation, and each node corresponds to the result of some intermediate subcomputation. Moreover it is assumed that the result at each node is determined by the results of the children nodes. Therefore, every node can be seen as the outcome of a single computation step.

Diagnosis proceeds by traversing the computation tree, asking questions to an external oracle (generally the user) looking for *erroneous nodes*. Notice, however, that an erroneous node which has some erroneous child does not necessarily correspond to a wrong computation step, since it can have some erroneous children which actually cause the symptom. In oder to avoid unsoundness, the debugging scheme continues the traversing until a so-called *buggy node* is found, that is a node whose result is erroneous, but its children have all correct results. The user does not need to understand detailed operational information about the computation. Any buggy node represents an erroneous computation step, and the debugger can display the program fragment responsible for it.

It often happens that an error symptom is caused by more than one bug. In such cases it is necessary to reiterate the application of the scheme to detect all the bugs related to

the same error symptom. For instance, suppose that a node corresponds truly to a wrong computation but it is not a buggy node, in a first stage it is not returned by the debugger, however, after fixing the bugs corresponding to the buggy nodes, it eventually becomes a buggy node which can be detected by using again the debugger.

The known declarative debuggers can be understood as concrete instances of the debugging scheme. Each particular instance is determined by three parameters: (i) a notion of computation tree, (ii) a notion of erroneous node, (iii) a method to extract a buggy program fragment from a buggy node. The choice of these parameters depends on the programming paradigm and the kind of errors to be debugged.

5.3 Abstract diagnosis of functional logic programs

Now, following the approach of [27], we define abstract diagnosis of functional logic programs. The framework of abstract diagnosis [27] comes from the idea of considering the abstract versions of Park's Induction Principle [80][1]. It can be considered as an extension of declarative debugging since there are instances of the framework that deliver the same results. However, in the general case, diagnosing w.r.t. *abstract* program properties relieves the user from having to specify in excessive detail the program behavior (which could be more error-prone than the coding itself).

There have been several proposals about Declarative Diagnosis of functional logic languages, like [72, 18, 17, 19]. The approach has revealed much more problematic in this paradigm than in the logic paradigm. As can be read from [18] "A more practical problem with existing debuggers for lazy functional (logic) languages is related to the presentation of the questions asked to the oracle". Actually the call-time choice model and the peculiarity of the needed narrowing strategy cannot be tackled by pure declarations about expected computed results. Roughly speaking, the oracle must "understand" neededness.

As already noted by [18], the "simplification" proposed in [72] to adopt the generic approach to FLP is not well-suited. [18] aims at a debugging method that asks the oracle about computed answers that do not involve function calls, plus possible occurrences of an "undefined symbol". However this is exactly the kind of information we have in our concrete semantics, which then makes it a suitable starting point for our diagnosis methodology.

In the following, $\mathcal{P}^{\nu}$ plays the role of the concrete semantic operator and is referred to as $\mathcal{P}$ omitting ν. $\mathcal{S}^{\alpha}$ is the specification of the intended behavior of a program w.r.t. the property α.

Definition 5.3.1 *Let P be a program, α be a property over domain $\mathbb{A}$ and $\mathcal{S}^{\alpha} \in \mathbb{A}$.*

1. *P is (abstractly)* partially correct *w.r.t. $\mathcal{S}^{\alpha}$ if $\alpha(\mathcal{F}[\![P]\!]) \leq \mathcal{S}^{\alpha}$.*
2. *P is (abstractly)* complete *w.r.t. $\mathcal{S}^{\alpha}$ if $\mathcal{S}^{\alpha} \leq \alpha(\mathcal{F}[\![P]\!])$.*
3. *P is* totally correct *w.r.t. $\mathcal{S}^{\alpha}$, if it is partially correct and complete.*

It is worth noting that the above definition is given in terms of the abstraction of the concrete semantics $\alpha(\mathcal{F}[\![P]\!])$ and not in terms of the (possibly less precise) abstract semantics $\mathcal{F}^{\alpha}[\![P]\!]$. $\mathcal{S}^{\alpha}$ is the abstraction of the intended concrete semantics of P. Thus, the user can

[1] a concept of formal verification that is undecidable in general

only reason in terms of the properties of the expected concrete semantics without being concerned with (approximate) abstract computations. Note also that our notion of total correctness does not concern termination (as well as finite failures). We cannot address termination issues here, since the concrete semantics we use is too abstract.

The *diagnosis* determines the "originating" symptoms and, in the case of incorrectness, the relevant rule in the program. This is captured by the definitions of *abstractly incorrect rule* and *abstract uncovered element*.

Definition 5.3.2 *Let P be a program, R a rule and $e, \mathcal{S}^\alpha \in \mathbb{A}$.*
R is abstractly incorrect *w.r.t. $\mathcal{S}^\alpha$ if $\mathcal{P}^\alpha[\![\{R\}]\!]_{\mathcal{S}^\alpha} \not\leq \mathcal{S}^\alpha$.*
e is an uncovered element *w.r.t. $\mathcal{S}^\alpha$ if $e \leq \mathcal{S}^\alpha$ and $e \wedge \mathcal{P}^\alpha[\![P]\!]_{\mathcal{S}^\alpha} = \bot$* [2].

Informally, R is abstractly incorrect if it derives a wrong abstract element from the intended semantics. e is uncovered if there are no rules deriving it from the intended semantics. It is worth noting that checking these conditions requires one application of $\mathcal{P}^\alpha[\![P]\!]$ to $\mathcal{S}^\alpha$, while the standard detection based on symptoms would require the construction of $\alpha(\mathcal{F}[\![P]\!])$ and therefore a fixpoint computation.

It is worth noting that correctness and completeness are defined in terms of $\alpha(\mathcal{F}^\alpha[\![P]\!])$, i.e., in terms of abstraction of the concrete semantics. On the other hand, abstractly incorrect rules and abstract uncovered elements are defined directly in terms of abstract computations (the abstract immediate consequence operator $\mathcal{P}^\alpha[\![P]\!]$). In this section, we are left with the problem of formally establishing the relation between the two concepts.

Theorem 5.3.3 *If there are no abstractly incorrect rules in P, then P is partially correct w.r.t. $\mathcal{S}^\alpha$.*

Proof.
By hypothesis $\forall r \in P.\ \mathcal{P}^\alpha[\![\{r\}]\!]_{\mathcal{S}^\alpha} \leq \mathcal{S}^\alpha$. Hence $\mathcal{P}^\alpha[\![P]\!]_{\mathcal{S}^\alpha} \leq \mathcal{S}^\alpha$, i.e., $\mathcal{S}^\alpha$ is a pre-fixpoint of $\mathcal{P}^\alpha[\![P]\!]$. Since $\alpha(\mathcal{F}[\![P]\!]) \leq \mathcal{F}^\alpha[\![P]\!] = \mathit{lfp}\, \mathcal{P}^\alpha[\![P]\!]$, by Knaster-Tarski's Theorem $\alpha(\mathcal{F}[\![P]\!]) \leq \mathcal{F}^\alpha[\![P]\!] \leq \mathcal{S}^\alpha$. The thesis follows by Point 1 of Definition 5.3.1.

Theorem 5.3.4 *Let P be partially correct w.r.t. $\mathcal{S}^\alpha$. If P has abstract uncovered elements then P is not complete.*

Proof.
By construction $\alpha \circ \mathcal{P}[\![P]\!] \circ \gamma \leq \mathcal{P}^\alpha[\![P]\!]$, hence $\alpha \circ \mathcal{P}[\![P]\!] \circ \gamma \circ \alpha \leq \mathcal{P}^\alpha[\![P]\!] \circ \alpha$. Since $id \sqsubseteq \gamma \circ \alpha$, it holds that $\alpha \circ \mathcal{P}[\![P]\!] \leq \alpha \circ \mathcal{P}[\![P]\!] \circ \gamma \circ \alpha$ and $\alpha \circ \mathcal{P}[\![P]\!] \leq \mathcal{P}^\alpha[\![P]\!] \circ \alpha$. Hence,

$\alpha(\mathcal{F}[\![P]\!]) =$	[since $\mathcal{F}[\![P]\!]$ is a fixpoint]
$\alpha(\mathcal{P}[\![P]\!]_{\mathcal{F}[\![P]\!]}) \leq$	[by $\alpha \circ \mathcal{P}[\![P]\!] \leq \mathcal{P}^\alpha[\![P]\!] \circ \alpha$]
$\mathcal{P}^\alpha[\![P]\!]_{\alpha(\mathcal{F}[\![P]\!])} \leq$	[since $\mathcal{P}^\alpha[\![P]\!]$ is monotone and P is partially correct]
$\mathcal{P}^\alpha[\![P]\!]_{\mathcal{S}^\alpha}$	

Now, if P has an abstract uncovered element e i.e., $e \leq \mathcal{S}^\alpha$ and $e \wedge \mathcal{P}^\alpha[\![P]\!]_{\mathcal{S}^\alpha} = \bot$, then $e \wedge \alpha(\mathcal{F}[\![P]\!]) = \bot$ and $\mathcal{S}^\alpha \not\leq \alpha(\mathcal{F}[\![P]\!])$. The thesis follows from Point 2 of Definition 5.3.1.

[2] Note that $e \wedge \mathcal{P}^\alpha[\![P]\!]_{\mathcal{S}^\alpha} = \bot$ implies $e \not\leq \mathcal{P}^\alpha[\![P]\!]_{\mathcal{S}^\alpha}$, but the converse is not true. Thus this definition is meant to detect "atomic" uncovered elements.

Abstract incorrect rules are in general just a warning about a possible source of errors. Because of the approximation, it can happen that a (concretely) correct rule is abstractly incorrect.

However, as shown by the following theorems, all concrete errors — that are "visible"[3] in the abstract domain — are detected as they lead to an abstract incorrectness or abstract uncovered.

Theorem 5.3.5 *Let r be a rule, $\mathcal{S}$ a concrete specification. If $\mathcal{P}[\![\{r\}]\!]_{\mathcal{S}} \not\sqsubseteq \mathcal{S}$ and $\alpha(\mathcal{P}[\![\{r\}]\!]_{\mathcal{S}}) \not\leq \alpha(\mathcal{S})$ then r is abstractly incorrect w.r.t. $\alpha(\mathcal{S})$.*

Proof.
Since $\mathcal{S} \sqsubseteq \gamma \circ \alpha(\mathcal{S})$, by monotonicity of α and the correctness of $\mathcal{P}^{\alpha}[\![\{r\}]\!]$, it holds that $\alpha(\mathcal{P}[\![\{r\}]\!]_{\mathcal{S}}) \leq \alpha(\mathcal{P}[\![\{r\}]\!]_{\gamma\circ\alpha(\mathcal{S})}) \leq \mathcal{P}^{\alpha}[\![\{r\}]\!]_{\alpha(\mathcal{S})}$. By hypothesis $\alpha(\mathcal{P}[\![\{r\}]\!]_{\mathcal{S}}) \not\leq \alpha(\mathcal{S})$, therefore $\mathcal{P}^{\alpha}[\![\{r\}]\!]_{\alpha(\mathcal{S})} \not\leq \alpha(\mathcal{S})$, since $\alpha(\mathcal{P}[\![\{r\}]\!]_{\mathcal{S}}) \leq \mathcal{P}^{\alpha}[\![\{r\}]\!]_{\alpha(\mathcal{S})}$. The thesis holds by Definition 5.3.2.

Theorem 5.3.6 *Let $\mathcal{S}$ be a concrete specification. If there exists an abstract uncovered element a w.r.t. $\alpha(\mathcal{S})$, such that $\gamma(a) \sqsubseteq \mathcal{S}$ and $\gamma(\bot) = \bot$, then there exists a concrete uncovered element e w.r.t. $\mathcal{S}$ (i.e., $e \sqsubseteq \mathcal{S}$ and $e \sqcap \mathcal{P}[\![P]\!]_{\mathcal{S}} = \bot$).*

Proof.
By hypothesis $a \leq \alpha(\mathcal{S})$ and $a \wedge \mathcal{P}^{\alpha}[\![P]\!]_{\alpha(\mathcal{S})} = \bot$. Hence, since $\gamma(\bot) = \bot$ and γ preserves greatest lower bounds, $\gamma(a) \sqcap \gamma(\mathcal{P}^{\alpha}[\![P]\!]_{\alpha(\mathcal{S})}) = \bot$. By construction $\mathcal{P}^{\alpha}[\![P]\!] = \alpha \circ \mathcal{P}[\![P]\!] \circ \gamma$, thus $\gamma(a) \sqcap \gamma(\alpha(\mathcal{P}[\![P]\!]_{\gamma(\alpha(\mathcal{S}))})) = \bot$. Since $id \sqsubseteq \gamma \circ \alpha$ and by monotonicity of $\mathcal{P}[\![P]\!]$, $\gamma(a) \wedge \mathcal{P}[\![P]\!]_{\mathcal{S}} = \bot$. By hypothesis $\gamma(a) \sqsubseteq \mathcal{S}$ hence $\gamma(a)$ is a concrete uncovered element.

The diagnosis w.r.t. approximate properties over Noetherian domains is always effective, because the abstract specification is finite. However, as one can expect, the results may be weaker than those that can be achieved on concrete domains just because of approximation. Namely,

- absence of abstractly incorrect rules implies partial correctness,
- every incorrectness error is identified by an abstractly incorrect rule. However an abstractly incorrect rule does not always correspond to a bug.
- every uncovered is identified by an abstract uncovered. However an abstract uncovered does not always correspond to a bug.
- there exists no sufficient condition for completeness.

It is important to note that our method, since it has been derived by (properly) applying abstract interpretation techniques, is correct by construction.

Another property of our proposal, which is particularly useful for application, is that

[3] A concrete symptom is visible if its abstraction is different from the abstraction of correct answers. For example if we abstract to the length of lists, an incorrect rule producing wrong lists of the same length of the correct ones is not visible.

- it can be used with partial specifications,
- it can be used with partial programs.

Obviously one cannot detect errors in rules involving functions which have not been specified. But for the rules that involve only functions that have a specification the check can be made, even if the whole program has not been written yet. This includes the possibility of applying our "local" method to all parts of a program not involving constructs which we cannot handle (yet). With other "global" approaches such programs could not be checked at all.

We now show the *depth*(k) instance of our methodology to provide a demonstrative application of our abstract diagnosis framework. It shows some encouraging results. Thus it will be interesting in the future to experiment also with other possible instances over more sophisticated domains.

5.4 Case Studies

Now we show how we can derive an efficacious debugger by choosing suitable instances of the general framework described in Chapter 4.

In Sections 5.4.1 and 5.4.2 we consider the groundness dependencies and the *depth*(k) abstractions of Sections 4.2.1 and 4.2.2 respectively.

5.4.1 Abstract diagnosis on $\mathcal{POS}$

In this subsection we consider as abstract property the groundness abstraction α_Γ introduced in Subsection 4.2.1.

Example 5.4.1

Consider the program P obtained adding to the program P_+ of Example 2.2.13 the following buggy rule.

```
R: isZero x = x =:= (y + x) where y free
```

where the sub-call `y + x` should have been `x + y` to be correct w.r.t. the intended groundness behavior

$$\mathcal{S}^{gr} := \begin{cases} x + y \triangleright \varrho \mapsto x \wedge (\varrho \leftrightarrow y) \\ x \texttt{ =:= } y \triangleright \varrho \mapsto \varrho \wedge x \leftrightarrow y \\ isZero(x) \triangleright \varrho \mapsto x \wedge \varrho \end{cases}$$

We detect that the rule R is abstractly incorrect since

$$\mathcal{P}^{gr}[\![\{R\}]\!]_{\mathcal{S}^{gr}} = \left\{ isZero(x) \triangleright \varrho \mapsto \varrho \right. \quad \left. \not\leq \mathcal{S}^{gr} \right.$$

Example 5.4.2

Consider the program

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a)

R: lookup k (Node (k',val) l r) =
         if k == k'
         then val
         else (lookup k l) ? (lookup k r)
```

which nondeterministically searches the values in a tree associated to a given key `k`. This program is correct w.r.t. the intended specification

$$\mathcal{S}^{gr} := \begin{cases} x \texttt{ ? } y \rhd \varrho \mapsto (x \wedge y) \rightarrow \varrho \\ \texttt{if } c \texttt{ then } x \texttt{ else } y \rhd \varrho \mapsto c \wedge (\varrho \rightarrow (x \vee y)) \wedge ((x \wedge y) \rightarrow \varrho) \\ x \texttt{ == } y \rhd \varrho \mapsto x \wedge y \wedge \varrho \\ \mathit{lookup}(k, t) \rhd \varrho \mapsto k \wedge (t \rightarrow \varrho) \end{cases}$$

but the rule R is abstractly incorrect, since

$$\mathcal{P}^{gr}[\![\{R\}]\!]_{\mathcal{S}^{gr}} = \left\{ \mathit{lookup}(k, t) \rhd \varrho \mapsto \mathit{true} \right. \quad \not\leq \mathcal{S}^{gr}$$

Example 5.4.2 shows the weakness of the diagnosis w.r.t. $\mathcal{POS}$ abstraction. Indeed, due to the level of approximation, the amount of *false positives* (i.e., abstract errors not corresponding to concrete ones) is not negligible.

Example 5.4.3

Consider the program `boyer` taken from the NOFIB-Buggy collection[4]. The program contains the following buggy piece of code

```
   find vid [] = (False, error)
R: find vid1 ((vid2,val2):bs) =
         if vid1 == vid2
         then (True, val2)
         else find vid2 ((vid1,val2):bs)
```

since the sub-call `find vid2 ((vid1,val2):bs)` in R should have been `find vid1 bs` to be correct. Providing as intended semantics on $\mathcal{POS}$ the specification

$$\mathcal{S}^{gr} := \begin{cases} \mathit{error} \rhd \varrho \mapsto \varrho \\ \texttt{if } c \texttt{ then } x \texttt{ else } y \rhd \varrho \mapsto c \wedge (\varrho \rightarrow (x \vee y)) \wedge ((x \wedge y) \rightarrow \varrho) \\ x \texttt{ == } y \rhd \varrho \mapsto x \wedge y \wedge \varrho \\ \mathit{find}(v, xs) \rhd \varrho \mapsto (v \vee xs) \wedge (xs \rightarrow \varrho) \end{cases}$$

we detect that R is abstractly correct, since

$$\mathcal{P}^{gr}[\![\{R\}]\!]_{\mathcal{S}^{gr}} = \left\{ \mathit{find}(v, xs) \rhd \varrho \mapsto (v \vee xs) \wedge (xs \rightarrow \varrho) \right. \quad \leq \mathcal{S}^{gr}$$

This happens because the concrete incorrectness symptoms caused by the bug do not affect the groundness behavior of the program[5].

---

[4] Available at http://einstein.dsic.upv.es/nofib-buggy.

[5] In Example 5.4.7 we will see that with the $\mathit{depth}(k)$ abstraction we can tackle this bug.

5.4.2 Abstract diagnosis on $depth(k)$

In this section we consider the $depth(k)$ abstraction κ introduced in Subsection 4.2.2. Note that, $depth(k)$ partial computed results that do not contain variables belonging to $\widehat{\mathcal{V}}$ are concrete partial computed results. An additional benefit w.r.t. the general outcomes is that errors exhibiting symptoms without cut variables are concrete (hence, real) errors (see Theorem 5.3.5).

Example 5.4.4

Consider the buggy program for `from` (proposed in [17]) with rule R: `from n = n : from n` and as intended specification $\mathcal{S}^\kappa$ the fixpoint from Example 4.2.5 obtained with $k = 3$. We detect that rule R is abstractly incorrect since

$$\mathcal{P}^\kappa[\![\{R\}]\!]_{\mathcal{S}^\kappa} = \Big\{ from(n) \mapsto \{\varepsilon \centerdot \varrho\text{–}n : n : \hat{x}_1 : \hat{x}_2\} \Big\} \not\trianglelefteq \mathcal{S}^\kappa$$

Example 5.4.5

Consider the buggy program P_{bug}

```
     main = C (h (f x)) x
     h (S x) = Z
R:   f (S x) = S Z
```

where rule R should have been `f x = S (h x)` to be correct w.r.t. the intended semantics on $depth(k)$, with $k > 2$,

$$\mathcal{S}^\kappa = \begin{cases} f(x) \mapsto \{\varepsilon \centerdot \varrho\text{–}S(\varrho_1),\ \{x/S(x')\} \centerdot S(Z)\} \\ h(x) \mapsto \{\varepsilon \centerdot \varrho,\ \{x/S(x')\} \centerdot Z\} \\ main \mapsto \{\varepsilon \centerdot \varrho\text{–}C(Z, x)\} \end{cases}$$

This error preserves the computed result behavior both for h and f, but not for $main$. In fact, $main$ evaluates to $\varepsilon \cdot C(Z, S(x'))$. Rule R is abstractly incorrect, since

$$\mathcal{P}^\kappa[\![\{R\}]\!]_{\mathcal{S}^\kappa} = \Big\{ f(x) \mapsto \{\varepsilon \centerdot \varrho, \{x/S(x')\} \centerdot S(\varrho_1)\text{–}S(Z)\} \Big\} \not\trianglelefteq \mathcal{S}^\kappa$$

Note that the diagnosis method of [19, 17] does not report incorrectness errors (as there are no incorrectness symptoms) while it reports the missing answer $\varepsilon \cdot C(Z, Z)$.

Example 5.4.6

Consider the following buggy program for $double$ w.r.t. the $depth(2)$ specification $\mathcal{S}^\kappa$.

```
R1: double Z   = S Z
R2: double (S x) = S (double x)
```

$$\mathcal{S}^\kappa := \left\{ double(x) \mapsto \left\{ \begin{array}{l} \varepsilon \centerdot \varrho,\ \{x/Z\} \centerdot Z,\ \{x/S(x')\} \centerdot S(\varrho_1)\text{–}S(S(\hat{y})), \\ \{x/S(Z)\} \centerdot S(S(\hat{y})),\ \{x/S(S(\hat{y}_1))\} \centerdot S(S(\hat{y}_2)) \end{array} \right\} \right.$$

We can detect that both R_1 and R_2 are abstractly incorrect, since

$$\mathcal{P}^\kappa[\![\{R_1\}]\!]_{\mathcal{S}^\kappa}(double(x)) = \{\varepsilon \centerdot \varrho,\ \{x/Z\} \centerdot S(\varrho_1)\text{–}S(Z)\}$$

$$\mathcal{P}^{\kappa}[\![\{R_2\}]\!]_{\mathcal{S}^{\kappa}}(double(x)) = \left\{ \begin{array}{l} \varepsilon \centerdot \varrho,\ \{x/S(x')\} \centerdot S(\varrho_1),\ \{x/S(Z)\} \centerdot S(Z), \\ \{x/S(S(\hat{y}_1))\} \centerdot S(S(\hat{y}_2)) \end{array} \right\}$$

However the debugger in [4] is not able to determine for $k = 2$ that rule R_2 is incorrect. This is because the errors in both the rules interfere with each other, making the goal `double (S Z)` asymptomatic.

Example 5.4.7

Consider the buggy program `boyer` of Example 5.4.3 and as intended $depth(4)$ specification $\mathcal{S}^{\kappa}$ defined by

$$\left\{ \begin{array}{l} error \mapsto \{\varepsilon \centerdot \varrho\text{–}Error\} \\ \texttt{if}\ c\ \texttt{then}\ x\ \texttt{else}\ y \mapsto \{\varepsilon \centerdot \varrho,\ \{c/True\} \centerdot x,\ \{c/False\} \centerdot y\} \\ x\ \texttt{==}\ y \triangleright \varrho \mapsto \left\{ \begin{array}{l} \varepsilon \centerdot \varrho,\ \{x/A, y/A\} \centerdot True,\ \{x/B, y/B\} \centerdot True, \\ \{x/A, y/B\} \centerdot False,\ \{x/B, y/A\} \centerdot False \end{array} \right\} \\ find(v, xs) \triangleright \varrho \mapsto \left\{ \begin{array}{l} \varepsilon \centerdot \varrho,\ \{xs/[]\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error), \\ \{v/A, xs/(A, x) : xs'\} \centerdot (\varrho_1, \varrho_2)\text{–}(True, x), \\ \{v/B, xs/(B, x) : xs'\} \centerdot (\varrho_1, \varrho_2)\text{–}(True, x), \\ \{v/A, xs/(B, x) : []\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error), \\ \{v/B, xs/(A, x) : []\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error), \\ \{v/A, xs/(B, x_1) : (A, x_2) : xs'\} \centerdot (\varrho_1, \varrho_2)\text{–}(True, x_2), \\ \{v/B, xs/(A, x_1) : (B, x_2) : xs'\} \centerdot (\varrho_1, \varrho_2)\text{–}(True, x_2), \\ \{v/A, xs/(B, x_1) : (B, x_2) : []\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error), \\ \{v/B, xs/(A, x_1) : (A, x_2) : []\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error), \\ \{v/A, xs/(B, x_1) : (B, x_2) : (\hat{y}_1, \hat{y}_2) : x'_s\} \centerdot (\varrho_1, \varrho_2)\text{–}(True, x_3), \\ \{v/B, xs/(A, x_1) : (A, x_2) : (\hat{y}_1, \hat{y}_2) : x'_s\} \centerdot (\varrho_1, \varrho_2)\text{–}(True, x_3), \\ \{v/A, xs/(B, x_1) : (B, x_2) : (\hat{y}_1, \hat{y}_2) : []\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error), \\ \{v/B, xs/(A, x_1) : (A, x_2) : (\hat{y}_1, \hat{y}_2) : []\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error), \\ \{v/A, xs/(B, x_1) : (B, x_2) : (\hat{y}_1, \hat{y}_2) : \hat{y}_3 : \hat{y}_4\} \centerdot (\varrho_1, \varrho_2)\text{–}(True, x_4), \\ \{v/B, xs/(A, x_1) : (A, x_2) : (\hat{y}_1, \hat{y}_2) : \hat{y}_3 : \hat{y}_4\} \centerdot (\varrho_1, \varrho_2)\text{–}(True, x_4), \\ \{v/A, xs/(B, x_1) : (B, x_2) : (\hat{y}_1, \hat{y}_2) : \hat{y}_3 : \hat{y}_4\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error), \\ \{v/B, xs/(A, x_1) : (A, x_2) : (\hat{y}_1, \hat{y}_2) : \hat{y}_3 : \hat{y}_4\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error) \end{array} \right\} \end{array} \right\}$$

In contrast to Example 5.4.3, using the $depth(k)$ abstraction we find that rule R is abstractly incorrect since $\mathcal{P}^{\kappa}[\![\{R\}]\!]_{\mathcal{S}^{\kappa}}(find(v, xs))$ is

$$\left\{\begin{array}{l}
\varepsilon \centerdot \varrho,\ \{v/A, xs/(A,x) : xs'\} \centerdot (\varrho_1, \varrho_2)\text{–}(True, x),\\
\{v/B, xs/(B,x) : xs'\} \centerdot (\varrho_1, \varrho_2)\text{–}(True, x),\\
\{v/A, xs/(B,x) : []\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error),\\
\{v/B, xs/(A,x) : []\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error),\\
\{v/B, xs/(A,x_1) : (B,x_2) : []\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error),\\
\{v/A, xs/(B,x_1) : (A,x_2) : []\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error),\\
\{v/B, xs/(A,x_1) : (B,x_2) : (\hat{y}_1, \hat{y}_2) : xs'\} \centerdot (\varrho_1, \varrho_2)\text{–}(True, x_3),\\
\{v/A, xs/(B,x_1) : (A,x_2) : (\hat{y}_1, \hat{y}_2) : xs'\} \centerdot (\varrho_1, \varrho_2)\text{–}(True, x_3),\\
\{v/B, xs/(A,x_1) : (B,x_2) : (\hat{y}_1, \hat{y}_2) : []\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error),\\
\{v/A, xs/(B,x_1) : (A,x_2) : (\hat{y}_1, \hat{y}_2) : []\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error),\\
\{v/B, xs/(A,x_1) : (B,x_2) : (\hat{y}_1, \hat{y}_2) : \hat{y}_3 : \hat{y}_4\} \centerdot (\varrho_1, \varrho_2)\text{–}(True, x_4),\\
\{v/A, xs/(B,x_1) : (A,x_2) : (\hat{y}_1, \hat{y}_2) : \hat{y}_3 : \hat{y}_4\} \centerdot (\varrho_1, \varrho_2)\text{–}(True, x_4),\\
\{v/B, xs/(A,x_1) : (B,x_2) : (\hat{y}_1, \hat{y}_2) : \hat{y}_3 : \hat{y}_4\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error),\\
\{v/A, xs/(B,x_1) : (A,x_2) : (\hat{y}_1, \hat{y}_2) : \hat{y}_3 : \hat{y}_4\} \centerdot (\varrho_1, \varrho_2)\text{–}(False, Error)
\end{array}\right\}$$

This example is particularly interesting, because the bug introduces non-termination, thus preventing declarative debuggers to be applied. Our methodology is not affected by this kind of problems because it does not compute fixpoints, and does not need any symptom in advance.

Example 5.4.8

The Haskell program `clausify` taken from the NOFIB-Buggy collection, contains the following buggy piece of code

```
data Formula = Sym Char | Not Formula
        | Dis Formula Formula | Con Formula Formula
        | Imp Formula Formula | Eqv Formula Formula

R1: conjunct (Con p q) = True
R2: conjunct p = True
```

Rule R should have been `conjunct p = False` to be correct w.r.t. the $depth(1)$ specification

$$\mathcal{S}^{\kappa} := \left\{ conjunct(x) \mapsto \left\{\begin{array}{l}
\varepsilon \centerdot \varrho,\ \{x/Con(p,q)\} \centerdot True,\ \{x/Sym(c)\} \centerdot False,\\
\{x/Not(p)\} \centerdot False,\ \{x/Imp(p,q)\} \centerdot False,\\
\{x/Dis(p,q)\} \centerdot False,\ \{x/Eqv(p,q)\} \centerdot False
\end{array}\right\}\right.$$

Using the conversion algorithm of Section 3.2 we obtain the semantically equivalent Curry program

```
D0: conjunct (Con p q) = True    D3: conjunct (Dis p q) = True
D1: conjunct (Sym c) = True      D4: conjunct (Imp p q) = True
D2: conjunct (Not p) = True      D5: conjunct (Eqv p q) = True
```

where rule D_0 has been generated from rule R_1, whereas rules $D_1, \ldots, D_5$ from rule R_2.

Trivially, rules $D_1, \ldots, D_5$ are abstractly incorrect w.r.t. $\mathcal{S}^\kappa$, whereas D_0 is the only abstractly correct one. Thus we can deduce that rule R_2 is abstractly incorrect.

We run our prototype on the possible benchmarks from the NOFIB-Buggy collection available at http://einstein.dsic.upv.es/nofib-buggy. Not all benchmarks can be checked yet since errors are in rules using higher-order or I/O or arithmetical features, which our semantic framework can not handle. However, since our methodology processes each rule independently, we can anyway find errors in the first-order rules of general programs, giving a partial specification. Indeed we discovered the incorrect rule in `boyer` at depth-4 (see Example 5.4.7); we detected at depth-1 the incorrect rules of `clausify`, `knights` and `lift`[6]; we found some uncovered elements for `pretty`, `sorting` and `fluid`[7] at depth-8, depth-3 and depth-2, respectively.

The case of `boyer` is particularly interesting, because the error introduces non-termination and prevents declarative debuggers to be applied. Our methodology is not concerned by this kind of errors because it does not compute fixpoints.

The selection of the appropriate depth for the abstraction is a sensitive point of this instance, since the abstraction might not be precise enough. For example if we consider $k = 2$ in Example 5.4.4 we do not detect the incorrectness. The question of whether an optimal depth exists such that no additional errors are detected by considering deeper cuts is an interesting open problem which we plan to investigate as further work.

It is important to note that the resulting abstract diagnosis encompasses some limitations of other works on declarative diagnosis:

- Symptoms involving infinite answers cannot be directly tackled by [19, 17], while, if the error manifests in the first part of the infinite answer we detect it (Example 5.4.4).
- If we just compare the actual and the intended semantics of a program some incorrectness bugs can be "hidden" because of an incompleteness bug. Our technique does not suffer of error interference (Examples 5.4.5 and 5.4.6), as each possible source of error is checked in isolation. Moreover we detect all errors simultaneously.
- It can be applied on partial programs.

5.5 Applicability of the framework

Depending on the chosen abstract domain very different scenarios arise about the pragmatic feasibility of our proposal. The main difference depends on the size of the abstract domain.

Small domains Specifications are small (and thus it is quite practical for the user to write them directly). However, due to the great amount in approximation,

[6] We exploited the conversion algorithm of Section 3.2 in order to simulate Haskell selection rule.
[7] Minor changes has been done to replace arithmetic operations with Peano's notation.

- the number of false positives (abstract errors not corresponding to concrete ones) increases;
- the number of not visible concrete errors increases.

For the former issue it is possible to adapt a multi-domain approach that uses, in addition to the first, other more concrete abstract domains. For all abstract incorrect rules detected with the first abstract domain the user can be asked to write a (partial) more concrete specification limited to the functions involved in the abstract incorrect rules. The overhead of giving a more concrete specification is just *localized* by need.

We have made some experiments on a small domain, namely the domain $\mathcal{POS}$ for groundness dependencies analysis. This experiments showed that manually writing the specification is certainly affordable in this case. However, as expected, the resulting instance it is not very powerful in detecting errors.

Big domains Due to the good precision of these domains, we can detect errors more precisely. However in these cases it is unreasonable to provide the whole specification manually. One possible solution to this issue is to compute the fixpoint of the buggy program and then present the user with the actual semantics so he can inspect it and just modify the results that he does not expect. All the explicit information given directly by the user can be saved (within the code itself) and then reused in successive sessions.

The pragmatical choice of an abstract domain is thus a tradeoff between the precision of errors that can be detected and the effort in providing the specification.

As a final remark about applicability of the current proposal beyond the first order fragment, let us note that we can tackle higher-order features and primitive operations of functional (logic) languages in the same way proposed by [52]: by using the technique known as "defunctionalization" and by approximating, very roughly, calls to primitive operations, that are not explicitly defined by rewrite rules, just with variables over $\widehat{\mathcal{V}}$.

5.6 Related Work

In particular we have considered the $depth(k)$ abstraction as a case study. $depth(k)$ is an interesting finite abstraction of an infinite set of constructor terms are sets of terms up to a particular depth k, which has already been used in call-pattern analysis for functional logic programs [52], the abstract diagnosis of functional programs [4] and logic programs [27] or in the abstraction of term rewriting systems [11] (with $k = 1$).

It is worth noting that the resulting abstract diagnosis encompasses some limitations of previous works

- The interference of two (or more) errors can be asymptomatic. The classical case is when we have an incompleteness error (something missing) and an incorrectness error that uses erroneously the missing part. Hence, until the incompleteness error is not fixed, there won't be any incorrectness symptom and declarative diagnosis cannot even start. Abstract diagnosis does not suffer of error interference, as each possible source of error is checked in isolation. Moreover it detects all errors simultaneously.

- Another relevant feature is the applicability in case of non-terminating programs. Frequently the error introduces non-termination and *prevents* declarative debuggers to be applied. Our methodology is not concerned by this kind of errors because it does not compute fixpoints and can indeed reveal the bug.
- "Real" Curry programs may contain higher-order, I/O and arithmetical features, which our semantic framework can not handle. However, since our methodology processes each rule independently, we can anyway find errors in the first-order rules of general programs, giving a partial specification. This is why we could anyway run our prototype on almost all benchmarks from the NOFIB-Buggy collection and we found most of the errors (actually all that do depend on first-order features).
- Symptoms involving infinite computed results cannot be directly tackled by declarative diagnosis, while, if the error manifests in the first part of the infinite computed result we detect it.

5.7 Discussion of the results

We formulated an efficacious generic scheme for the declarative debugging of functional logic programs based on approximating the $\mathcal{P}[\![P]\!]$ operator by means of an abstract $\mathcal{P}^\alpha[\![P]\!]$ operator obtained by abstract interpretation. Our approach is based on the ideas of [4, 27] which we apply to the diagnosis of functional logic programs. The framework of abstract diagnosis [27] comes from the idea of considering the abstract versions of Park's Induction Principle [80][8].

We showed that, given the intended abstract specification $\mathcal{S}^\alpha$ of the semantics of a program P, we can determine all the rules which are wrong w.r.t. the considered abstract property by a single application of $\mathcal{P}^\alpha[\![P]\!]$ to $\mathcal{S}^\alpha$. In the general case, diagnosing w.r.t. *abstract* program properties relieves the user from having to specify in excessive detail the program behavior (which could be more error-prone than the coding itself).

There are some features of this application that benefit from the basic semantics we have proposed in this thesis. Namely, it can be used both with partial specifications and partial programs.

For future work, we intend to derive the instances of the framework for the reduced products between $depth(k)$ and several domains typical of program analysis, like $\mathcal{POS}$, sharing, types, *etc.*. This could provide better tradeoffs between precision and size of specifications.

We also plan to develop abstract diagnosis on a more concrete base semantics, like the small-step semantics of Chapter 3, which can model "functional dependencies" (in order to tackle pre-post conditions) that could be employed to define abstract verification [24] for functional logic programs.

[8] a concept of formal verification that is undecidable in general

6

Automatic Synthesis of Specifications

Abstract

This chapter presents a technique to automatically infer algebraic property-oriented specifications from first-order Curry programs. Our technique statically infers from the source code of a Curry program a specification which consists of a set of semantic properties that the program operations satisfy. These properties are given by means of equations relating (nested) operation calls that have the same behavior. Our method relies on the semantics of Section 3.1 for achieving, to some extent, the correctness of the inferred specification, as opposed to other similar tools based on testing.

We will present the inference approach emphasizing the difficulties addressed for the case of the Curry language.

6.1 Introduction

Specifications have been widely used for several purposes: they can be used to aid (formal) verification, validation or testing, to instrument software development, as summaries in program understanding, as documentation of programs, to discover components in libraries or services in a network context, etc. [5, 83, 22, 54, 43, 96, 76, 42]. Depending on the context and the use of specifications, they can be defined, either manually or automatically, before coding (e.g. for validation purposes), during the program coding (e.g. for testing or understanding purposes), or after the code has been written (for verification or documentation). We can find several proposals of (automatic) inference of high-level specifications from an executable or the source code of a system, like [5, 22, 54, 42], which have proven to be very helpful.

There are many classifications in the literature depending on the characteristics of specifications [58]. It is common to distinguish between *property-oriented* specifications and *model-oriented* or *functional* specifications. Property-oriented specifications are of higher description level than other kinds of specifications; they consists in an indirect definition of the system's behavior by means of stating a set of properties, usually in the form of axioms that the system must satisfy [95, 94]. In other words, a specification does not represent the functionality of the program (the output of the system) but its properties in terms of relations among the operations that can be invoked in the program (i.e., different calls that have the same behavior when executed). This kind of specifications is particularly well suited for program understanding: the user can realize non-evident information about the behavior of a given function by observing its relation to other

functions. Moreover, the inferred properties can manifest potential symptoms of program errors which can be used as input for (formal) validation and verification purposes.

The task of automatically inferring program specifications has shown to be very complex. There exists a large number of different proposals and, due to the complexity of the problem, it becomes natural to impose some restrictions on the general case. Many aspects vary from one solution to another: the restriction of the considered programming language, the kind of specifications that are computed (model-oriented vs. property-oriented specifications), the kind of programs considered, etc.

We can identify two main stream approaches for the inference of specifications: glass-box and black-box. The glass-box approach [5, 22] assumes that the source code of the program is available. In this context, the goal of inferring a specification is mainly applied to document the code, or to understand it [22]. Therefore, the specification must be more succinct and comprehensible than the source code itself. The inferred specification can also be used to automatize the testing process of the program [22] or to verify that a given property holds [5]. The black-box approach [54, 42] works only by running the executable. This means that the only information used during the inference process is the input-output behavior of the program. In this setting, the inferred specification is often used to discover the functionality of the system (or services in a network) [42]. Although black-box approaches work without any restriction on the considered language—which is rarely the case in a glass-box approach—in general, they cannot *guarantee* the correctness of the results (whereas indeed semantics-based glass-box approaches can).

For this work, we took inspiration from QuickSpec [22], which is a black-box approach based on testing defined to infer algebraic property-oriented specifications from Haskell programs [82]. However, in our proposal

- we aim to infer *correct* (algebraic) property-oriented specifications. So we propose a glass-box *semantic-based* approach.

- we consider the functional logic language Curry [53, 51]. Curry is a multi-paradigm programming language that combines functional and logic programming. Because of all its very high-level features, the problem of inferring specifications for this kind of languages poses several additional problems w.r.t. other paradigms. We discuss these issues in Section 6.2.

In the rest of the chapter, we first introduce the problem of generating useful specifications for the functional logic paradigm. We illustrate the kind of specification that we compute and the problems that we have to address in our setting by means of a guiding example. Thereafter, we explain how the specifications are computed in detail. We also show some examples of specifications computed by the prototype implementing the technique and, finally, we conclude.

6.2 Property-oriented Specifications for Curry

Like QuickSpec [22], we are interested in automatically inferring program specifications as a set of equations of the form $e_1 = e_2$ where e_1, e_2 are generic program expressions that have the same computational behavior. Several theoretical (and pragmatical) questions immediately arise:

- What is the exact meaning of the equality symbol[1]?
- Which equations can we obtain?
- Are they useful for the user? Are they easy to understand?
- Which expressions should we consider?
- How can we ensure termination of the inference?

We will answer these questions throughout the chapter, but let us start first by introducing an illustrative example, which will be used both to clarify the aspects of the problem and thereafter to demonstrate the proposed technique.

Example 6.2.1 (The illustrative Queue example)
The following Curry program implements a two-sided queue where it is possible to insert or delete elements on both left and right sides:

```
data Queue a = Q [a] [a]

new = Q [] []
inl x (Q xs ys) = Q (x:xs) ys
inr x (Q xs ys) = Q xs (x:ys)
outl (Q [] ys) = Q (tail (reverse ys)) []
outl (Q (_:xs) ys) = Q xs ys
outr (Q xs []) = Q [] (tail (reverse xs))
outr (Q xs (_:ys)) = Q xs ys
null (Q [] []) = True
null (Q (_:_) _) = False
null (Q [] (_:_)) = False
```

The queue is implemented as two lists where the first list corresponds to the first part of the queue and the second list is the second part of the queue reversed. The `inl` function adds the new element to the head of the first list, whereas the `inr` function adds the new element to the head of the second list (the last element of the queue). The `outl` (`outr`) function drops one element from the left (right) list, unless the list is empty, in which case it reverses the other list and then swaps the two lists before removal.

Note that a Haskell programmer would write a different definition for the `null` function, using just one rule for the false case, namely `null _ = False`. This is wrong in Curry since, because of non-determinism, we would always have a `False` computed result in addition to the possible `True`.

In order to be useful, the specification must be concise enough, avoiding redundancies. With a comprehensible specification the user can easily learn the properties of the program from the equations, and also detect when some property is missing.

For the Queue program, one could expect a property-oriented specification with equations like

```
null new = True
```
(6.2.1)

[1] Different notions of computational behavior can be used.

```
inl y (inr x q) = inr x (inl y q)                    (6.2.2)
outr (inr x q) = outl (inl x q)                      (6.2.3)
null (inl x q) = False                               (6.2.4)
outl (inl x q) = q                                   (6.2.5)
```

Equation (6.2.2) states that `inl` and `inr` are in some sense commutative; Equation (6.2.4) says that, after inserting an element, the queue is not null; Equation (6.2.5) states that, if we execute an `outl` after an `inl`, we get the original queue back.

These equations, of the form $e_1 = e_2$, can be read as

all possible outcomes for e_1 *are also outcomes for* e_2, and vice versa. (6.2.6)

In the following, we call this equivalence *computed result equivalence* and we denote it by $=_{CR}$.

Actually, Equations (6.2.1), (6.2.2) and (6.2.3) are *literally* valid in this sense since, in Curry, free variables are admitted in expressions, and the mentioned equations are valid *as they are*. This is quite different from the purely functional case where equations *have to be interpreted* as properties that hold for any *ground* instance of the variables occurring in the equation.

On the contrary, Equations (6.2.4) and (6.2.5) are not *literally* valid because `null (inl x q)` computes $\{$`q/Q xs ys`$\} \cdot$ `False` [2] instead of just $\{\} \cdot$ `False`, and `outl (inl x q)` computes $\{$`q/Q xs ys`$\} \cdot$ `Q xs ys`, instead of just $\{\} \cdot$ `q`. Note however that, because there is only one data constructor `Q` of type `Queue a`, there is no pragmatical difference between a variable `q` (of type `Queue a`) and `Q xs ys`, where `xs`, `ys` are free variables (all ground/non-ground instances of both expressions are the same). Indeed, expression `null (inl x (Q xs ys))` computes $\{\} \cdot$ `False` and expression `outl (inl x (Q xs ys))` computes $\{\} \cdot$ `Q xs ys`. So, if we denote with `q:nonvar` the term `Q xs ys`, then the following equations, slight variations of Equations (6.2.4) and (6.2.5), now can be stated.

```
null (inl x q:nonvar) = False                        (6.2.7)
outl (inl x q:nonvar) = q:nonvar                     (6.2.8)
```

We think that *also* this kind of equations can be nevertheless interesting for the user so, in addition to the previous "strict" notion of variable in equations, we also compute equations employing this "weaker" notion. In general, the notation `q:nonvar` will denote the instantiation of variable `q` to a term constructed by applying the data constructors (of the right type) to free distinct variables. For brevity, we call this instance *non-variable free instantiation* (which is unique, up to renaming). An equality involving `:nonvar` marked variables will denote an equality which holds if we replace the variable `q` by the non-variable free instantiation.

Because of the presence of logical variables there is another very relevant difference w.r.t. the purely functional case, concerned with *contextual equivalence*: given a valid equation $e_1 = e_2$, is it true that, for any context C, the equation $C[e_1] = C[e_2]$ still holds? This may seem a nonsensical question in purely functional languages, which are

[2] The expression $\{$`q/Q xs ys`$\} \cdot$ `False` denotes that the data value `False` has been reached with computed answer substitution $\{$`q/Q xs ys`$\}$.

referentially transparent ([91]) by language definition[3]. However, Curry is not referentially transparent w.r.t. its operational behavior, i.e., an expression can produce different computed results when it is embedded in a context that binds its free variables (as shown by the following example).

Example 6.2.2

Given a program with the following rules

```
g x = C (h x)          g' A = C A
h A = A                f (C x) B = B
```

the expressions `g x` and `g' x` compute the same result, namely $\{\texttt{x/A}\} \cdot \texttt{C A}$. However, the expression `f (g x) x` computes one result, namely $\{\texttt{x/B}\} \cdot \texttt{B}$, while expression `f (g' x) x` computes none.

Thus, in the Curry case, it is (much) more appropriate to *additionally* ask in (6.2.6) that the outcomes must be equal also when the two terms are embedded within any context. We call this equivalence *contextual equivalence* and we denote it by $=_C$. Actually, Equations (6.2.1), (6.2.2), (6.2.3), (6.2.7) and (6.2.8) are valid also in this sense.

Since $=_C$ is (obviously) stronger than $=_{CR}$, there may be equations produced by $=_{CR}$ but not produced by $=_C$ that can be nevertheless interesting for the user. So, in the following, we use both notions.

6.2.1 Our Property-oriented Specifications

Besides $=_C$ and $=_{CR}$, there are also even weaker forms of equality that we are interested in. In this section, we formally present all the kinds of term equivalence notions that are used to compute equations of the specification. We need first to introduce some basic formal notions that are used throughout the chapter.

We say that a first order Curry program is a set of rules P built over a signature Σ partitioned in $\mathcal{C}$, the *constructor* symbols, and $\mathcal{D}$, the *defined* symbols. $\mathcal{V}$ denotes a (fixed) countably infinite set of variables and $\mathcal{T}(\Sigma, \mathcal{V})$ denotes the terms built over signature Σ and variables $\mathcal{V}$. We partition $\mathcal{V}$ into two sets $\mathcal{V}_n$ and $\mathcal{V}_{nv}$ of *normal* variables and variables representing non-variable freely instantiated values. A *fresh* variable is a variable that appears nowhere else.

We evaluate first order Curry programs on the condensed, goal-independent semantics defined in Section 3.1 for functional logic programs. We do not use the traditional Curry semantics, namely the operational and the I/O semantics, because they do not fulfill referential transparency, so they are not an appropriate base semantics for computing specifications w.r.t. $=_C$. The (more elaborated) semantics of Section 3.1 instead fulfills referential transparency. Moreover, this semantics has another pragmatical property which makes it a very appropriate base semantics for our purposes: it is condensed, meaning that denotations are, to some extent, the smallest possible (between all those semantics which are fully abstract).

The denotation $\mathcal{F}[\![P]\!]$ of a program P is the least fixed-point of an immediate consequences operator $\mathcal{P}[\![P]\!]$, which is based on a term evaluation function $\mathcal{E}[\![t]\!]$ which, for any

[3]The concept of referential transparency of a language can be stated in terms of a formal semantics as: the semantics equivalence of two expressions e, e' implies the semantics equivalence of e and e' when used within any context $C[\cdot]$. Namely, $\forall e, e', C.\ [\![e]\!] = [\![e']\!] \implies [\![C[e]]\!] = [\![C[e']]\!]$.

term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, gives the semantics of t as $\mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]}$. Intuitively, the evaluation $\mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]}$ computes a set of intervals of the form $\sigma \centerdot s\text{–}s'$ which collects the "relevant history" of the computation of all computed results of t, abstracting from function calls and focusing only on the way in which the result is built. In particular, every interval $\sigma \centerdot s\text{–}v$ such that v is a data value represents a normal form of the initial term. An interval is a triple of the form $\sigma \centerdot s\text{–}s'$ where σ is a substitution binding variables of the initial expression with linear constructor terms, and s, s' are partial results, i.e., terms in $\mathbb{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho)$ that may contain special variables (ϱ_0, ϱ_1, $\ldots \in \mathcal{V}_\varrho$, a set which is disjoint from $\mathcal{V}$) indicating a non-completed evaluation (see Section 3.1). Intervals $\sigma \centerdot s\text{–}v$ whose ending term s do not contain such variables induce a computed result $\sigma \cdot v$. We will denote as $cr(S)$ the set of computed results of the semantic set S, namely $cr(S) := \{\sigma \cdot v \,|\, \sigma \centerdot s\text{–}v \in S, v \in \mathbb{T}(\mathcal{C}, \mathcal{V})\}$.

Example 6.2.3

Consider the program P of Example 6.2.1. The semantics of function `outr` in $\mathcal{F}[\![P]\!]$ is

$$outr(q) \mapsto \left\{ \begin{array}{l} \varepsilon \centerdot \varrho,\ \{q/Q(xs, y : ys)\} \centerdot Q(\varrho_1, \varrho_2)\text{–}Q(xs, ys), \\ \{q/Q(xs, [])\} \centerdot Q(\varrho_1, \varrho_2)\text{–}Q([], \varrho_2),\ \{q/Q([x_1], [])\} \centerdot Q([], []), \\ \{q/Q([x_1, x_2], [])\} \centerdot Q([], \varrho_1 : \varrho_2)\text{–}Q([], [x_2]), \\ \qquad\qquad \vdots \\ \{q/Q([x_1, \ldots, x_n], [])\} \centerdot Q([], \varrho_1 : \varrho_2)\text{–}Q([], [x_2, \ldots, x_n]), \\ \qquad\qquad \vdots \end{array} \right\}$$

Formally, the (algebraic) specifications $\mathcal{S}$ which we infer are sets of (sequences of) equations of the form $t_1 =_K t_2 =_K \ldots =_K t_n$, with $K \in \{C, CR, CBI, G\}$ and $t_1, t_2, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$. K distinguishes the kinds of computational equalities that we handle, which we now present formally.

Contextual Equivalence $=_C$. States that two terms t_1 and t_2 are equivalent if $C[t_1]$ and $C[t_2]$ have the same behavior for any context $C[\cdot]$. This is the most difficult equivalence to be established, but the use of the semantics $\mathcal{F}[\![P]\!]$ eases the task since it is fully abstract w.r.t. the contextual program behavior equivalence. Therefore, two terms t_1 and t_2 are related by the contextual relation $=_C$ if their semantics coincide, namely

$$t_1 =_C t_2 \iff \mathcal{E}[\![t_1]\!]_{\mathcal{F}[\![P]\!]} = \mathcal{E}[\![t_2]\!]_{\mathcal{F}[\![P]\!]}$$

Intuitively, due to the definition of this semantics, this means that *all the ways* in which these two terms reach their computed results coincide. Note that $=_C$ does not capture termination properties, which is out of our current scope. However, the technique that we are now proposing can work even if we have a non-terminating function, a situation in which black-box approaches cannot work.

Computed-result equivalence $=_{CR}$. This equivalence states that two terms are equivalent when the outcomes of their evaluation are the same. Therefore, this notion abstracts from the way in which the results evolve during computation.

The $=_{CR}$ equivalence is coarser than $=_C$ ($=_C \subseteq =_{CR}$) as shown by Example 6.2.2.

It is important to note that we can determine $=_{CR}$ just by collecting computed results induced by $\mathcal{E}[\![e]\!]_{\mathcal{F}[\![P]\!]}$. Namely

$$t_1 =_{CR} t_2 \iff cr(\mathcal{E}[\![t_1]\!]_{\mathcal{F}[\![P]\!]}) = cr(\mathcal{E}[\![t_2]\!]_{\mathcal{F}[\![P]\!]})$$

Constructor Based Instances Equivalence $=_{CBI}$. It states that two terms are equivalent if, for all possible instances of the variables with constructor terms $\mathcal{T}(\mathcal{C}, \mathcal{V})$, which compute the empty substitution, the computed values are the same.

This notion equalizes non-deterministic operations which produce computed results that are subsumed by other computed results of the same operation, with operations which produce just the most general results. This equivalence is coarser than $=_{CR}$.

It is worth noting that we can determine $=_{CBI}$ from the computed results collected so far, just by filtering out from them all the computed results which are not the most general one, namely

$$t_1 =_{CBI} t_2 \iff cbi \circ cr(\mathcal{E}[\![t_1]\!]_{\mathcal{F}[\![P]\!]}) = cbi \circ cr(\mathcal{E}[\![t_2]\!]_{\mathcal{F}[\![P]\!]})$$

where $cbi(S^{cr}) := \{\alpha \in S^{cr} \mid \forall \beta \in (S^{cr} \setminus \{\alpha\}).\, \nexists \vartheta \in \mathcal{C}Substs.\, \beta\vartheta \simeq \alpha\}$.

Ground Equivalence $=_G$. This equivalence states that two terms are equivalent if all possible ground instances have the same outcomes. Equivalence $=_G$ is coarser than $=_{CBI}$.

The last equivalence is the easiest equivalence to be established. Note that it is the only sensible notion in the purely functional paradigm. This fact allows one to have an intuition of the reason why the problem of specification synthesis is more complex in the functional logic paradigm.

To summarize, we have $=_C \subseteq =_{CR} \subseteq =_{CBI} \subseteq =_G$ and only $=_C$ is referentially transparent (i.e., a congruence w.r.t. contextual embedding).

6.3 Deriving Specifications from Programs

Now we are ready to describe the process of inferring specifications. The input of the process consists of the Curry program to be analyzed and two additional parameters: a *relevant* API, Σ^r, and a maximum term size, *max_size*. The *relevant* API allows the user to choose the operations in the program that will be present in the inferred specification, whereas the maximum term size limits the size of the terms in the specification. As a consequence, these two parameters tune the granularity of the specification, both making the process terminating and allowing the user to keep the specification concise and easy to understand.

The inference process consists of three phases, as depicted in Figure 6.1. In the following we explain in detail the phases of the inference process by referring to the pseudocode given in Algorithm 1. For the sake of comprehension, we present an untyped version of the algorithm. The actual one is a straightforward modification conformant w.r.t. types.

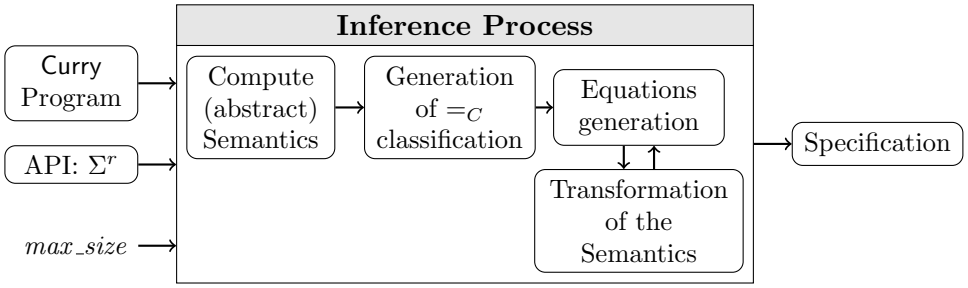


Figure 6.1: A general view of the inference process.

Computation of the abstract semantics (and initial classification). The first phase of the algorithm, Lines 1 to 2 (in Algorithm 1), is the computation of the initial classification that is needed to compute the classification w.r.t. $=_C$. It is based on the computation of an approximation of the semantics of the program (abstract semantics). The semantics of a program is in general infinite, therefore, something has to be done in order to have a terminating method. We use the $depth(k)$ abstract semantics of Subsection 4.2.2 in order to achieve termination by giving up precision of the results. We discuss this issue and, in particular, how and when we can ensure correctness at the end of the section. To improve readability, now we describe the process based on the concrete semantics of the language.

Terms are classified by their semantics into a data structure, which we call *partition*, consisting of a set of *equivalence classes* ec formed by

- $sem(ec)$: the semantics of (all) the terms in that class
- $rep(ec)$: the *representative term* of the class, which is defined as the smallest term in the class (w.r.t. the function *size*), and
- $terms(ec)$: the set of terms belonging to that equivalence class.

The *representative term* is used in order to avoid much redundancy in the generation of equations. Instead of using every term in an equivalence class to build terms of greater size, we use only the representative term.

With the program's semantics the *initial_part* function can build the initial partition which contains:

- the classes $\langle \mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]}, t, \{t\}\rangle$, for all $t = f(x_1, \ldots, x_n)$;
- one class for a free (logical) variable $\langle \mathcal{E}[\![x]\!], x, \{x\}\rangle$;
- one class for a non-variable freely instantiated value `q:nonvar` for each user-defined data type;
- the classes for any built-in or user-defined constructor.

Algorithm 1 Inference of an algebraic specification

Require: Program P; Program's *relevant* API Σ^r; Maximum term size *max_size*

1. Compute $\mathcal{F}[\![P]\!]$: the (abstract) semantics of P
2. $part \leftarrow initial\_part(\mathcal{F}[\![P]\!])$
3. **repeat**
4. $part' \leftarrow part$
5. **for all** $f/n \in \Sigma^r$ **do**
6. **for all** $ec_1, \ldots, ec_n \in part$ **do**
7. $t \leftarrow f(rep(ec_1), \ldots, rep(ec_n))$ where the $rep(ec_i)$ are renamed apart
8. **if** t not in $part$ **and** $size(t) \leq max\_size$ **then**
9. $s \leftarrow \mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]}$: Compute the (abstract) semantics of term t
10. $add\_to\_partition(t, s, part')$
11. **end if**
12. **end for**
13. **end for**
14. **until** $part' \neq part$
15. $specification \leftarrow \emptyset$; $add\_equations(specification, part)$
16. **for all** $kind \in [CR, CBI, G]$ **do**
17. $part \leftarrow transform\_semantics(kind, part)$
18. $add\_equations(specification, part)$
19. **end for**
20. **return** *specification*

Generation of $=_C$ classification. The second phase of the algorithm, Lines 3 to 14, is the computation of the classification of terms w.r.t. $=_C$. As we have said before, this is also the basis for the generation of the other kinds of equivalence classes.

We iteratively select all symbols f/n of the relevant API Σ^r (Line 5) [5] and n equivalence classes $ec_1, \ldots, ec_n$ from the current partition (Line 6). We build the term $t = f(t_1, \ldots, t_n)$, where each t_i is the representative term of ec_i, $rep(t_i)$; compute the semantics $s = \mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]}$ and update the current partition $part'$ with $add\_to\_partition(t, s, part')$ (Lines 7 to 11). Thanks to the compositionality of the semantics, and since the semantics s_i for the terms t_i are already stored in ec_i, i.e., $s_i = sem(ec_i)$, the computation of the semantics of t can be done in an efficient way, just by *nesting* the semantics s_i into the semantics of $f(x_1, \ldots, x_n)$ (this semantics nesting operation is the core of the $\mathcal{E}$ operation). Namely, $\mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]} = nest(\mathcal{E}[\![f(x_1, \ldots, x_n)]\!]_{\mathcal{F}[\![P]\!]}, sem(ec_1), \ldots, sem(ec_n))$.

The $add\_to\_partition(t, s, part)$ function looks for an equivalence class ec in $part$ whose semantics coincides with s. Then, the term t is added to the set of terms in ec. If no equivalence class is found with that semantics s, then a new equivalence class $\langle s, t, \{t\}\rangle$ is created.

If we have produced some changes in the partition, then we iterate. This phase is doomed to terminate because at each iteration we consider, by construction, terms which are different from those already existing in the partition and whose size is strictly greater than the size of its subterms (but the size is bounded by max_size).

Let us show an example:

[5] Following the standard notation from the functional and logic paradigms, f/n denotes a function f of arity n.

Example 6.3.1

Let us recall the program of Example 6.2.1 and choose as relevant API the functions `new`, `inl`, `outl` and `null`.

During the first iteration, one of the terms which is built is the term t_0 = `inl x q:nonvar` (starting from `x` and `q:nonvar` of the initial partition). Since it has a "new" semantics, it is classified by itself. We also build term t_1 = `null (q:nonvar)` whose semantics results to be equivalent to that of term the `False`, which is already stored in an equivalence class, let us call it ec_1, of the initial partition; Therefore, t_1 is added to the set of terms of ec_1. Later this generates Equation (6.2.7).

During the second iteration, we build the term t_2 = `outl` t_0, whose semantics results to be equivalent to that of `q:nonvar`, already stored in an equivalence class of the initial partition, let us call it ec_2; Thus, t_2 is included in the set of terms of ec_2. This generates Equation (6.2.8) later.

From now on the term t_2 will not be used in the construction of new terms, since it belongs to $terms(ec_2)$ but is not $rep(ec_2)$.

Although the overall strategy has been organized in order to avoid much redundancy in equations, there is one additional issue that may introduce a small amount of redundancy. Let us discuss this with an artificial example. In the second iteration we build the terms `inl x (outl q)`, which generates a new class, and `outl (inl x q)`, which for the sake of this discussion we assume it generates equation `outl (inl x q)` $=_C$ `q`. Then, in the third iteration we build the term `outl (inl x (outl q))`, whose semantics is the same of `outl q`, thus the equation `outl (inl x (outl q))` $=_C$ `outl q` is generated. However, this equation is redundant because it is an instance of `outl (inl x q)` $=_C$ `q`.

Nevertheless, we note that these redundant equations are not common (actually, the example above is not a real one, since in our example the generated equation is `q:nonvar` $=_C$ `outl (inl x q:nonvar)`). Moreover, the eventual presence of these bad equations does not propagate to other equations.

Generation of the specification. The third phase of the algorithm (Lines 15 to 19) constructs the specification for the provided Curry program. First, Line 15 computes the $=_C$ equations from the current partition. Since we have avoided much redundancy thanks to the strategy used to generate the equivalence classes, the *add_equations* function needs only to take each equivalence class with more than one term and generate equations for these terms. This function generates also a side effect on the equivalence classes that is needed in the successive steps. Namely, it replaces the (non-singleton) set of terms with a singleton set containing just the representative term.

Then, Lines 16 to 19 compute the equations corresponding to the rest of equivalence notions defined in Section 6.2.1. Let us explain in detail the case for the computed result equations (kind *CR*). As already noted, from the semantics S in the equivalence classes computed during the second phase of the algorithm, it is possible to construct (by losing the internal structure and collecting just the computed results $cr(S)$) the semantics that models the computed result behavior. Therefore, we apply this transformation to the semantic values of each equivalence class. After the transformation, some of the equivalence classes which had different semantic values may now collapse into the same class. This

transformation and reclassification is performed by the *transform_semantics* function. The resulting (coarser) partition is then used to produce the $=_{CR}$ equations by an application of *add_equations*.

Thanks to the fact that *add_equations* ends with a partition made of just singleton term sets, we cannot generate (again) equations $t_1 =_{CR} t_2$ when an equation $t_1 =_C t_2$ had been already released.

Let us clarify this phase by an example.

Example 6.3.2

Assume we have a partition consisting of three equivalence classes with semantics s_1, s_2 and s_3 and representative terms t_{11}, t_{22} and t_{31}:

$$ec_1 = \langle s_1, t_{11}, \{t_{11}, t_{12}, t_{13}\}\rangle \qquad ec_2 = \langle s_2, t_{22}, \{t_{21}, t_{22}\}\rangle \qquad ec_3 = \langle s_3, t_{31}, \{t_{31}\}\rangle$$

The *add_equations* procedure adds equations $\{t_{11} =_C t_{12} =_C t_{13}, t_{21} =_C t_{22}\}$ and the partition becomes

$$ec_1 = \langle s_1, t_{11}, \{t_{11}\}\rangle \qquad ec_2 = \langle s_2, t_{22}, \{t_{22}\}\rangle \qquad ec_3 = \langle s_3, t_{31}, \{t_{31}\}\rangle$$

Now, assume that $cr(s_1) = x_0$ and $cr(s_2) = cr(s_3) = x_1$. Then, after applying *transform_semantics*, we obtain the new partition

$$ec_4 = \langle x_0,\ t_{11},\ \{t_{11}\}\rangle \qquad ec_5 = \langle x_1,\ t_{22},\ \{t_{22}, t_{31}\}\rangle$$

Hence, the only new equation is $t_{22} =_{CR} t_{31}$. Indeed, equation $t_{11} =_{CR} t_{12}$ is uninteresting, since we already know $t_{11} =_C t_{12}$ and equation $t_{21} =_{CR} t_{31}$ is redundant (because $t_{21} =_C t_{22}$ and $t_{22} =_{CR} t_{31}$).

In summary, if $t_1 =_C t_2$ holds, then $t_1 =_{\{CR,CBI,G\}} t_2$ are not present in the specification.

The same strategy is used to generate the other two kinds of equations. Intuitively, the semantics are transformed by removing further internal structure to compute the constructor-based and ground versions. Since these relations are less precise than the previous ones, again classes may collapse and new equations (w.r.t. a different equivalence notion) are generated.

Effectivity/Efficiency considerations. In a semantic-based approach, one of the main problems to be tackled is effectiveness. The semantics of a program is in general infinite and thus we use abstract interpretation [29] in order to have a terminating method. Namely, we use the *depth*(k) abstraction defined in Subsection 4.2.2. In the case of the *depth*(k) abstraction, terms are "cut" at depth k by replacing them with cut variables, distinct from program variables. Thanks to this structure *depth*(k) semantics is technically an over approximation of the semantics, but simultaneously it contains also an under approximation, because terms which do not contain cut variables belong to the concrete semantics.

Thus, equations coming from equivalence classes whose *depth*(k) semantics does not contain cut variables are *correct* equations, while for the others we don't know (yet). If we use a bigger k, the latter can definitively become valid or not. Thus, equations involving approximation are equations that *have not been falsified up to that point*, analogously to what happens in the testing-based approach. We call these *unfalsified* equations.

The main advantage of our proposal w.r.t. the testing-based approaches is the fact that we are able to distinguish when an equation *certainly* holds, and when it just *can* hold.

Since the overall construction is (almost) independent upon the actual structure of the abstract semantics, it would be possible in the future to use other abstract domains to reach different compromises between efficiency of the computation and accuracy of the specifications.

6.4 The prototype in practice

The AbsSpec 1.0 prototype is a component of a suite of tools written in Haskell which we describe in Chapter 7. On top of the shared suite components which compute the abstract semantics, the interface module implements some functions that allow the user both to check if a specific set of equations hold, or to get the whole specification. It is worth noting that, although in the presentation we considered as input only Curry programs, the prototype also accepts programs written in (the first order fragment of) Haskell (which are automatically converted by means of the orthogonalization into Curry equivalent programs described in Section 3.2).

Let us now discuss the results for two program examples. For the Queue program of Example 6.2.1, an extract of the computed specification is the following one:

$$\texttt{null new} =_C \texttt{True} \tag{6.4.1}$$

$$\texttt{new} =_C \texttt{outl (inl x new)} =_C \texttt{outr (inr x new)}$$
$$=_C \texttt{outr (inl x new)} =_C \texttt{outl (inr x new)} \tag{6.4.2}$$

$$\texttt{outr (inl x new)} =_C \texttt{outl (inr x new)} \tag{6.4.3}$$

$$\texttt{inl y new} =_C \texttt{outl (inr y (inr x new))} \tag{6.4.4}$$

$$\texttt{inr y new} =_C \texttt{outr (inl y (inl x new))} \tag{6.4.5}$$

$$\texttt{outl (inl x q)} =_C \texttt{outr (inr x q)} \tag{6.4.6}$$

$$\texttt{q:nonvar} =_C \texttt{outl (inl x q:nonvar)} =_C \texttt{outr (inr x q:nonvar)} \tag{6.4.7}$$

$$\texttt{inl x (outl (inl y q))} =_C \texttt{outr (inl x (inr y q))} \tag{6.4.8}$$

$$\texttt{outl (inl x (outl q))} =_C \texttt{outl (outl (inl x q))} \tag{6.4.9}$$

$$\texttt{outr (outl (inl x q))} =_C \texttt{outl (inl x (outr q))} \tag{6.4.10}$$

$$\texttt{null (inl x new)} =_C \texttt{null (inr x new)} =_C$$
$$=_C \texttt{null (inl x q:nonvar)} =_C \texttt{False} \tag{6.4.11}$$

$$\texttt{outr new} =_{CR} \texttt{outl new} =_{CR} \texttt{outl q:nonvar} \tag{6.4.12}$$

Equations (6.4.1), (6.4.6), (6.4.7) and (6.4.11) are (or contain) Equations (6.2.1), (6.2.3), (6.2.8) and (6.2.7), respectively. Equation (6.2.2) does not show up because we do not handle yet semantics equivalence up to permutation of variables in the initial goals (`x` and `y` in this case).

Equation (6.4.2) , together with Equation (6.4.3), state that if we have an empty queue, adding and removing one element produces always the same result independently from the side in which we add and remove it. Equations (6.4.4) and (6.4.5) state that two successive insertions on one side of an empty queue and a deletion on the opposite side,

corresponds to an insertion of the second element on the opposite side. Equation (6.4.8) shows a sort of *restricted* commutativity between functions.

The $=_{CR}$ equations of (6.4.12) show up because all these terms have no computed results.

The second considered program example, written in Haskell, transforms a propositional formula to its conjunctive normal form implementing the classic axioms. Let us show an excerpt of the program:

```
data Sentence = Conn Sentence Conn Sentence
              | Not Sentence | Proposition String
data Conn = And | Or | Imply

toCNF, elimImpl, distAndOverOr :: Sentence -> Sentence

toCNF s = distAndOverOr (moveNotInw (elimImpl s))

elimImpl (Conn s1 Imply s2) =
  Conn (Not (elimImpl s1)) Or (elimImpl s2)

distAndOverOr (Conn (Conn s1 And s2) Or s3) =
  distAndOverOr (Conn (Conn (distrAndOverOr s1) Or
                            (distrAndOverOr s3)) And...
```

Our prototype computes equations like

```
toCNF (toCNF x) =C toCNF (elimImpl x) =C toCNF (distAndOverOr x)
 =C moveNotInw (toCNF x) =C toCNF x
moveNotInw (moveNotInw x) =CR moveNotInw x
...
```

which allows the user to see that, for instance, the function `moveNotInw` does not affect the results when applied to `toCNF`, or that it is idempotent. One can also see that the behavior of `elimImpl` and `distAndOverOr` is *subsumed* by the behavior of `toCNF`.

Our preliminary experiments show that many interesting properties hold with a low k in $depth(k)$ (we run the prototype with depth 4 by default). Moreover, although it is almost unfeasible to have more than 3 nested calls (caused by the exponential explosion of all possible combinations) the computed equations involving much nested calls tend to be little comprehensible for the user, thus they are rarely needed.

Termination and quality of correct equations strongly depends on the structure of the program. We are currently studying possible clever strategies, possibly depending on the program, that would check "more promising" terms first.

6.5 Related Work

To the best of our knowledge, in the functional logic setting there are currently no proposals for specification synthesis. There is a testing tool, EasyCheck [21], in which specifications are *used* as the input for the testing process. Given the properties, EasyCheck executes

ground tests in order to check whether the property holds. This tool could be used as a companion tool of ours in order to check the unfalsified equations.

The mentioned QuickSpec [22] computes an algebraic specification for Haskell programs by means of (almost black-box) testing. Its inferred specifications are complete up to a certain depth of the analyzed terms because of its exhaustiveness. However, the specification may be incorrect due to the use of testing for the equation generation. Instead, we follow a (glass-box) semantic-based approach that allows us to compute specifications as complete as those of QuickSpec, but with correctness guarantees on part of them (depending on the abstraction).

6.6 Discussion on the results

We presented a method to automatically infer high-level, property-oriented (algebraic) specifications in a functional logic setting. The specification represents relations that hold between operations (nested calls) in the program.

The method computes a concise specification of program properties from the source code of the program. We hope to have convinced the reader that we reached our main goal, that is to get a concise and clear specification that is useful for the programmer in order to detect possible errors, or to check that the program corresponds to the intended behavior.

The computed specification is particularly well suited for program understanding since it allows to discover non-evident behaviors, and also to be combined with testing. In the context of (formal) verification, the specification can be used to ease the verification tasks, for example by using the correct equations as annotations, or unfalsified equations as candidate axioms to be proven.

The approach relies on the computation of the semantics. Therefore, to achieve effectiveness and good performance results, we use a a suitable abstract semantics instead of the concrete one. This means that we may not guarantee correctness of all the equations in the specification, but we can nevertheless infer correct equations thanks to a good compromise between correctness and efficiency.

We have developed a prototype that implements the basic functionality of the approach. We are working on the inclusion of all the functionality described in this chaper.

As future work, we plan to add another notion of equivalence class. More specifically, when dealing with a user-defined data type, the user may have defined a specific notion of equivalence by means of an "equality" function. In this context, some interesting equations could show up. For instance, in the Queue example, a possible equality may identify queues which contain the same elements. Then, we would have the equation `inl x new = inr x new`. Note that, although the internal structure of the queue differs, the content coincides. These equivalence classes would depend on how the user defines the equality notion, thus we will surely need some assumptions (to ensure that indeed the user-defined function induces an equality relation) in order to get useful specifications.

7

Implementation

In this chapter we describe the overall architecture of the proof of concept prototype that some members of the research group have developed so far. It is written in Haskell and developed with GHC. I will describe the whole project and then highlight the components I have developed personally.

The architecture has been planned for a long term implementation project that is able to allot various tools for abstract diagnosis, abstract verification, analysis, abstract specification synthesis and (in the future) even more.

The experience of the senior group member has shown that in the development of former prototypes of abstract semantics-based program manipulation tools there are large parts of code that tend to be very similar. So they came up with the idea to have a unique collection of tools which could share all common parts, gaining all the typical benefits of such architecture (especially in term of maintainability, scalability, *etc.*).

In Figure 7.1 we can see a picture of the current system which will help for the following description.

7.1 Parser Suite description

All the (pure) declarative languages which we consider have a high variety of concrete syntax constructs, but most of them are syntactic sugar for a small core of constructs. Like in the construction of suite of parsers (e.g. GCC), we have identified an *Intermediate Common Language* which constitutes the input abstract syntax of the abstract semantics engines. Then for each different (pure) declarative languages we have a translator that converts the abstract syntax generated by the language parser to the Intermediate Common Language.

7.1.1 Intermediate Common Language

By taking inspiration from the GHC's Core Language and MCC's Intermediate Language and by analyzing the abstract syntax produced by the GHC and MCC parsers we have determined our Intermediate Common Language as a compromise between

- ability to translate comfortably the concrete syntax of
  - Curry;
  - Haskell;

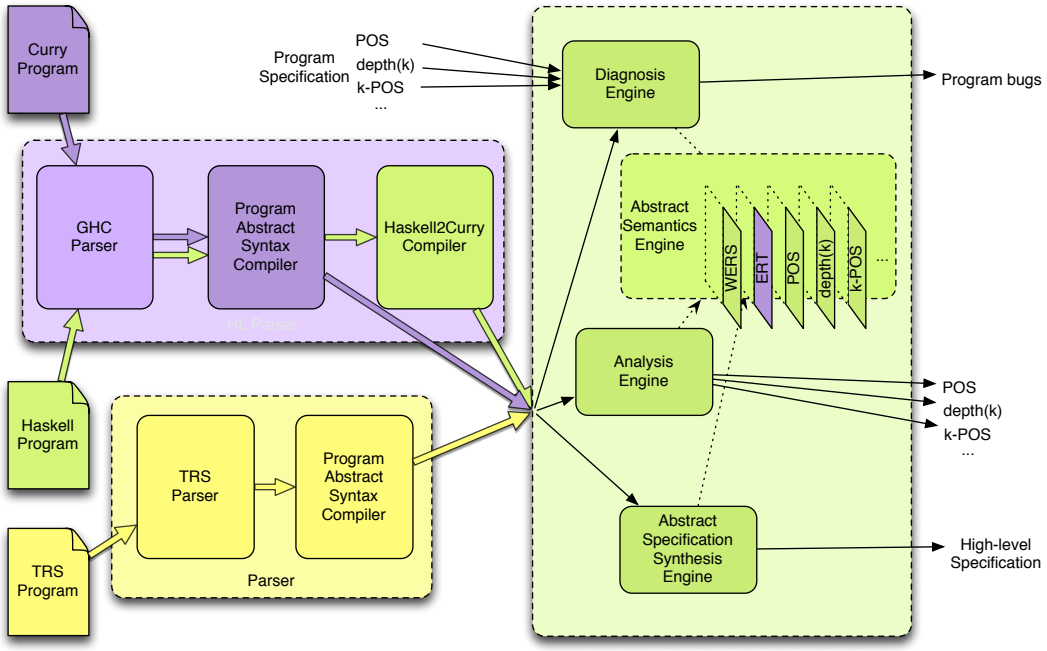


Figure 7.1: Tool Suite Layout

  - Term Rewriting Systems, both in TPDB format and OBJ format (which is also compatible with the Maude syntax);
  - Maude

- minimality of the constructs to reduce the implementation effort for the abstract semantics computations.

7.1.2 Haskell/Curry Parse Suite

Given the extreme similarity of Haskell and Curry concrete syntax, in the Haskell/Curry Parse Suite we share a (minor modification) of the GHC internal parser which allows free variable declarations for Curry and recognizes different built-ins names.

After the translation into Intermediate Common Language, in case we are handling an Haskell source, we additionally perform the orthogonalization described in Subsection 3.2.1.

This part consists of about 4000 lines of code (in addition to the patched sources of the GHC parser).

7.1.3 TPDB TRS Parser

This module (at the moment) parses a TRS written in TPDB format and then converts it into the Intermediate Common Language.

This part consists of about 2000 lines of code.

7.2 Abstract Semantics Engine

This part is the core infrastructure to compute over abstract domains which is used by all the tools of the suite. It is composed by a *domain independent* part, which is a collection of functions that implement abstract semantics evaluation functions $\mathcal{E}^\alpha[\![t]\!]$, $\mathcal{P}^\alpha[\![P]\!]$, $\mathcal{F}^\alpha[\![P]\!]$, by using abstract domain primitive operations which are realized by suitable *abstract domain modules*. In detail

domain independent engine This part is essentially a large set of Class declarations layered into different levels of abstraction (e.g. interpretation level, single interpretation binding level, domain level) with all the code that defines the default methods relying on lower level functions. For instance, the evaluation function $\mathcal{E}^\alpha[\![t]\!]$, having as input an expression of the Intermediate Common Language syntax, is defined at the domain independent interpretation level and its definition (for the base cases of the syntax) depends on the domain specific primitive for a variable evaluation and the primitive for domain embedding.

Thanks to this structure we have factorized in a unique definition the code in common with all abstract computations, but is nevertheless possible to "bypass" a default (generic) realization with a domain specific version (this in view of future extensions).

This part consists of about 6000 lines of code.

abstract domain modules These modules have to implement the structure and operation of an abstract domain (equivalence, comparison, meet, join, renaming application, variable restriction, *etc.*).

Each module provides also a parser for the abstract domain elements which is used by the abstract diagnosis engine to acquire the programs' intended abstract specification.

Note thath also concrete domains can be instancies of abstract domain modules, but clearly one cannot expect (full) semantics computations to terminate.

Up to now the only abstract domain module which is fully completed is the one for the evolving result tree $\mathbb{ERT}$ concrete domain. The other implemented modules are still incomplete, but provide nethertheless some of the required functionalities which can give outputs for some test (toy) programs that we have used to validate the complete suite.

The (partially) implemented modules are

$\mathbb{ERT}$ **module**) This module has been implemented to validate the evolving result tree semantics of Subsection 3.1.1 and to validate the complete suite. However, it can also be used to provide a quick implementation of a $depth(k)$ abstraction of evolving result tree semantics by a $depth(k)$ cut of the result of $\mathbb{ERT}$ operations.

This module consists of about 2000 lines of code.

$\mathbb{GR}$ **module**) The implementation of the abstract domain $\mathbb{GR}$ of Subsection 4.2.1 has been made using Reduced Binary Decision Diagrams (ROBDDs)[1], since in the literature this is reported as one of the most efficient ways for managing symbolic boolean formulas. The advantages in using this data structure are the compact

[1] See [15] for more details on how to manipulate boolean functions by means of ROBDDs.

representation for positive formulas and the efficient operations available for their manipulations. ROBBDs have been implemented following [20], but we also have developed and implemented some efficient variants for manipulation and definition of the specific positive formulas which are built by the $\mathcal{P}^{gr}[\![P]\!]$ operator.

This module consists of about 1000 lines of code.

$\mathbb{WERS}$ **module**) This module is at an in embryo stage. At the moment it is used just to validate some operations needed for implementing the weak evolving result set semantics of Subsection 3.1.3.

***depth*(*k*) (over $\mathbb{WERS}$) module**) Up to now, this is just a quick implementation of a $depth(k)$ abstraction of weak evolving result set semantics by a $depth(k)$ cut of the result currently implemented $\mathbb{WERS}$ operations.

I have personally developed the $\mathbb{ERT}$, $\mathbb{GR}$ and $\mathbb{WERS}$ modules and have substantially contributed to the domain independent engine.

7.3 Abstract Diagnosis Engine

This part implements the core functionality described in Chapter 5, in particular, the part for the detection of abstract incorrect rules[2]. It is essentially a Class declaration which defines the general interface of the diagnosis framework. Most of the methods are defined at the domain independent interpretation level. For instance, the detection of abstract incorrect rules, having as input a program P of the Intermediate Common Language and an abstract domain instance α (with its associated parser), depends on the domain specific primitives for parsing a user defined specification, computing the immediate consequences of the specification w.r.t. the input program, and comparing the result with the specification.

So far, we have used the abstract domain instance $\mathbb{GR}$ for testing purposes.

The instance over $depth(k)$ is lacking the parser, thus, at the moment, it has to be invoked manually.

7.4 Abstract Specification Synthesis Tool

The core of the AbsSpec 1.0 prototype consists of about 1500 lines of code implementing the tasks of generating and classifying terms. On top of the core part of the prototype, the interface module implements some functions that allow the user both to check if a specific set of equations hold, or to get the whole specification.

7.5 Analysis Tool

This tool computes iterates of $\mathcal{P}^{\alpha}[\![P]\!]$ until a fixpoint is reached, thus it is terminating for instances over Noetherian domains (e.g. $\mathbb{GR}$ and $depth(k)$). Otherwise, it should be

[2]The detection of abstract uncovered elements has been developed but have still to be tested on an actual abstract domain instance.

stopped after a finite number of iterations, but in such a case it does not necessarily provide a correct (over) approximation.

We plan to extend it with narrowing and widening operators, two typical constructions of the abstract interpretation setting, which are mainly used to achieve termination for non-Noetherian domains, but are also frequently used to speedup convergence for Noetherian "big" domains (like $depth(k)$).

Up to now, we are able to analyze the groundness behaviour of Curry programs by using the $\mathbb{GR}$ module.

Conclusions

In this thesis (Chapter 3) we have developed a semantics (the weak evolving result set semantics $\mathcal{F}^{\nu}$) which models the computed result behaviour of programs of first order fragment of Curry. Denotations of $\mathcal{F}^{\nu}$ consist in a collection of weak evolving result sets which describe how partial computed results evolve one constructor symbol at a time. $\mathcal{F}^{\nu}$ fulfills the all (our) desired requirements:

- is fully abstract w.r.t. the $\approx_{cr}^{us}$ behaviour,
- has a goal-independent definition,
- is the fixpoint of a bottom-up construction, and
- is as condensed as possible.

$\mathcal{F}^{\nu}$ has been obtained by abstraction of a concrete bottom-up fixpoint semantics which we showed to be fully abstract w.r.t. the small-step behavior of first order fragment of Curry. This methodology has served both to design the $\mathcal{F}^{\nu}$ semantics and to relate it with the small-step operational semantics in a formal way.

Furthermore, we introduced a method to automatically convert a well typed first order Haskell program into a semantic equivalent inductively sequential Curry program. We investigated on the adequateness of this transformation both w.r.t. the small-step and the big-step behaviour. As a consequence, our semantics can be used to handle well typed first order Haskell programs as well.

Among all other proposals, the weak evolving result set semantics is the first one which achieves, at the same time, goal-independency, full abstraction and condensedness. These properties are particularly helpful for the development of efficacious semantics-based program manipulation tools, e.g. automatic program analyzers, debuggers, *etc.*. To support this we have developed four applications of $\mathcal{F}^{\nu}$ semantics.

- Groundness dependencies of computed results.
- Abstract diagnosis over $\mathcal{POS}$ and $depth(k)$ domains.
- Automatic synthesis of property-oriented specifications over $depth(k)$ domain.

The applications are based on a parametric abstraction scheme for the systematic derivation, from the $\mathcal{F}^{\nu}$ semantics, of goal-independent bottom-up fixpoint (approximate) abstract semantics (Chapter 4). In particular we have shown two specific instances of this framework which model interesting properties of the computed result behavior of Curry programs, such as the groundness behavior of computed results (Subsection 4.2.1), and the $depth(k)$ abstraction of computed results (Subsection 4.2.2). To the best of our knowledge, the abstract semantics of Subsection 4.2.1 is the first attempt of groundness analysis of computed results for lazy functional logic languages.

With this abstraction framework we formulated two other applicative frameworks

Abstract diagnosis for FLP framework, an efficacious parametric scheme for the declarative debugging of functional logic programs, based on the abstract $\mathcal{P}^\alpha[\![P]\!]$ operators obtained by the abstraction framework (Chapter 5).

This approach is based on the ideas of [27, 4] which we apply to the diagnosis of functional logic programs. The framework of abstract diagnosis [27] comes from the idea of considering the abstract versions of Park's Induction Principle [80].

We showed that, given the intended abstract specification $\mathcal{S}^\alpha$ of the semantics of a program P, we can determine all the rules which are wrong w.r.t. the considered abstract property by a single application of $\mathcal{P}^\alpha[\![P]\!]$ to $\mathcal{S}^\alpha$. In the general case, diagnosing w.r.t. *abstract* program properties relieves the user from having to specify in excessive detail the program behavior (which could be more error-prone than the coding itself).

The most interesting feature of this proposal, differently from other proposals based on declarative debugging [19, 17, 18, 14], is that it can be used both with partial specifications and partial programs and works even with non-terminating programs. Moreover it does not require the user to provide error symptoms in advance and it detects multiple errors simultaneously.

Automatic synthesis of property-oriented specifications framework, a method to automatically infer high-level, property-oriented (algebraic) specifications of functional logic programs, which is parametric w.r.t. an abstract domain of computation (Chapter 6). The specifications are given by means of equations which represent behavioral relations over nested program calls.

The method computes a concise specification of program properties from the source code of the program.

Differently from other approaches based on testing, our approach relies on the computation of an approximate abstract semantics (to achieve effectiveness and good performance results). This means that, in general, we may not guarantee correctness of all the equations in the specification, but with suitable domains (like $depth(k)$) we can nevertheless infer correct equations thanks to a good compromise between correctness and efficiency.

We have proposed some instancies (case studies) of the applicative frameworks to give tangible examples. Moreover we have implemented in Haskell four proof of concept prototypes that work on the case studies of the applicative frameworks.

Groundness dependencies of computed results. A static analyzer for groundness dependencies. Due to both the high level of abstraction of the modeled property and the use of efficient data structures for the representation of the semantic denotations (i.e., positive formulas) we expect that the analysis will scale well with the size of the programs.

Abstract diagnosis over $\mathcal{POS}$ and $depth(k)$ domains. We have considered the $depth(k)$ and the $\mathcal{POS}$ abstractions as case studies, showing, my means of several examples, that the resulting abstract diagnosis encompasses some limitations of previous works.

Automatic synthesis of property-oriented specifications over $depth(k)$ domain. The use of the $depth(k)$ domain permits to have a terminating synthesis where some of the equations are provably correct (those that do not involve cut variables in the abstract semantics).

In the future, on the semantics side, we are interested in extending our results to all the features of Curry: equational constraints (i.e., strict equality), residuation (in order to tackle strict primitive arithmetic operations) and higher-order. To do so, we think it would be profitable to formalize the partial computed results as elements of a cylindric constraint system ([86]) and, once the semantics has been reformulated in such terms, then extend the constraint system with strict equality and arithmetic primitives. Furthermore, cylindric constraint systems go particularly well with abstract interpretation techniques, as good abstractions can be obtained by abstracting only the constraint system while keeping the overlying structure.

On the abstraction side we intend to derive the instances of the framework for the reduced products between $depth(k)$ and several domain typical of program analysis, like $\mathcal{POS}$, sharing, types, *etc.*. This could provide better tradeoffs between precision and size of (extensional) specifications. In particular, for the applications in abstract diagnosis, we hope that these domains will detect the same errors of $depth(k)$ but at a smaller k, which could make feasible the use of an extensional intended specification in place of other intensional versions.

Clearly in this framework we cannot handle properties like termination or observations over call patterns, namely properties which are not an abstraction of weak evolving result sets. To this end we need more concrete semantics, which could be developed by (more concrete than ν) abstractions of the small-step semantics introduced in Section 2.2. Note that, if a direct construction of $\mathcal{F}^{\nu}$ were adopted, then a different base semantics should be designed from scratch.

In the future, we want to develop (by abstraction of small-step trees, along the lines of the construction for the evolving result tree semantics) a semantics which can model "functional dependencies". Such a semantics is needed to tackle pre-post conditions and will be employed to define the homologous of abstract verification for logic programs [24] in the functional logic paradigm.

Bibliography

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005. (document), 2.1.1, 3.3

[2] M. Alpuente, D. Ballis, F. J. Correa, and M. Falaschi. An Integrated Framework for the Diagnosis and Correction of Rule-Based Programs. *Theoretical Computer Science*, 411(47):4055–4101, 2010. 5

[3] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and J. Iborra. A Compact Fixpoint Semantics for Term Rewriting Systems. *Theoretical Computer Science*, 411(37):3348–3371, 2010. 3.3

[4] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract Diagnosis of Functional Programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation – 12th International Workshop, LOPSTR 2002, Revised Selected Papers*, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16, Berlin, 2003. Springer-Verlag. 3.3, 4.2.2, 4.2.2, 4.3, 5.4.6, 5.6, 5.7, 7.5

[5] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'02)*, pages 4–16, New York, NY, USA, 2002. Acm. 6.1, Why useful?

[6] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000. (document), 1.4.3, 1.4.3, 2.1.1

[7] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In B. Le Charlier, editor, *Proceedings of Static Analysis Symposium (SAS'94)*, volume 864 of *Lecture Notes in Computer Science*, pages 266–280, Berlin, 1994. Springer-Verlag. 4.2.1

[8] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK, 1998. 1.4.1

[9] G. Bacci and M. Comini. Abstract Diagnosis of First Order Functional Logic Programs. In M. Alpuente, editor, *Logic-based Program Synthesis and Transformation, 20th International Symposium*, volume 6564 of *Lecture Notes in Computer Science*, pages 215–233, Berlin, 2011. Springer-Verlag. 3.3

[10] G. Bacci, M. Comini, M. A. Feliú, and A. Villanueva. Automatic Synthesis of Specifications for Curry Programs. In *Logic-based Program Synthesis and Transformation, 21th International Symposium*, Lecture Notes in Computer Science, Berlin, 2011. Springer-Verlag. To appear.

[11] D. Bert and R. Echahed. Abstraction of Conditional Term Rewriting Systems. In J. W. Lloyd, editor, *Proceedings of the 1995 Int'l Symposium on Logic Programming*

(ILPS'95), pages 162–176, Cambridge, Mass., 1995. The MIT Press. 3.3, 4.2.2, 4.2.6, 4.2.2, 4.3, 5.6

[12] G. Birkhoff. *Lattice Theory*, volume 25 of *AMS Colloquium Publication*. American Mathematical Society, 1967. 1, 1.5.1

[13] G. Birkhoff and S. MacLane. *A Survey of Modern Algebra*, volume 25. MacMillan, 1965. Third edition. 1

[14] B. Braßel. A Technique to build Debugging Tools for Lazy Functional Logic Languages. In M. Falaschi, editor, *Proceedings of the 17th Workshop on Functional and (Constraint) Logic Programming (WFLP 2008)*, pages 63–76, 2008. I.4, 5, 5.1, 7.5

[15] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24:293–318, September 1992. 1

[16] R. Caballero and J. Sanchez. TOY: A Multiparadigm Declarative Language, Version 2.2.3. Technical report, UCM, Madrid, 2006. (document)

[17] R. Caballero-Roldán. A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming*, pages 8–13, New York, NY, USA, 2005. ACM Press. I.4, 5, 5.1, 5.3, 5.4.4, 5.4.5, 5.4.2, 7.5

[18] R. Caballero-Roldán, F. J. López-Fraguas, and M. Rodríquez-Artalejo. Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. In *Proceedings of Fifth International Symposium on Functional and Logic Programming*, volume 2024 of *Lecture Notes in Computer Science*, pages 170–184, Berlin, 2001. Springer-Verlag. I.4, 5, 5.1, 5.3, 7.5

[19] R. Caballero-Roldán, M. Rodríquez-Artalejo, and R. del Vado-Vírseda. Declarative Diagnosis of Missing Answers in Constraint Functional-Logic Programming. In *Proceedings of 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 305–321, Berlin, 2008. Springer-Verlag. I.4, 5, 5.1, 5.3, 5.4.5, 5.4.2, 7.5

[20] J. Christiansen and F. Huch. A purely functional implementation of ROBDDs in Haskell. In Henrik Nilsson, editor, *Trends in Functional Programming*, volume 7, Bristol, UK, 2006. Intellect. 7.2

[21] Jan Christiansen and Sebastian Fischer. EasyCheck – Test Data for Free. In *Proceedings of the 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2008. 6.5

[22] Koen Claessen, Nicholas Smallbone, and John Hughes. Quickspec: Guessing formal specifications using testing. In *4th International Conference on Tests and Proofs (TAP 2010)*, volume 6143, pages 6–21, Málaga, Spain, July 1-2, 2010. 6.1, Why useful?, 6.2, 6.5

[23] M. Comini. *An Abstract Interpretation Framework for Semantics and Diagnosis of Logic Programs*. PhD thesis, Dipartimento di Informatica, Universitá di Pisa, Pisa, Italy, 1998. I.3, 4.2.1

[24] M. Comini, R. Gori, G. Levi, and P. Volpe. Abstract Interpretation based Verification of Logic Programs. *Science of Computer Programming*, 49(1–3):89–123, 2003. 2.2, 3.4, 5.7, 7.5

[25] M. Comini, G. Levi, and M. C. Meo. A Theory of Observables for Logic Programs. *Information and Computation*, 169:23–80, 2001. 4.2.1

[26] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of Logic Programs by Abstract Diagnosis. In M. Dams, editor, *Proceedings of Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop (LOMAPS'96)*, volume 1192 of *Lecture Notes in Computer Science*, pages 22–50, Berlin, 1996. Springer-Verlag. (document), 5

[27] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract Diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999. (document), 4.2.2, 5, 5.3, 5.6, 5.7, 7.5

[28] M. Comini, G. Levi, and G. Vitiello. Declarative Diagnosis Revisited. In J. W. Lloyd, editor, *Proceedings of the 1995 Int'l Symposium on Logic Programming (ILPS'95)*, pages 275–287, Cambridge, Mass., 1995. The MIT Press. 5

[29] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Los Angeles, California, January 17–19*, pages 238–252, New York, NY, USA, 1977. ACM Press. I.3, 1.5, 6.3

[30] P. Cousot and R. Cousot. A constructive characterization of the lattices of all retracts, pre-closure, quasi-closure and closure operators on a complete lattice. *Portugaliæ Mathematica*, 38(2):185–198, 1979. 1.5.1

[31] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, San Antonio, Texas, January 29–31*, pages 269–282, New York, NY, USA, 1979. ACM Press. I.3, 1.5, 1.5.1, 1.5.2, 1.5.2

[32] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992. 1.5.2

[33] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming, Proceedings PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295, Berlin, 1992. Springer-Verlag. 1.5.3

[34] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *Proceedings of the IEEE International Conference on*

Computer Languages (ICCL'94), pages 95–112, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. 1.5.4

[35] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997. 1.5.4

[36] R. Echahed and J.-C. Janodet. On Constructor-based Graph Rewriting Systems. In *Research Report 985-I.* IMAG, 1997. 1.4.1

[37] R. Echahed and J.-C. Janodet. Admissible Graph Rewriting and Narrowing. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 325–340. MIT Press, 1998. 1.4.1, 1.4.1, 1.4.1, 2.1.1, 2.1.1, 1, 8

[38] H. Ehrig and G. Taentzer. Computing by graph transformation: A survey and annotated bibliography. *Bulletin of the European Association for Theoretical Computer Science*, 59:182–226, 1996. 1.4.1

[39] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[40] G. Ferrand. Error Diagnosis in Logic Programming, an Adaptation of E. Y. Shapiro's Method. *Journal of Logic Programming*, 4(3):177–198, 1987. I.4

[41] G. Filè, R. Giacobazzi, and F. Ranzato. A Unifying View on Abstract Domain Design. *ACM Computing Surveys*, 28(2):333–336, 1996. 1.5.3

[42] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *31st International Conference on Software Engineering (ICSE'09)*, pages 430–440, 2009. 6.1, Why useful?

[43] Carlo Ghezzi and Andrea Mocci. Behavior model based component search: an initial assessment. In *Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation (SUITE'10)*, pages 9–12, New York, NY, USA, 2010. ACM. 6.1

[44] R. Giacobazzi. *Semantic Aspects of Logic Program Analysis.* PhD thesis, Dipartimento di Informatica, Universitá di Pisa, Pisa, Italy, 1992. I.3

[45] R. Giacobazzi and F. Ranzato. Functional dependencies and Moore-set completions of abstract interpretations and semantics. In J. W. Lloyd, editor, *Proceedings of the 1995 Int'l Symposium on Logic Programming (ILPS'95)*, pages 321–335, Cambridge, Mass., 1995. The MIT Press. 1.5.3

[46] R. Giacobazzi and F. Ranzato. Completeness in abstract interpretation: A domain perspective. In M. Johnson, editor, *Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology (AMAST'97)*, Lecture Notes in Computer Science, Berlin, 1997. Springer-Verlag. 1.5.4

[47] R. Giacobazzi and F. Scozzari. Intuitionistic Implication in Abstract Interpretation. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proceedings of Ninth International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, volume 1292 of *Lecture Notes in Computer Science*, pages 175–189, Berlin, 1997. Springer-Verlag. 1.5.3

[48] J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages. In Frank Pfenning, editor, *Term Rewriting and Applications, 17th International Conference, RTA 2006, Proceedings*, volume 4098 of *Lecture Notes in Computer Science*, pages 297–312. Springer-Verlag, 2006. 2.1.2

[49] J. C. González-Moreno, M. T. Hortalá-González, F. J. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In H. Nielson, editor, *Programming Languages and Systems - ESOP '96*, volume 1058 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 1996. (document), 3.1.1, 3.1.3, 3.1.37, 3.3

[50] J. C. González-Moreno, M. T. Hortalá-González, F. J. López-Fraguas, and M. Rodríquez-Artalejo. An approach to declarative programming based on a rewriting logic. *The Journal of Logic Programming*, 40(1):47–87, 1999. (document), 2.1.1, 3.1.1, 3.1.3, 3.1.37, 3.3

[51] M. Hanus. Curry: An integrated functional logic language (vers. 0.8.2), 2006. Available at URL: `http://www.informatik.uni-kiel.de/~curry`. (document), 1.3, 1.4.1, 5, 2.1.1, 2.2, 9, 3.3, Why useful?

[52] M. Hanus. Call pattern analysis for functional logic programs. In *Proc. of the 10th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'08)*, pages 67–78. ACM Press, 2008. 3.3, 4.2.2, 4.2.7, 4.2.2, 4.3, 5.5, 5.6

[53] Michael Hanus. A Unified Computation Model for Functional and Logic Programming. In *24th ACM Symposium on Principles of Programming Languages (POPL 97)*, pages 80–93, 1997. Why useful?

[54] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. Discovering Documentation for Java Container Classes. *IEEE Transactions on Software Engineering*, 33(8):526–542, 2007. 6.1, Why useful?

[55] P. Hudak, J. Peterson, and J. Fasel. *A Gentle Introduction to Haskell - Version 98*. `http://www.haskell.org/tutorial/`, 1999. 1.3

[56] H. Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *J. Log. Program.*, 12:237–255, 1992. 1.3.2

[57] N. D. Jones and H. Søndergaard. A Semantics-based Framework for the Abstract Interpretation of PROLOG. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis-Horwood, 1987. I.3

[58] Amir A. Khwaja and Joseph E. Urban. A property based specification formalism classification. *The Journal of Systems and Software*, 83:2344–2362, 2010. 6.1

[59] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992. 1.4.1

[60] H. Kuchen, R. Loogen, J. J. Moreno-Navarro, and M. Rodríguez-Artalejo. The Functional Logic Language BABEL and Its Implementation on a Graph Machine. *New Generation Computing*, 14(4):391–427, 1996. 6

[61] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proceedings of the ACM Symp. on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993. 2.1.1

[62] J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987. I.4

[63] F. J. López-Fraguas and J. Rodríguez-Hortalá. The Full Abstraction Problem for Higher Order Functional-Logic Programs. *CoRR*, abs/1002.1833, 2010. Available at URL: `http://arxiv.org/abs/1002.1833`. 2.2, 3.1.2, 3.1.2, 3.1.3, 3.3, 3.4

[64] F. J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In M. Leuschel and A. Podelski, editors, *Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 197–208. ACM, 2007. (document), 3.1.1, 3.1.3, 3.1.37, 3.3

[65] F. J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A Multiparadigm Declarative System. In P. Narendran and M. Rusinowitch, editors, *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 244–247, Berlin, 1999. Springer-Verlag. 2.2

[66] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, NewYork, 1974. 1

[67] K. Marriott and H. Søndergaard. Abstract Interpretation of Logic Programs: the Denotational Approach. In A. Bossi, editor, *Proceedings of Fifth Italian Conference on Logic Programming*, pages 399–425, 1990. I.3

[68] J. M. Molina-Bravo and E. Pimentel. Composing programs in a rewriting logic for declarative programming. *Theory and Practice of Logic Programming*, 3:189–221, 2003. 3.3, 3.4

[69] A. Mycroft. Completeness and predicate-based abstract interpretation. In *Proceedings of the ACM Symp. on Partial Evaluation and Program Manipulation (PEPM'93)*, pages 179–185, New York, NY, USA, 1993. ACM Press. 1.5.4

[70] L. Naish. Declarative Debugging of Lazy Functional Programs. *Australian Computer Science Communications*, 15(1):287–294, 1993. 5.1

[71] L. Naish. A Declarative Debugging Scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997. 5, 5.1

[72] L. Naish and Timothy Barbour. A Declarative Debugger for a Logical-Functional Language. *Eighth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, 2:91–99, 1995. I.4, 5, 5.1, 5.3

[73] H. R. Nielson and F. Nielson. Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis. In *Symposium on Principles of Programming Languages*, pages 332–345, 1997.

[74] H. Nilsson and P. Fritzson. Algorithmic debugging of lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, 1994. 5.1

[75] H. Nilsson and P. Fritzson. Algoritmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(1):337–370, 1994. I.4, 5

[76] Isabel Nunes, Antónia Lopes, and Vasco Vasconcelos. Bridging the Gap between Algebraic Specification and Object-Oriented Generic Programming. In Saddek Bensalem and Doron Peled, editors, *9th International Workshop on Runtime Verification (RV 2009)*, volume 5779 of *Lecture Notes in Computer Science*, pages 115–131. Springer Berlin / Heidelberg, 2009. 6.1

[77] O. Ore. Galois Connections. *Transactions of the American Mathematical Society*, 55:493–513, 1944. 1.5.2, 1.5.2

[78] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988. 1.4.1

[79] C. Palamidessi. Algebraic Properties of Idempotent Substitutions. In M. S. Paterson, editor, *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *Lecture Notes in Computer Science*, pages 386–399, Berlin, 1990. Springer-Verlag. 1.4.1

[80] D. Park. Fixpoint Induction and Proofs of Program Properties. *Machine Intelligence*, 5:59–78, 1969. 5.3, 5.7, 7.5

[81] L. M. Pereira. Rational debugging in logic programming. In E. Y. Shapiro, editor, *Proceedings of Third Int'l Conf. on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 203–210, Berlin, 1986. Springer-Verlag. I.4

[82] S. Peyton Jones. *Haskell 98 Language and Libraries - The Revised Report*. Cambridge University Press, Cambridge, UK, 2003. Available at `http://www.haskell.org/definition/`. 1.3, Why useful?

[83] Derek Rayside, Aleksandar Milicevic, Kuat Yessenov, Greg Dennis, and Daniel Jackson. Agile specifications. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA 2009)*, pages 999–1006. ACM, 2009. 6.1

[84] U. S. Reddy and S. N. Kamin. On the power of abstract interpretation. In *Proceedings of the 1992 IEEE Internat. Conf. on Computer Languages* (*ICCL'92*), Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. 1.5.4

[85] Russell, B. *The Principles of Mathematics.* Number vol. 1 in The Principles of Mathematics. University Press, 1903. 1.1.1

[86] V. A. Saraswat and M. C. Rinard. Concurrent constraint programming. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, NY, USA, 1990. Acm. 3.4, 7.5

[87] R. C. Sekar, P. Mishra, and I. V. Ramakrishnan. On the power and limitation of strictness analysis. *Journal of the ACM*, 44(3):505–525, 1997. 1.5.4

[88] E. Y. Shapiro. Algorithmic Program Debugging. In *Proceedings of Ninth Annual ACM Symp. on Principles of Programming Languages*, pages 412–531, New York, NY, USA, 1982. ACM Press. I.4

[89] E. Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, Cambridge, Mass., 1983. ACM Distinguished Dissertation. 5.1

[90] M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen. *Term Graph Rewriting. Theory and Practice.* J. Wiley & Sons, Chichester, UK, 1993. 1.4.1

[91] H. Søndergaard and P. Sestoft. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, 27(6):505–517, 1990. bla bla

[92] A. Tarski. A Lattice-theoretical Fixpoint Theorem and its Applications. *Pacific J. Math.*, 5:285–309, 1955. 1.2.1

[93] TeReSe, editor. *Term Rewriting Systems.* Cambridge University Press, Cambridge, UK, 2003. 1.4.1

[94] H. van Vliet. *Software Engineering–Principles and Practice.* John Wiley, 1993. 6.1

[95] J.M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):10–24, 1990. 6.1

[96] Bo Yu, Liang Kong, Yufeng Zhang, and Hong Zhu. Testing Java Components based on Algebraic Specifications. In *First International Conference on Software Testing, Verification, and Validation (ICST 2008)*, pages 190–199. IEEE Computer Society, 2008. 6.1