

A Compact Fixpoint Semantics for Term Rewriting Systems*

M. Alpuente[†] M. Comini[‡] S. Escobar[†] M. Falaschi[§]
J. Iborra[†]

June 2, 2010

Abstract

This work is motivated by the fact that a “compact” semantics for term rewriting systems, which is essential for the development of effective semantics-based program manipulation tools (e.g. automatic program analyzers and debuggers), does not exist. The big-step rewriting semantics that is most commonly considered in functional programming is the set of values/normal forms that the program is able to compute for any input expression. Such a big-step semantics is unnecessarily oversized, as it contains many “semantically useless” elements that can be retrieved from a smaller set of terms. Therefore, in this article, we present a compressed, goal-independent collecting fixpoint semantics that contains the smallest set of terms that are sufficient to describe, by semantic closure, all possible rewritings. We prove soundness and completeness under ascertained conditions. The compactness of the semantics makes it suitable for applications. Actually, our semantics can be finite whereas the big-step semantics is generally not, and even when both semantics are infinite, the fixpoint computation of our semantics produces fewer elements at each step. To support this claim we report several experiments performed with a prototypical implementation.

*This work has been partially supported by the EU (FEDER), by the Spanish MEC/MICINN, under grant TIN 2007-68093-C02, and by the Italian MUR under grant RBIN04M8S8, FIRB project, Internationalization 2004.

[†]Departamento de Sistemas Informáticos y Computación. Technical University of Valencia, Camino de Vera s/n, 46022 Valencia, Spain. {alpuente,escobar,jiborra}@dsic.upv.es

[‡]Dipartimento di Matematica e Informatica. University of Udine, Via delle Scienze 206, 33100 Udine, Italy. comini@dimi.uniud.it

[§]Dipartimento di Scienze Matematiche e Informatiche. Pian dei Mantellini 44, 53100 Siena, Italy. moreno.falaschi@unisi.it

1 Introduction

1.1 Why a new semantics

Finding program bugs is a long-standing problem in software construction. Unfortunately, the debugging support is rather poor for functional languages (see [40, 25, 31] and references therein), and there are no good general-purpose semantics-based debuggers available. One of the basic reasons for the lack of simple, handy development tools for functional programs (like program analyzers, debuggers, and correctors) is the lack of a suitable semantics that is compact and goal-independent, i.e., a program semantics defined only by determining the operational behavior of a small set of terms that are sufficient to describe, by semantic closure, the meaning of all input expressions. The idea of such a semantics seems clear-cut, as already demonstrated by several works in other programming paradigms like logic programming [8, 18]; however, to the best of our knowledge, it has never been investigated in the context of functional programming and term rewriting systems. Furthermore, the definition of such a semantics is not trivial, since we want the semantics to ensure interesting computational properties while still dealing with a broad class of TRSs.

We came up with the idea of a compressed semantics when investigating on how to convey to functional programming the semantics-based, debugging approach of [11, 12], for logic programs, which essentially consists of the application of Abstract Interpretation techniques [15] to an “appropriate” goal-independent fixpoint semantics. Among other valuable facilities, this debugging approach supports the development of cogent diagnostic tools that find program errors without having to determine symptoms in advance. 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 starting with “most general” calls (goals¹), while still providing a complete characterization of the program behavior.

Defining a *collecting semantics* is usually the first crucial step in adapting the general methodology of Abstract Interpretation to the semantic framework of the programming language at hand [36]. In [1], we developed a (preliminary) Abstract Diagnosis framework for functional languages by using the formalism of Term Rewriting Systems (TRSs), which is widely recognized [5, 10, 28, 38] as a suitable computational model for functional programming languages (e.g. Haskell, Hope, Miranda, or Maude). We used a collecting semantics that is very similar to the big-step rewriting semantics as a basis for the abstract debugger. However, the resulting tool was inefficient. Not surprisingly, the main reason for this inefficiency was the accidental high redundancy of the semantics, which caused the algorithms to

¹Following the terminology of logic programming, we often refer to an input expression (i.e., a program call) as “the goal”.

use and produce redundant information at each stage. In the best case, this redundant information reduces the performance and, in the worst case, ends in ineffective methods. In contrast, the same methodology gave good results in [12] because it was applied to a compact semantics (i.e., the s -semantics).

Thus, our specific objective in this paper is to avoid all such redundant elements in the semantics while still characterizing the meaning of any expression. We improve previous attempts by some of the authors [4, 1] to produce a compact, goal-independent semantics for functional programs.

1.2 Denotations as syntactic objects

Term rewriting systems [16, 28] provide an adequate computational model for functional languages. A *functional program* is a set of functions that is defined by (oriented) equations or *rules*. A *functional computation* consists of replacing subexpressions by equal subexpressions (w.r.t. the function definitions) until no more replacements (or *reductions*) are possible, and a result is obtained.

In the literature of TRSs, equations $t = s$ over terms t, s (with variables) are sometimes used to represent program semantics, i.e., the computed input/output relation of all functions. In this paper, we prefer to use (special cases of) oriented equations $t \mapsto s$ (modulo renaming) to make explicit the (semantic) fact that a term t reduces to term s (in symbols, $t \rightarrow^* s$) and not vice versa. Moreover, the equation notation is more suited in the case of confluent TRSs. However, several of our results in this paper do not require confluence or other properties that are often assumed in traditional rewriting-based languages. Modern functional languages such as Maude [10], and functional-logic languages such as Curry [23], or TOY [30] deal with non-determinism, and thus the confluence requirement is done away with. We also prefer $t \mapsto s$ instead of $t \rightarrow s$ to avoid confusion with program rules, as $t \mapsto s$ can indeed represent several reduction steps with program rules of the form $t \rightarrow s$.

The idea of using syntactic domains for describing program semantics, and in particular the use of rules in the denotation, is inspired from the literature of logic programming (see [8]) where it is commonly used to capture various computational aspects (like *computed answers*, *call patterns*, *resultants*) in a goal-independent way. For instance, the Ω -semantics of Bossi et al. [8] collects sets of Horn clauses, which model the program *resultants*, so that goals can be solved by simply “executing them in the semantics”. It is important to note that this is generally far cheaper than executing goals in the original program, since the “real computation” is pretty much embedded in the representation.

Following the Ω -semantics approach, sometimes we will use the elements $t \mapsto s$ in the denotation as standard rewrite rules $t \rightarrow s$ (obviously this does not imply that these rules belong to the original program), and vice versa.

1.3 The standard big-step rewriting semantics

Given the TRS \mathcal{R} , the coarse (goal-dependent) big-step semantics of term rewriting systems collects, for every possible ground input term t , a representation $t \mapsto s$ of all rewritings $t \rightarrow^* s$ in \mathcal{R} , where s is a normal form² of t . As an illustrative example, consider the following term rewriting system (variable names start with an uppercase letter).

Example 1.1

Consider the TRS $\mathcal{R}_{INC} := \{\text{inc}(X) \rightarrow \mathbf{s}(X), \text{plus2}(X) \rightarrow \text{inc}(\text{inc}(X))\}$. The standard big-step rewriting semantics for TRS \mathcal{R}_{INC} contains the following rewritings:

$$\begin{aligned} &\{0 \mapsto 0, \mathbf{s}(0) \mapsto \mathbf{s}(0), \mathbf{s}(\mathbf{s}(0)) \mapsto \mathbf{s}(\mathbf{s}(0)), \dots, \\ &\text{inc}(0) \mapsto \mathbf{s}(0), \text{inc}(\mathbf{s}(0)) \mapsto \mathbf{s}(\mathbf{s}(0)), \text{inc}(\mathbf{s}(\mathbf{s}(0))) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(0))), \dots, \\ &\mathbf{s}(\text{inc}(0)) \mapsto \mathbf{s}(\mathbf{s}(0)), \mathbf{s}(\text{inc}(\mathbf{s}(0))) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(0))), \dots, \\ &\mathbf{s}(\mathbf{s}(\text{inc}(0))) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(0))), \mathbf{s}(\mathbf{s}(\text{inc}(\mathbf{s}(0)))) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{s}(0)))) \dots, \\ &\text{inc}(\text{inc}(0)) \mapsto \mathbf{s}(\mathbf{s}(0)), \text{inc}(\text{inc}(\mathbf{s}(0))) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(0))), \dots, \\ &\text{inc}(\text{inc}(\text{inc}(0))) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(0))), \dots, \\ &\text{plus2}(0) \mapsto \mathbf{s}(\mathbf{s}(0)), \text{plus2}(\mathbf{s}(0)) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(0))), \dots, \\ &\mathbf{s}(\text{plus2}(0)) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(0))), \mathbf{s}(\text{plus2}(\mathbf{s}(0))) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{s}(0)))) \dots, \\ &\text{plus2}(\text{inc}(0)) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(0))), \text{plus2}(\text{inc}(\mathbf{s}(0))) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{s}(0)))) \dots, \\ &\text{inc}(\text{plus2}(0)) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(0))), \text{inc}(\text{plus2}(\mathbf{s}(0))) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{s}(0)))) \dots\} \end{aligned}$$

The meaning of the expression $\text{inc}(\text{plus2}(\mathbf{s}(0)))$ is directly accessible in the denotation element $\text{inc}(\text{plus2}(\mathbf{s}(0))) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{s}(0))))$ of the semantics. However, this meaning would also be retrievable by “composing” the meanings of the terms $\text{plus2}(\mathbf{s}(0))$ and $\text{inc}(\mathbf{s}(\mathbf{s}(\mathbf{s}(0))))$, which are given by the semantic rules $\text{plus2}(\mathbf{s}(0)) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(0)))$ and $\text{inc}(\mathbf{s}(\mathbf{s}(\mathbf{s}(0)))) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{s}(0))))$, respectively.

Actually, the value of any expression t could be easily derived from a (more compact) goal-independent representation by “executing” t in such a compact semantics (using the elements in the denotation as program rules), similarly to semantics defined for logic programming in [8]. In our case, the input term can be “reduced in the semantics” (by using standard term rewriting), which needs much fewer computation steps than running the original program, since many of the necessary partial computations have been performed in advance and recorded in the semantics once and for all. In fact, we need to apply (at most) one semantic rule for each nested expression (redex), since no nested calls appear at the right-hand sides of the rules in the denotation. Thus, we achieve a good balance between compactness and agile executability of the model.

²Functional languages often consider *values* (i.e., ground constructor terms) as the interesting results of computations, whereas *normal forms* (i.e., irreducible terms) are more commonly considered in term rewriting systems.

Example 1.2

By using the semantic rules $\text{inc}(\mathbf{s}(\mathbf{s}(\mathbf{s}(0)))) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{s}(0))))$ and $\text{plus2}(\mathbf{s}(0)) \mapsto \mathbf{s}(\mathbf{s}(\mathbf{s}(0)))$ of Example 1.1, the meaning of the goal $\text{inc}(\text{plus2}(\mathbf{s}(0)))$ is achievable in two reduction steps:

$$\text{inc}(\text{plus2}(\mathbf{s}(0))) \rightarrow \text{inc}(\mathbf{s}(\mathbf{s}(\mathbf{s}(0)))) \rightarrow \mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{s}(0)))),$$

whereas reducing $\text{inc}(\text{plus2}(\mathbf{s}(0)))$ in \mathcal{R}_{INC} requires twice the number of steps.

It is important to emphasize that both the goal-independency and compactness of the semantics are necessary for the development of *efficient* semantics-based program manipulation tools, including abstract analyzers and debuggers that are based on abstract interpretation [2, 11, 12].

1.4 Relations with other semantics

In [4], a naïve, goal-independent, collecting semantics that models computed answers for canonical TRSs was developed. This semantics relies on narrowing [20, 26], which is a generalization of term rewriting where pattern matching is replaced by syntactic unification. Following the approach of [8, 18, 19], the semantics of [4] was built by narrowing in \mathcal{R} all “flat calls” $f(x_1, \dots, x_n)$ and then collecting all expressions $f(x_1, \dots, x_n)\theta \mapsto t$ whenever $f(x_1, \dots, x_n)$ narrows in \mathcal{R} to t with computed answer substitution θ . This semantics allows the reconstruction of the computed answers (solutions) of any reachability goal $t \rightarrow^* y$ (i.e., the substitutions σ such that $\mathcal{R} \vdash t\sigma \rightarrow^* y\sigma$) by recursively unifying $t \mapsto y$ with the elements in the denotation. Roughly speaking, nested expressions in t have to be flattened first so that the term structure is directly accessible to unification. Flattening works as follows. Let $t[s]$ denote that s is a subterm of t . Then, the non-variable subterm s of t is replaced with a fresh variable z in order to transform the goal $t \mapsto y$ into the goals $s \mapsto z, t[z] \mapsto y$, which are interpreted (“run”) in the denotation by standard unification. A more refined version of this semantics modeling the computed answers under different narrowing strategies, including lazy narrowing, was proposed in [2].

Unfortunately, the narrowing-based semantics of [2, 4] essentially have the same redundancy drawback of the big-step rewriting semantics, since they contain many “semantically useless” elements that can be reconstructed from a smaller set of terms, as shown in the following example.

Example 1.3

Consider the TRS $\mathcal{R}_{ID} := \{\text{id}(\mathbf{X}) \rightarrow \mathbf{X}, \text{id}(0) \rightarrow 0, \text{id}(\mathbf{s}(\mathbf{X})) \rightarrow \mathbf{s}(\text{id}(\mathbf{X}))\}$. The narrowing-based semantics of [4] yields

$$\{\text{id}(\mathbf{s}^n(\mathbf{t})) \mapsto \mathbf{s}^n(\mathbf{t}) \mid n \geq 0, \mathbf{t} \in \{0, \mathbf{X}\}\},$$

instead of the much more compact, accomplishable representation $\{\text{id}(\mathbf{X}) \mapsto \mathbf{X}\}$, which still allows the (value or normal form) meaning of any ground input expression to be retrieved by standard rewriting.

Nevertheless, it is important to note that the (apparent) redundancy in the narrowing-based semantics of [2, 4] is the key point in modeling the observable of computed answers. For instance, in the original program, the expression $\text{id}(\mathbf{Z})$ narrows to 0 with computed answer $\{\mathbf{Z}/0\}$. This is perfectly captured by the semantics of [4] illustrated above, whereas it could not be inferred from the more “compact semantics” just containing $\text{id}(\mathbf{X}) \mapsto \mathbf{X}$. However, the exuberance in the semantics of [4] is not interesting or admissible for the purposes of this paper, which is modeling the (value or normal form) meaning of every expression in a pure functional language that can be given a rewriting semantics such as Maude [10] or Haskell [38]. In this paper, we are not interested in modeling computed answers.

To manifest tangibly the degree of compactness of our semantics, we present the experimental results obtained by a proof-of-concept implementation. The benchmarks show that our compact semantics is a suitable basis for the development of efficient automatic program analyzers and debuggers.

Plan of the paper The paper is organized as follows. We present some preliminary notions in Section 2. In Section 3, we introduce a compression mechanism that is based on the removal of every element that is a “rewriting consequence” of other elements in the semantics. In other words, the compact semantics is obtained by collecting only a representative set of the “most general” rewriting sequences. In order to support fixpoint computations, an effective “decompression” mechanism is also formalized, which is able to retrieve the original semantics from the compact one when it is needed (e.g. for membership test). In Section 4, we associate a (continuous) immediate consequence operator $T_{BT, \mathcal{R}}$ to a program \mathcal{R} that allows us to derive the compact semantics. Similarly to [1, 4], the immediate consequence operator $T_{BT, \mathcal{R}}$ is computed by narrowing, which provides the most general rewriting sequences, and we ascertain the conditions that ensure that the operator is effectively computable. Then, we prove the soundness and completeness of our semantics w.r.t. the natural big-step rewriting semantics for some particular classes of TRSs. In Section 5 we present a prototypical implementation of a tool that computes (finite approximations of) the compressed semantics, together with some experimental evaluations. Finally, Section 6 concludes.

2 Preliminaries

Let us briefly recall some known results about rewrite systems [5, 39]. Throughout this paper, \mathcal{V} denotes a countably infinite set of variables and Σ denotes a finite set of function symbols, or signature, each of which has a fixed associated arity. $\mathcal{T}(\Sigma, \mathcal{V})$ and $\mathcal{T}(\Sigma)$ denote the non-ground and ground term algebra built on $\Sigma \cup \mathcal{V}$ and Σ , respectively. Terms $\mathcal{T}(\Sigma, \mathcal{V})$ are viewed as labelled trees in the usual way. Positions are represented by sequences of natural numbers denoting an access path in a term. The top or root position is denoted by ϵ . Given $S \subseteq \Sigma \cup \mathcal{V}$, $O_S(t)$ denotes the set of positions of a term t that are rooted by symbols or variables in S . $O_{\{f\}}(t)$ with $f \in \Sigma \cup \mathcal{V}$ is simply denoted by $O_f(t)$. $t|_p$ is the subterm at the position p of t . $t[s]_p$ is the term t with the subterm at the position p replaced with term s . Syntactic equality of two terms t and s is represented by $t = s$. By $Var(s)$, we denote the set of variables occurring in the syntactic object s . A *fresh* variable is a variable that appears nowhere else. A *linear term* is a term where every variable occurs only once.

A *substitution* is a mapping from the set of variables \mathcal{V} into the set of terms $\mathcal{T}(\Sigma, \mathcal{V})$, which differs from the identity only for a finite set of variables. A substitution is represented as $\{x_1/t_1, \dots, x_n/t_n\}$ for variables x_1, \dots, x_n and terms t_1, \dots, t_n . The *empty substitution* is denoted by *id*. The application of substitution θ to term t is denoted by $t\theta$. The composition of substitutions θ and σ , denoted by $\theta\sigma$, satisfies $t(\theta\sigma) = (t\theta)\sigma$. A substitution θ is more general than σ , denoted by $\theta \leq \sigma$, if $\sigma = \theta\gamma$ for some substitution γ . We write $\theta|_s$ to denote the restriction of the substitution θ to the set of variables in the syntactic object s . A *renaming* is a substitution σ for which there exists the inverse σ^{-1} , such that $\sigma\sigma^{-1} = \sigma^{-1}\sigma = id$. A *unifier* of terms s and t is a substitution ϑ such that $s\vartheta = t\vartheta$. The *most general unifier* of terms s and t , denoted by $\text{mgu}(s, t)$, is a unifier θ such that for each other unifier θ' , $\theta \leq \theta'$.

A *term rewriting system* \mathcal{R} (TRS for short) is a pair (Σ, R) , where R is a finite set of reduction (or rewrite) rule schemes of the form $l \rightarrow r$ such that $l, r \in \mathcal{T}(\Sigma, \mathcal{V})$, $l \notin \mathcal{V}$, and $Var(r) \subseteq Var(l)$. We will often write just R instead of (Σ, R) . For TRS \mathcal{R} , $l \rightarrow r \ll \mathcal{R}$ denotes that $l \rightarrow r$ is a new variant of a rule in R such that $l \rightarrow r$ contains only *fresh* variables, i.e., it contains no variable previously met during any computation (standardized apart). We say that a TRS \mathcal{R} is *left-linear* (*right-linear*) if the left-hand (right-hand) side of any rule $l \rightarrow r \in R$ is a linear term. We say that the TRS \mathcal{R} is *linear* if it is left and right-linear.

A TRS \mathcal{R} is called *topmost* if, for every term t , all rewritings on t are performed at the root position of t . Although topmost TRSs are not commonly used in term rewriting, they are relevant in programming languages. For instance, in Haskell [37] or Maude [10], rewrite rules can be defined so that the type (or sort) information forces rewrites to happen only at the top of

terms. In Maude, it is also possible to introduce freezing specifications that block rewrites at any proper subterm position. Actually, many concurrent systems of interest, including the vast majority of distributed algorithms, admit quite natural topmost specifications [33]. In an unsorted setting like ours, topmost TRSs are only those that do not contain any function symbol whose arity is greater than 0 (that is, all rules have the form $a \rightarrow b$).

Given a TRS $\mathcal{R} = (\Sigma, R)$, we assume that the signature Σ is partitioned into two disjoint sets $\mathcal{D} := \{f \mid f(t_1, \dots, t_n) \rightarrow r \in R\}$ and $\mathcal{C} := \Sigma \setminus \mathcal{D}$. Symbols in \mathcal{C} are called *constructors* and symbols in \mathcal{D} are called *defined symbols* or *functions*. The elements of $\mathcal{T}(\mathcal{C}, \mathcal{V})$ are called *constructor terms*. A *pattern* is a term of the form $f(d_1, \dots, d_k)$ where $f \in \mathcal{D}$ and d_1, \dots, d_k are constructor terms. We say that a TRS is a *constructor term rewriting system* (CS for short) if the left-hand sides of \mathcal{R} are patterns.

A rewrite step is the application of a rewrite rule to an expression. A term $s \in \mathcal{T}(\Sigma, \mathcal{V})$ *rewrites* to a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, denoted by $s \rightarrow_{\mathcal{R}} t$, if there exist $p \in O_{\Sigma}(s)$, $l \rightarrow r \in \mathcal{R}$, and substitution σ such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$. When we want to emphasize the position p where a rewriting step has taken place, we write $s \xrightarrow{p}_{\mathcal{R}} t$. When we want to emphasize that the position q where a rewriting step has taken place is greater than some position p , we write $s \xrightarrow{>p}_{\mathcal{R}} t$.

For substitutions σ, ρ and a set of variables V , we define $\sigma|_V \rightarrow \rho|_V$ if there is $x \in V$ such that $x\sigma \rightarrow x\rho$ and for all other $y \in V$ we have $y\sigma = y\rho$.

A term s is a *normal form* w.r.t. \mathcal{R} (or simply a normal form), if there is no term t such that $s \rightarrow_{\mathcal{R}} t$. We denote the transitive and reflexive closure of $\rightarrow_{\mathcal{R}}$ by $\rightarrow_{\mathcal{R}}^*$. We write $s \xrightarrow{!}_{\mathcal{R}} t$ whenever $s \rightarrow_{\mathcal{R}}^* t$ with t being a normal form. A TRS \mathcal{R} is *terminating* (also called strongly normalizing or noetherian) if there are no infinite reduction sequences $t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} \dots$. In other words, every reduction sequence eventually ends in a normal form. A TRS \mathcal{R} is *confluent* if, whenever $t \rightarrow_{\mathcal{R}}^* s_1$ and $t \rightarrow_{\mathcal{R}}^* s_2$, there exists a term w s.t. $s_1 \rightarrow_{\mathcal{R}}^* w$ and $s_2 \rightarrow_{\mathcal{R}}^* w$. A substitution σ is *normalized* if, for each $x \in \mathcal{V}$, $x\sigma$ is a normal form.

Two (possibly renamed) rules $l \rightarrow r$ and $l' \rightarrow r'$ *overlap*, if there is a non-variable position $p \in O_{\Sigma}(l)$ and a most-general unifier σ such that $l|_p\sigma = l'\sigma$. The pair $\langle (l[r']_p)\sigma, r\sigma \rangle$ is called a *critical pair* and is also called an *overlay* if $p = \epsilon$. A critical pair $\langle t, s \rangle$ is *trivial* if $t = s$. A left-linear TRS without critical pairs is called *orthogonal*. A left-linear TRS where its critical pairs are trivial overlays is called *almost orthogonal*. Note that orthogonality and almost orthogonality of a TRS \mathcal{R} implies confluence of $\rightarrow_{\mathcal{R}}$.

Given a TRS $\mathcal{R} = (\Sigma, R)$, the set of normal forms of \mathcal{R} is denoted by $\text{nf}_{\mathcal{R}}$, and the set of constructor terms (or *values*) of \mathcal{R} is denoted by $\text{eval}_{\mathcal{R}}$. Membership in any of these two sets is decidable. When no confusion can arise, we omit the subscript \mathcal{R} . A different but also relevant set of terms is the set of all possible *reducts* of \mathcal{R} , which is denoted by $\text{red}_{\mathcal{R}} = \{s \in$

$\mathcal{T}(\Sigma, \mathcal{V}) \mid t \in \mathcal{T}(\Sigma, \mathcal{V}), t \rightarrow_{\mathcal{R}}^* s$. Membership in $\text{red}_{\mathcal{R}}$ is also decidable.

In this paper, we are also interested in *rigid normal forms* [3].

2.1 Narrowing and Rigid Normal Forms

Narrowing is a generalization of term rewriting that allows free variables in terms (as in logic programming). In contrast to term rewriting, the left-hand side of a rule is *unified* with the subterm under evaluation in order to (non-deterministically) reduce this term. Narrowing was originally introduced as a mechanism for solving equational unification problems [20] and then generalized to solve the more general problem of symbolic reachability [33]. Since narrowing subsumes both rewriting and SLD-resolution, it is complete in the sense of functional programming (computation of normal forms) as well as logic programming (computation of answers); see [22, 23] for a survey. That is, under appropriate conditions, narrowing is able to find “more general” solutions for the variables of terms s and t , such that s rewrites to t in \mathcal{R} in a number of steps.

Formally, a term $s \in \mathcal{T}(\Sigma, \mathcal{V})$ *narrows* to $t \in \mathcal{T}(\Sigma, \mathcal{V})$, denoted by $s \rightsquigarrow_{\theta, \mathcal{R}} t$ (or simply $s \rightsquigarrow_{\theta} t$) if there exist $p \in O_{\Sigma}(s)$, $l \rightarrow r \ll \mathcal{R}$, and substitution θ such that $\theta = \text{mgu}(s|_p, l)$ and $t = s[r]_p\theta$. When we want to emphasize the position p where a narrowing step has taken place, we write $s \overset{p}{\rightsquigarrow}_{\theta, \mathcal{R}} t$ (or simply $s \overset{p}{\rightsquigarrow}_{\theta} t$). A narrowing *derivation* for t in \mathcal{R} with a (partially computed) answer substitution θ is defined by $t \rightsquigarrow_{\theta, \mathcal{R}}^* t'$ iff $\exists \theta_1, \dots, \theta_n$ s.t. $t \rightsquigarrow_{\theta_1} \dots \rightsquigarrow_{\theta_n} t'$ and $\theta = (\theta_1 \dots \theta_n)|_t$. We say that the derivation has length n .

Example 2.1

Consider the following TRS $\mathcal{R}_{SUM} := \{0 + Y \rightarrow Y, \mathbf{s}(X) + Y \rightarrow \mathbf{s}(X + Y)\}$. The goal $X + Y$ can be narrowed by instantiating X to either 0 or $\mathbf{s}(X')$ in order to apply, respectively, the first or second rewrite rule:

$$\begin{aligned} X + Y &\rightsquigarrow_{\{X/0\}} Y \\ X + Y &\rightsquigarrow_{\{X/\mathbf{s}(X')\}} \mathbf{s}(X' + Y) \end{aligned}$$

Note that the last expression $\mathbf{s}(X' + Y)$ can be further narrowed. Actually, there is an infinite number of narrowing sequences

$$\begin{aligned} \mathbf{s}(X' + Y) &\rightsquigarrow_{\{X'/0\}}^* \mathbf{s}(Y) \\ \mathbf{s}(X' + Y) &\rightsquigarrow_{\{X'/\mathbf{s}(0)\}}^* \mathbf{s}(\mathbf{s}(Y)) \\ \mathbf{s}(X' + Y) &\rightsquigarrow_{\{X'/\mathbf{s}(\mathbf{s}(0))\}}^* \mathbf{s}(\mathbf{s}(\mathbf{s}(Y))) \\ &\vdots \end{aligned}$$

For the reachability problem $\exists X, Y$ s.t. $X + Y \rightarrow^* \mathbf{s}(Y)$, narrowing delivers the answer substitution $\{X/\mathbf{s}(0)\}$.

Strong reachability-completeness of narrowing (i.e., completeness w.r.t. not necessarily normalized solutions) means that for every solution to a reachability goal, a more general solution (*modulo* \mathcal{R}) is computed by narrowing. In the reachability setting, where confluence cannot be assumed, this means that, for each pair $t, t' \in \mathcal{T}(\Sigma, \mathcal{V})$ and substitution ρ such that $t\rho \rightarrow_{\mathcal{R}}^* t'$, there are substitutions η, ρ', θ and a term $t'' \in \mathcal{T}(\Sigma, \mathcal{V})$ such that $t \rightsquigarrow_{\eta, \mathcal{R}}^* t''$, $\rho|_t \rightarrow_{\mathcal{R}}^* \rho'$, $\rho' = (\eta\theta)|_t$, and $t' = t''\theta$. In other words, there is a reduct ρ' of solution ρ such that ρ' is a (syntactic) instance of the substitution η computed by narrowing. In [33], strong reachability-completeness is proved to hold only in particular classes of TRS:

1. topmost TRSs and
2. right-linear TRSs (restricted to linear input terms).

The terminology “complete TRSs” is used in [27, 29, 34] to refer to the well-known class of TRSs where narrowing is complete as a procedure for solving equations (namely the class of confluent and terminating TRSs). The following definition adapts this idea to the reachability setting.

Definition 2.2 (Reachability-complete TRS) *A TRS \mathcal{R} is reachability-complete iff narrowing is strongly reachability-complete for \mathcal{R} .*

Let us now introduce the notion of *rigid normal form (rnf)*, which is the most important ingredient in Section 4.2.3 for effectively computing a compact semantics by deploying narrowing computations. Actually, we will only collect in the denotation semantic rules whose right-hand sides (RHSs) are rigid normal forms, since narrowing terminates for wide classes of TRSs where the RHSs of the rewrite rules satisfy this condition (see [3] and Corollary 4.9 below).

Definition 2.3 (Rigid normal form [3]) *Given a TRS $\mathcal{R} = (\Sigma, R)$, a term s is a rigid normal form (rnf) if there is no substitution θ and position p such that $s \overset{p}{\rightsquigarrow}_{\theta, \mathcal{R}} t$.*

Note that the notion of rigid normal form (rnf) is stronger than the standard notion of (rewriting) normal form but can still be easily decided by simply checking that no subterm of the considered term unifies with the left-hand side (LHS) of any rule in \mathcal{R} .

Example 2.4

Consider the TRS \mathcal{R}_{ID} of Example 1.3, and let $\mathcal{R}'_{ID} := \mathcal{R}_{ID} - \{\text{id}(X) \rightarrow X\}$, i.e., $\mathcal{R}'_{ID} = \{\text{id}(0) \rightarrow 0, \text{id}(s(X)) \rightarrow s(\text{id}(X))\}$. The term $\text{id}(X)$ is not a normal form of \mathcal{R}_{ID} , whereas it is a normal form of \mathcal{R}'_{ID} . However, $\text{id}(X)$ is not a rigid normal form of \mathcal{R}'_{ID} , since it can be unified with $\text{id}(0)$ and $\text{id}(s(Y))$.

Nevertheless, if we add to the signature of \mathcal{R}'_{ID} a fresh unary constructor symbol c , then the term $\text{id}(c(Y))$ (which happens to be an instance of

$\text{id}(X)$ is a normal form as well as a rigid normal form.

The set of rigid normal forms of \mathcal{R} is denoted by $\text{rnf}_{\mathcal{R}}$. Actually, $\text{eval}_{\mathcal{R}} \subseteq \text{rnf}_{\mathcal{R}} \subseteq \text{nf}_{\mathcal{R}} \subseteq \text{red}_{\mathcal{R}}$. Note that when we restrict ourselves to ground terms, normal and rigid normal forms coincide, i.e., $\text{nf}_{\mathcal{R}} \cap \mathcal{T}(\Sigma) = \text{rnf}_{\mathcal{R}} \cap \mathcal{T}(\Sigma)$.

3 A compact, goal-independent rewriting semantics

A compact rewriting semantics that is goal-independent should ideally include only semantic rules for expressions rooted by a “defined” symbol and “relevant” arguments. Essentially, our semantics is based on the idea of collecting only those (unfolded) rules that are not a rewriting consequence of other elements in the semantics; in other words, the compact semantics is obtained by collecting only a representative set of the “most general” rewriting sequences. As in the Ω -semantics of [8] and the narrowing-based computed answers semantics of [4] recalled above, we need to introduce variables in the denotation instead of collecting the meaning of ground expressions. This allows us to see the semantics (when it might be convenient) as a “more efficient program” where it is still possible to run any input expression.

Example 3.1

Consider again the TRS \mathcal{R}_{INC} of Example 1.1. A naïve, non-ground semantics for this program can be obtained by selecting the following elements from the big-step semantics:

$$\{\text{inc}(s^n(t)) \mapsto s^{n+1}(t), \text{plus2}(s^n(t)) \mapsto s^{n+2}(t) \mid n \geq 0, t \in \{0, X\}\}.$$

However, we prefer the “most compact” one

$$\{\text{inc}(X) \mapsto s(X), \text{plus2}(X) \mapsto s(s(X))\},$$

which still captures the meaning of ground input expressions.

The problem of computing a highly compact representation for the semantics is far from trivial. Obviously, the naïve solution based on starting from the (generally infinite) coarse semantics, and then trying to filter out all redundant semantic rules (e.g. by developing program optimization/synthesis techniques to infer “most general rules”) is not an option. The alternative solution to start from the original rewrite rules and specialize them by rewriting-based, folding/unfolding transformations might end up in undesired, infinite computations without redundancy being removed. Examples are shown below where the intended compact semantics could not

be obtained by using this methodology. In the following, we effectively define a suitable compact semantics for TRSs in the fixpoint style. Unlike the big-step semantics, ours is truly “goal-independent”.

3.1 Rewriting Consequences

Given a signature Σ , we denote by \mathbb{W}_Σ the set of all possible semantic rules (up to renaming) built with the elements of Σ , i.e., $\mathbb{W}_\Sigma = \{t \mapsto s \mid t, s \in \mathcal{T}(\Sigma, \mathcal{V})\} / \equiv$ where \equiv is the variance relation extended to semantic rules, i.e., $(t \mapsto s) \equiv (t' \mapsto s')$ iff there exists a renaming substitution ρ s.t. $(t \mapsto s)\rho = t' \mapsto s'$. We simply write \mathbb{W} when no confusion about Σ can arise.

In the following, any $I \subseteq \mathbb{W}$ is implicitly considered as an arbitrary set of semantic rules obtained by choosing an arbitrary representative of the elements of I in the equivalence class generated by \equiv . Actually, in the following, all the operators that we use on \mathbb{W} are also independent of the choice of the representative. Therefore, we can define any operator on \mathbb{W} in terms of its counterpart defined on sets of semantic rules, and denote the corresponding operators by the same name.

In order to be as general as possible, we will use a family of reference (ground) big-step rewriting semantics for a TRS \mathcal{R} , parametric on a set BT of terms known as *blocking terms*, where $BT \in \{\text{eval}_{\mathcal{R}}, \text{rnf}_{\mathcal{R}}, \text{nf}_{\mathcal{R}}, \text{red}_{\mathcal{R}}\}$. They model different (relevant) “*rewriting behaviors*” (often also called *observable properties*) which are properties that can be observed in the computation space (behavior) of \mathcal{R} . Our goal is to define suitable families of (compressed) goal-independent semantics that are *correct* w.r.t. these behaviors.

Definition 3.2 (Ground big-step behaviour) *Given a TRS \mathcal{R} and a set BT of blocking terms, the ground big-step semantics $\mathcal{B}_{BT}(\mathcal{R})$ of \mathcal{R} w.r.t. BT is*

$$\mathcal{B}_{BT}(\mathcal{R}) := \{s \mapsto t \in \mathbb{W} \mid s \in \mathcal{T}(\Sigma), s \rightarrow_{\mathcal{R}}^* t, t \in BT\}. \quad (3.1)$$

Note that $\mathcal{B}_{\text{eval}}(\mathcal{R}) \subseteq \mathcal{B}_{\text{rnf}}(\mathcal{R}) = \mathcal{B}_{\text{nf}}(\mathcal{R}) \subseteq \mathcal{B}_{\text{red}}(\mathcal{R})$.

In the case of TRSs and functional programs, all the BT sets introduced above can be relevant but lead to different standard rewriting semantics. For instance, $\mathcal{B}_{\text{nf}}(\mathcal{R})$ is the big-step semantics of term rewriting systems whereas $\mathcal{B}_{\text{eval}}(\mathcal{R})$ corresponds to the standard big-step semantics associated to functional programs. Actually, normal forms are not necessarily the interesting results of functional computations [23], as the following example shows.

Example 3.3

Consider the TRS $\mathcal{R} := \{\text{head}(\text{cons}(X, Y)) \rightarrow X\}$, which returns the first element of a non-empty list. Then, a (rigid) normal form like “`head(nil)`” is

usually considered to be an error rather than a result. Actually, Haskell [38] reports an error for evaluating the term “`head(nil)`” rather than delivering the (rigid) normal form “`head(nil)`”.

Let us stress that values are not necessarily the interesting results of TRS computations and, thus, in the sequel we want to develop semantics that can tackle both the values and rigid normal form cases.

As we have already shown, even when collecting a subset of all possible rewritings (e.g. only those rewritings leading to a value or normal form), we still keep many “useless” elements that produce redundant information and cause useless overhead³. These elements arise as a natural consequence of the main “compositional” properties of rewriting:

stability i.e., the property of being closed under substitution: $s \rightarrow_{\mathcal{R}} t$ implies $s\sigma \rightarrow_{\mathcal{R}} t\sigma$, for every substitution σ , and

replacement i.e., the compatibility with contexts: $t \rightarrow_{\mathcal{R}} s$ implies $u[t]_p \rightarrow_{\mathcal{R}} u[s]_p$, for all u and p .

When we additionally consider the transitivity of rewriting, we have the notion of a sequence $t \rightarrow_{\mathcal{R}}^* s$ implied by the demonstration of several sequences $v_1 \rightarrow_{\mathcal{R}}^* v'_1, \dots, v_k \rightarrow_{\mathcal{R}}^* v'_k$. For instance, for the TRS \mathcal{R}_{INC} of Example 1.1, we have that $\text{inc}(\text{plus2}(\mathbf{s}(0))) \rightarrow_{\mathcal{R}}^* \mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{s}(0))))$ is implied by $\text{inc}(X) \rightarrow_{\mathcal{R}} \mathbf{s}(X)$ and $\text{plus2}(X) \rightarrow_{\mathcal{R}} \text{inc}(\text{inc}(X)) \rightarrow_{\mathcal{R}} \text{inc}(\mathbf{s}(X)) \rightarrow_{\mathcal{R}} \mathbf{s}(\mathbf{s}(X))$.

The key idea of the paper is to build a compact semantics by collecting only those rules (unfolded by narrowing) that are not a *rewriting consequence* of other rules in the semantics.

Definition 3.4 (Rewriting consequence) *Given $R \subseteq \mathbb{W}$, we say that the semantic rule $t \mapsto s \in \mathbb{W}$ is a rewriting consequence of R , denoted by $R \vdash (t \mapsto s)$, if t rewrites to s in R in a finite number of steps. Note that the denotation element $t \mapsto t$ with $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is implied by any set R of rules.*

This definition is easily extended to sets, i.e., $R \vdash I$ if for each $t \mapsto s \in I$, $R \vdash t \mapsto s$.

Example 3.5

Consider again the TRS \mathcal{R}_{SUM} of Example 2.1. Some examples of rewriting consequences are

$$\{0 + Y \mapsto Y\} \vdash \mathbf{s}(0 + \mathbf{s}(0 + X)) \mapsto \mathbf{s}(\mathbf{s}(X))$$

³For a goal-dependent semantics that collects the meaning of a set of main calls, the redundancy of the semantics might be less harmful. However, the compression can still bring significant improvement, since we keep only the “most general” calls rather than all of them.

$$\{0 + Y \mapsto Y\} \vdash s(0 + s(0 + X)) \mapsto s(0 + s(X))$$

Some negative examples are

$$\begin{aligned} \{0 + Y \mapsto Y\} &\not\vdash (X + 0) + 0 \mapsto X \\ \{0 + Y \mapsto Y\} &\not\vdash s(X + 0) \mapsto s(X) \end{aligned}$$

as $X + 0$ cannot be rewritten in any way.

In order to achieve a compact version of the semantics based on “more general” rules (with variables), we introduce a non-ground generalization of our big-step rewriting semantics as the basis.

Definition 3.6 (Non-ground big-step behaviour) *Given a TRS \mathcal{R} and a set BT of blocking terms, the non-ground big-step semantics $\mathcal{B}_{BT}^{\mathcal{V}}(\mathcal{R})$ of \mathcal{R} w.r.t. BT is*

$$\mathcal{B}_{BT}^{\mathcal{V}}(\mathcal{R}) := \{s \mapsto t \in \mathbb{W} \mid s \in \mathcal{T}(\Sigma, \mathcal{V}), s \rightarrow_{\mathcal{R}}^* t, t \in BT\}. \quad (3.2)$$

Note that $\mathcal{B}_{\text{eval}}^{\mathcal{V}}(\mathcal{R}) \subseteq \mathcal{B}_{\text{rnf}}^{\mathcal{V}}(\mathcal{R}) \subseteq \mathcal{B}_{\text{nf}}^{\mathcal{V}}(\mathcal{R}) \subseteq \mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})$.

In the following, we show how to build the compact (“zipped”) denotation of the rewriting semantics of a TRS \mathcal{R} by getting rid of superfluous rewriting consequences found in the (non-ground) semantics $\mathcal{B}_{BT}^{\mathcal{V}}(\mathcal{R})$.

3.2 Zip and unzip operators

In this section we introduce the operators zip/unzip that we will use to compress/decompress denotations.

Definition 3.7 (unzip and zip operations) *Given $I \subseteq \mathbb{W}$, we define the set of rewriting consequences of I as*

$$\text{unzip}(I) := \{r \in \mathbb{W} \mid I \vdash r\}. \quad (3.3)$$

We say that I is closed under rewriting consequences (or, more briefly, unzipped) whenever $\text{unzip}(I) = I$.

Conversely, we define the non-redundant subset of I as

$$\text{zip}(I) := \{r \in I \mid I - \{r\} \not\vdash r\}. \quad (3.4)$$

We say that I is zipped whenever $\text{zip}(I) = I$.

Roughly speaking, $\text{zip}(I)$ throws away all “redundant elements” of I . Let us illustrate the above definition by means of an example.

Example 3.8

Consider again the TRS \mathcal{R}_{ID} and \mathcal{R}_{SUM} of Examples 1.3 and 2.1, respectively. We have

$$\begin{aligned}
\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R}_{ID}) &= \text{unzip}(\mathcal{R}_{ID}) \\
\mathcal{B}_{\text{nf}}^{\mathcal{V}}(\mathcal{R}_{ID}) &= \mathcal{B}_{\text{rnf}}^{\mathcal{V}}(\mathcal{R}_{ID}) = \mathcal{B}_{\text{eval}}^{\mathcal{V}}(\mathcal{R}_{ID}) \\
&= \text{unzip}(\{\text{id}(\mathbf{X}) \mapsto \mathbf{X}\}) \cap (\mathcal{T}(\Sigma, \mathcal{V}) \times \text{nf}) \\
\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R}_{SUM}) &= \text{unzip}(\mathcal{R}_{SUM}) \\
\mathcal{B}_{\text{rnf}}^{\mathcal{V}}(\mathcal{R}_{SUM}) &= \mathcal{B}_{\text{eval}}^{\mathcal{V}}(\mathcal{R}_{SUM}) \\
&= \text{unzip}(\{\mathbf{s}^n(0) + \mathbf{X} \mapsto \mathbf{s}^n(\mathbf{X}) \mid n \geq 0\}) \\
&\quad \cap (\mathcal{T}(\Sigma, \mathcal{V}) \times \text{rnf}) \\
\text{zip}(\mathcal{B}_{\text{eval}}^{\mathcal{V}}(\mathcal{R}_{ID})) &= \text{zip}(\mathcal{B}_{\text{rnf}}^{\mathcal{V}}(\mathcal{R}_{ID})) = \text{zip}(\mathcal{B}_{\text{nf}}^{\mathcal{V}}(\mathcal{R}_{ID})) = \{\text{id}(\mathbf{X}) \mapsto \mathbf{X}\} \\
\text{zip}(\mathcal{B}_{\text{nf}}^{\mathcal{V}}(\mathcal{R}_{SUM})) &= \mathcal{R}_{SUM} \\
\text{zip}(\mathcal{B}_{\text{eval}}^{\mathcal{V}}(\mathcal{R}_{SUM})) &= \text{zip}(\mathcal{B}_{\text{rnf}}^{\mathcal{V}}(\mathcal{R}_{SUM})) \\
&= \{\mathbf{s}^n(0) + \mathbf{X} \mapsto \mathbf{s}^n(\mathbf{X}) \mid n \geq 0\}
\end{aligned}$$

Note that $\mathcal{B}_{\text{nf}}^{\mathcal{V}}(\mathcal{R}_{SUM}) \neq \mathcal{B}_{\text{rnf}}^{\mathcal{V}}(\mathcal{R}_{SUM})$ because $\mathbf{X} + \mathbf{Y} \mapsto \mathbf{X} + \mathbf{Y} \in \mathcal{B}_{\text{nf}}^{\mathcal{V}}(\mathcal{R}_{SUM})$ but $\mathbf{X} + \mathbf{Y}$ is not a rigid normal form.

Usually $\text{zip}(I) \vdash I$ but, unfortunately, there are some “pathological” circumstances when the zip operator removes more elements than the necessary, in the sense that the result is no longer able to regenerate the argument (i.e., $\text{zip}(I) \not\vdash I$). This may happen, for instance, when I has mutually recursive rewriting dependencies.

Example 3.9

Consider the TRS $\mathcal{R} := \{a \rightarrow b, b \rightarrow c, c \rightarrow a\}$ and the set $I := \{a \mapsto a, a \mapsto b, a \mapsto c, b \mapsto a, b \mapsto b, b \mapsto c, c \mapsto a, c \mapsto b, c \mapsto c\}$. Note that $I = \mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})$. We have $\text{zip}(I) = \emptyset$, since I has “circular dependencies”, in the sense that every semantic rule in I is a rewriting consequence of the others, i.e., $\forall t \mapsto s \in I, I - \{t \mapsto s\} \vdash t \mapsto s$.

In the following, we restrict our interest to *compactable denotations*, i.e., sets that maintain all the relevant rewriting consequences under zip. In Appendix B we study the general case where interpretations are not necessarily compactable.

Definition 3.10 (Compactable Denotation) *We say that $I \subseteq \mathbb{W}$ is compactable if $\text{zip}(I) \vdash I$.*

Note that any zipped set is compactable, while unzipped ones may not (as shown in Example 3.9).

Now, we state some general properties of zip and unzip.

Proposition 3.11 *Let $I, I' \subseteq \mathbb{W}$ be sets of rules.*

1. *unzip and zip are idempotent, i.e., $\text{unzip}(\text{unzip}(I)) = \text{unzip}(I)$ and $\text{zip}(\text{zip}(I)) = \text{zip}(I)$.*
2. *unzip is monotone w.r.t. \subseteq , i.e., $I \subseteq I' \implies \text{unzip}(I) \subseteq \text{unzip}(I')$.*
3. *unzip is extensive w.r.t. \subseteq , i.e., $I \subseteq \text{unzip}(I)$.*
4. *zip is reductive w.r.t. \subseteq , i.e., $\text{zip}(I) \subseteq I$.*
5. *$\text{zip}(\text{unzip}(I)) = \text{zip}(I)$.*
6. *$\text{zip}(\text{unzip})$ is reductive w.r.t. \subseteq , i.e., $\text{zip}(\text{unzip}(I)) \subseteq I$.*
7. *$\text{zip}(\text{unzip}(\text{zip}(I))) = \text{zip}(I)$.*
8. *$\text{zip}(I)$ is compactable.*
9. *I is compactable if and only if $\text{unzip}(I)$ is compactable.*

Moreover if I is compactable

10. *$I \subseteq \text{unzip}(\text{zip}(I))$.*
11. *$\text{unzip}(\text{zip}(\text{unzip}(I))) = \text{unzip}(I)$.*

Proof. Points 1, 3 and 4 are straightforward. Point 2 is immediate, since the rewriting relation for I is included in the rewriting relation for I' , i.e., $\rightarrow_I \subseteq \rightarrow_{I'}$.

For Point 5, first observe that if $I \subseteq I'$ and $\forall e \in I' \setminus I, I \vdash e$, then $\text{zip}(I) = \text{zip}(I')$. We have that, by Point 3, $I \subseteq \text{unzip}(I)$, and thus, $\text{zip}(I) = \text{zip}(\text{unzip}(I))$ by the property we have just observed.

Point 6 is immediate, by applying Point 5 first, and then Point 4.

For Point 7, we have that, by Point 5, $\text{zip}(\text{zip}(I)) = \text{zip}(\text{unzip}(\text{zip}(I)))$. We conclude that $\text{zip}(I) = \text{zip}(\text{unzip}(\text{zip}(I)))$ by Point 1.

For Point 8, we have that, by Point 1, $\text{zip}(\text{zip}(I)) = \text{zip}(I) \vdash \text{zip}(I)$.

For Point 9, if $\text{unzip}(I)$ is compactable, then, by definition, $\text{zip}(\text{unzip}(I)) \vdash \text{unzip}(I)$. By Point 5, $\text{zip}(I) = \text{zip}(\text{unzip}(I))$. Thus $\text{zip}(I) \vdash \text{unzip}(I) \supseteq I$. Vice versa, if I is compactable, then $\text{zip}(I) \vdash I$ and by Point 5 $\text{zip}(\text{unzip}(I)) = \text{zip}(I)$. Thus, $\text{zip}(\text{unzip}(I)) \vdash I \vdash \text{unzip}(I)$.

For Point 10, for $e \in I$ we have that, by hypothesis, $\text{zip}(I) \vdash e$. Thus, by Definition 3.7, $e \in \text{unzip}(\text{zip}(I))$.

For Point 11, we have, by Point 10, that $\text{unzip}(I) \subseteq \text{unzip}(\text{zip}(\text{unzip}(I)))$. Moreover, by Point 4, $\text{zip}(\text{unzip}(I)) \subseteq \text{unzip}(I)$ and thus, by Point 2, $\text{unzip}(\text{zip}(\text{unzip}(I))) \subseteq \text{unzip}(\text{unzip}(I))$. Hence, by Point 1, $\text{unzip}(\text{zip}(\text{unzip}(I))) \subseteq \text{unzip}(I)$. ■

Observation 3.12 *There is an isomorphism between the class of zipped sets and the class of compactable unzipped sets. Indeed, let us consider the class of zipped sets $z\mathbb{W} := \{Z \subseteq \mathbb{W} \mid Z = \text{zip}(Z)\}$ and the class of compactable unzipped sets $uc\mathbb{W} := \{U \subseteq \mathbb{W} \mid U \text{ is compactable, } U = \text{unzip}(U)\}$. By Point 8, $z\mathbb{W}$ contains only compactable sets. Then, zip and unzip are inverse functions on these classes, since by Point 7 it holds that for all $Z \in z\mathbb{W}$, $\text{zip}(\text{unzip}(Z)) = Z$, and by Point 11 for all $U \in uc\mathbb{W}$, $\text{unzip}(\text{zip}(U)) = U$.*

Note that, though unzip is monotone w.r.t. \subseteq (Point 2 of Proposition 3.11), zip is neither monotone nor anti-monotone w.r.t. \subseteq , as shown in the following example.

Example 3.13

Consider the TRSs \mathcal{R}_{ID} and \mathcal{R}'_{ID} of Example 2.4. Then, we have that $\mathcal{B}_{\text{eval}}^{\mathcal{V}}(\mathcal{R}'_{ID}) = \mathcal{B}_{\text{eval}}^{\mathcal{V}}(\mathcal{R}_{ID}) - \{\text{id}(X) \mapsto X\}$ whereas $\mathcal{B}_{\text{eval}}(\mathcal{R}'_{ID}) = \mathcal{B}_{\text{eval}}(\mathcal{R}_{ID})$. However, the compressed sets

$$\begin{aligned} \text{zip}(\mathcal{B}_{\text{eval}}^{\mathcal{V}}(\mathcal{R}_{ID})) &= \{\text{id}(X) \mapsto X\} \\ \text{zip}(\mathcal{B}_{\text{eval}}^{\mathcal{V}}(\mathcal{R}'_{ID})) &= \{\text{id}(s^k(0)) \mapsto s^k(0) \mid k \geq 0\} \end{aligned}$$

are not contained in each other.

In the following, we show that, the higher a BT set is in the chain $\text{eval}_{\mathcal{R}} \subseteq \text{rnf}_{\mathcal{R}} \subseteq \text{nf}_{\mathcal{R}} \subseteq \text{red}_{\mathcal{R}}$, the less effectiveness it provides when compacting the corresponding denotation.

3.3 Relevant observables

In this section, we show that only the observables modeled by $\text{eval}_{\mathcal{R}}$ and $\text{rnf}_{\mathcal{R}}$ are serviceable in computing compact semantics. This claim is based on the following points, explained in detail below:

1. The semantics $\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})$ and $\mathcal{B}_{\text{nf}}^{\mathcal{V}}(\mathcal{R})$ are not useful when we look for a compact denotation that is computationally “richer” and “faster” than \mathcal{R} itself.
2. The semantics $\mathcal{B}_{\text{eval}}^{\mathcal{V}}(\mathcal{R})$ and $\mathcal{B}_{\text{rnf}}^{\mathcal{V}}(\mathcal{R})$ are compactable.

Let us first show that the semantics of reducts of a TRS is ineffective, in the sense that its compression delivers (a subset of) the TRS itself as compressed semantics. This can still have the advantage of removing some “redundant rules” in some cases, but it generally implies that no speed-up is possible by “computing” in the semantics $\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})$, since it is almost equivalent to rewriting with the original set of rewrite rules. Being able to move as much computation as possible in the semantics is *particularly* relevant for Abstract Interpretation, since it reduces the total amount of abstract iteration steps needed to compute the fixpoint approximation and also improves the precision of the induced abstract semantics.

Proposition 3.14 *Given a TRS \mathcal{R} s.t. $\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})$ is compactable,*

- $\text{unzip}(\mathcal{R}) = \mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})$ and
- $\text{zip}(\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})) \subseteq \mathcal{R}$.

If we additionally require \mathcal{R} to be orthogonal, then $\text{zip}(\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})) = \mathcal{R}$.

Proof. By Definition 3.7, $\text{unzip}(\mathcal{R}) = \{r \in \mathbb{W} \mid \mathcal{R} \vdash r\} = \{s \mapsto t \in \mathbb{W} \mid s \in \mathcal{T}(\Sigma, \mathcal{V}), s \rightarrow_{\mathcal{R}}^* t, t \in \text{red}\}$, and by Definition 3.6, $\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R}) = \{s \mapsto t \in \mathbb{W} \mid s \in \mathcal{T}(\Sigma, \mathcal{V}), s \rightarrow_{\mathcal{R}}^* t, t \in \text{red}\}$. Thus, it is proved that $\text{unzip}(\mathcal{R}) = \mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})$.

We prove $\text{zip}(\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})) \subseteq \mathcal{R}$ by contradiction. Take $t \mapsto s \in \text{zip}(\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R}))$ and assume that $t \mapsto s \notin \mathcal{R}$ (modulo renaming). Since $t \rightarrow_{\mathcal{R}}^* s$, $\exists k \geq 1$, $l_1 \rightarrow r_1, \dots, l_k \rightarrow r_k \in \mathcal{R}$, and $u_1, \dots, u_{k-1} \in \mathcal{T}(\Sigma, \mathcal{V})$ s.t. $t \rightarrow_{l_1 \rightarrow r_1} u_1 \cdots u_{k-1} \rightarrow_{l_k \rightarrow r_k} s$. Given $L = \{l_1 \mapsto r_1, \dots, l_k \mapsto r_k\}$, $L \vdash t \mapsto s$. But we have that $L \subseteq \mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})$ and thus $\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R}) \vdash t \mapsto s$. Since $t \mapsto s \notin \mathcal{R}$, we have that $t \mapsto s \notin L$ and thus $\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R}) - \{t \mapsto s\} \vdash t \mapsto s$, which contradicts that $t \mapsto s \in \text{zip}(\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R}))$.

Finally, we prove by contradiction that $\mathcal{R} \subseteq \text{zip}(\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R}))$ if \mathcal{R} is orthogonal. Consider $l \mapsto r \in \mathcal{R}$ and assume that $l \mapsto r \notin \text{zip}(\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R}))$. Since $l \mapsto r \in \mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})$ and $\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})$ is compactable, we have that $\text{zip}(\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})) \vdash l \mapsto r$. Note that for any orthogonal set \mathcal{R}' of rules and $l' \mapsto r' \in \mathcal{R}'$, $(\mathcal{R}' - \{l' \mapsto r'\}) \not\vdash l' \mapsto r'$, since the LHS's of any pair of rules in \mathcal{R}' cannot unify. Then, $(\mathcal{R} - \{l \mapsto r\}) \not\vdash l \mapsto r$ and, since $\text{zip}(\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})) \subseteq \mathcal{R}$ and $l \mapsto r \notin \text{zip}(\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R}))$, we have $\text{zip}(\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})) \not\vdash l \mapsto r$, which contradicts $\text{zip}(\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})) \vdash l \mapsto r$. ■

Let us now show that, for similar reasons, the normal-form semantics is also pointless.

Proposition 3.15 *Let \mathcal{R} be a TRS where $\mathcal{B}_{\text{nf}}^{\mathcal{V}}(\mathcal{R})$ is compactable and for each rule $l \mapsto r \in \mathcal{R}$, $r \in \text{nf}$. Then, $\text{zip}(\mathcal{B}_{\text{nf}}^{\mathcal{V}}(\mathcal{R})) \subseteq \mathcal{R}$. If we additionally require \mathcal{R} to be orthogonal, then $\text{zip}(\mathcal{B}_{\text{nf}}^{\mathcal{V}}(\mathcal{R})) = \mathcal{R}$.*

Proof. The proof is perfectly analogous to the proof of Proposition 3.14 by considering that if for each rule $l \mapsto r \in \mathcal{R}$, $r \in \text{nf}$, then $\mathcal{R} \subseteq \mathcal{B}_{\text{nf}}^{\mathcal{V}}(\mathcal{R})$. ■

Fortunately, since $\mathcal{B}_{\text{nf}}(\mathcal{R}) = \mathcal{B}_{\text{rnf}}(\mathcal{R})$, by focusing on the observable $\text{rnf}_{\mathcal{R}}$ instead of $\text{nf}_{\mathcal{R}}$, we are still able to capture the (ground) normal form observable of \mathcal{R} (hence also the values) without degenerating into “useless” compact representations as above.

Now we prove that $\mathcal{B}_{\text{eval}}^{\mathcal{V}}(\mathcal{R})$ and $\mathcal{B}_{\text{rnf}}^{\mathcal{V}}(\mathcal{R})$ are compactable.

Lemma 3.16 *For $BT \in \{\text{eval}_{\mathcal{R}}, \text{rnf}_{\mathcal{R}}\}$, any subset $I \subseteq \mathcal{B}_{BT}^{\mathcal{V}}(\mathcal{R})$ is compactable.*

Proof. By contradiction. Let us assume that there is $t \mapsto s \in I$ s.t. $\text{zip}(I) \not\vdash t \mapsto s$. Let $I' := I - \{t \mapsto s\}$. Note that $\text{zip}(I) \not\vdash t \mapsto s$ implies $t \mapsto s \notin \text{zip}(I)$ and, by definition of zip , $I' \vdash t \mapsto s$. Since $\text{zip}(I) \not\vdash t \mapsto s$ and $I' \vdash t \mapsto s$, there is a rewrite sequence $\alpha : t \rightarrow_{I'}^* s$ and a rule $t' \mapsto s' \in I'$ s.t. $\text{zip}(I) \not\vdash t' \mapsto s'$ and α contains at least one use of rule $t' \mapsto s'$. Let $I'' := I - \{t' \mapsto s'\}$. Note that $\text{zip}(I) \not\vdash t' \mapsto s'$ implies $t' \mapsto s' \notin \text{zip}(I)$ and, by definition of zip , $I'' \vdash t' \mapsto s'$. Indeed, there is a rewrite sequence $\beta : t' \rightarrow_{I''}^* s'$ and we can replace every occurrence of the rule $t' \mapsto s'$ in the sequence α by β , yielding a new sequence α' . Note that β may use rule $t \mapsto s$, since $t \mapsto s \in I''$. We now consider whether β uses the rule $t \mapsto s$ or not.

1. If β does contain the rule $t \mapsto s$, then there is a “circular dependency” between $t \mapsto s$ and $t' \mapsto s'$ and the length of β must be greater than 1; otherwise, there is a renaming between $t \mapsto s$ and $t' \mapsto s'$ but this leads to a contradiction because we consider rules in $\mathcal{B}_{BT}^{\mathcal{V}}(\mathcal{R})$ different up to renaming. Now, we can replace every occurrence of the rule $t \mapsto s$ in the sequence α' by α , yielding a new α'' that contains some occurrences of the rule $t \mapsto s$ and this can be repeated infinitely many times. However, this gives a contradiction because, for $BT \in \{\text{eval}_{\mathcal{R}}, \text{rnf}_{\mathcal{R}}\}$, any subset $A \subseteq \mathcal{B}_{BT}^{\mathcal{V}}(\mathcal{R})$ is terminating, i.e., there are no infinite rewriting sequences using the rules in A .
2. If β does not contain any use of the rule $t \mapsto s$, then, since $\text{zip}(I) \not\vdash t' \mapsto s'$ and $I'' \vdash t' \mapsto s'$, there is a rule $t'' \mapsto s'' \in I''$ s.t. $\text{zip}(I) \not\vdash t'' \mapsto s''$ and β contains at least one use of rule $t'' \mapsto s''$. Let $I''' := I - \{t'' \mapsto s''\}$. Note that $\text{zip}(I) \not\vdash t'' \mapsto s''$ implies $t'' \mapsto s'' \notin \text{zip}(I)$ and, by definition of zip , $I''' \vdash t'' \mapsto s''$. Indeed, there is a rewrite sequence $\beta' : t'' \rightarrow_{I'''}^* s''$ and we can replace every occurrence of the rule $t'' \mapsto s''$ in the sequence α' by β' , yielding a new sequence α'' . Note that β' may use rule $t' \mapsto s'$, since $t' \mapsto s' \in I'''$. Then, we can consider whether β' uses the rule $t' \mapsto s'$ or not. The same reasoning of the whole proof can be repeated again, ending in case 1 or applying case 2 infinitely many times. If we apply case 2 infinitely many times, then we have two possibilities: either (i) the resulting sequence after all those infinitely many replacements is infinite (in length); or (ii) the resulting sequence after all those infinitely many replacements is finite (in length).
 - (i) If the resulting sequence is infinite, then there is a contradiction because, for $BT \in \{\text{eval}_{\mathcal{R}}, \text{rnf}_{\mathcal{R}}\}$, any subset $A \subseteq \mathcal{B}_{BT}^{\mathcal{V}}(\mathcal{R})$ is terminating, i.e., there are no infinite rewriting sequences using the rules in A .
 - (ii) If the resulting sequence is finite, then what happens is that the successive sequences β can be split into two sets: (a) a finite set of those successive sequences β whose length is greater than 1

and (b) an infinite set of those successive sequences β that have length 1. That is, we consider a step in the sequence of infinitely many applications of case 2 where we have a sequence $\hat{\alpha} : t \rightarrow_{\hat{I}}^* s$ and two rules $\hat{t} \mapsto \hat{s}$, $\hat{t}' \mapsto \hat{s}'$ such that $\hat{\alpha}$ does not use rule $t \mapsto s$ but uses rule $\hat{t} \mapsto \hat{s}$, $\hat{I} := I - \{\hat{t} \mapsto \hat{s}\}$, $\hat{\beta} : \hat{t} \rightarrow_{\hat{I}}^* \hat{s}$, β contains at least one use of rule $\hat{t}' \mapsto \hat{s}'$, and $\hat{\beta}$ has length 1. However, if we replace $\hat{t} \mapsto \hat{s}$ in $\hat{\alpha}$ by $\hat{\beta}$, which has length 1, then there is a substitution ρ such that $\hat{t}'\rho = \hat{t}$ and $\hat{s}'\rho = \hat{s}$. Note that ρ cannot be a renaming, since we consider rules in $\mathcal{B}_{BT}^{\mathcal{V}}(\mathcal{R})$ that are different up to renaming along the infinitely many applications of case 2. Finally, there is a contradiction in case (ii) because it is impossible to keep using rules whose left-hand sides are strictly more and more general than the previous ones, since we consider only finite terms in all rewrite sequences. ■

This result enables $\mathcal{B}_{\text{eval}}(\mathcal{R})$ and $\mathcal{B}_{\text{rnf}}(\mathcal{R})$ as the semantics of choice for compression-based optimization techniques.

4 The compressed semantics

In this section, we define the compact goal-independent semantics for functional programs in the fixpoint style. First, we formalize our semantic domain. We do this by:

1. taking the most compact (i.e., zipped) representations of \mathbb{W} , and
2. providing these sets with an adequate ordering.

4.1 The Semantic Domain

It is easy to see that semantics $\mathcal{B}_{BT}(\mathcal{R})$ and $\mathcal{B}_{BT}^{\mathcal{V}}(\mathcal{R})$ are closed under rewriting consequences for $BT \in \{\text{eval}_{\mathcal{R}}, \text{rnf}_{\mathcal{R}}\}$ (Definition 3.7). Thus $(\mathcal{P}(\mathbb{W}), \subseteq)$ is certainly an overabundant candidate as semantic domain, as it contains many sets which cannot be the semantics of a TRS. Thus let us restrict our attention to sets closed under rewriting consequences, i.e., to the domain $\mathbb{S} := \{I \subseteq \mathbb{W} \mid \text{unzip}(I) = I\}$ ⁴ ordered by set inclusion. $\mathbb{S}(\subseteq, \cup, \cap, \emptyset, \mathbb{W})$ is a complete lattice.

However, as already argued before, any $S \in \mathbb{S}$ contains several redundant semantic equations, which can be removed by using zip. Actually, for all semantics S of interest (i.e., $S = \mathcal{B}_{BT}(\mathcal{R})$ for some \mathcal{R} and $BT \in \{\text{eval}_{\mathcal{R}}, \text{rnf}_{\mathcal{R}}\}$), $\text{zip}(S)$ contains all the relevant information of S because, by Lemma 3.16 and Point 11 of Proposition 3.11, $\text{unzip}(\text{zip}(S)) = S$. Hence $\text{zip}(S)$ can be

⁴Note that, by idempotence of unzip, for any set $I \subseteq \mathbb{W}$, the set $\text{unzip}(I)$ is closed under rewriting consequences, i.e., $\text{unzip}(\text{unzip}(I)) = \text{unzip}(I)$. Thus \mathbb{S} is just the image of $\mathcal{P}(\mathbb{W})$ by unzip, in symbols $\mathbb{S} = \{\text{unzip}(I) \mid I \subseteq \mathbb{W}\}$.

taken as the “canonical” (compact) representative of any set that is one possible semantics of interest.

In other words, zipped sets can be considered as “smart” representations of their own closure under rewriting consequences (which is a “natural” semantics), where we filter out one (useless) “infinite dimension”. Even if this is not a breakthrough from a purely theoretical semantics point of view, it does matter for applications of this semantics (as we have argued before). Thus, we are going to use only zipped sets: the denotation of a TRS (program) will be a zipped set of semantic rules, whose unzipping allows us to recover the original natural semantics.

Example 4.1

Consider \mathcal{R}_{ID} and \mathcal{R}_{SUM} of Examples 1.3 and 2.1. The semantics

$$\text{zip}(\mathcal{B}_{\text{rnf}}^{\mathcal{V}}(\mathcal{R}_{ID})) = \{\text{id}(\mathbf{X}) \mapsto \mathbf{X}\}$$

is the canonical representative of $\mathcal{B}_{\text{rnf}}^{\mathcal{V}}(\mathcal{R}_{ID})$ and the semantics

$$\text{zip}(\mathcal{B}_{\text{rnf}}^{\mathcal{V}}(\mathcal{R}_{SUM})) = \{\mathbf{s}^n(0) + \mathbf{X} \mapsto \mathbf{s}^n(\mathbf{X}) \mid n \geq 0\}$$

is the canonical representative of $\mathcal{B}_{\text{rnf}}^{\mathcal{V}}(\mathcal{R}_{SUM})$.

Zipped sets can be ordered trivially by using the underlying set inclusion order of the represented closures. Now we are ready to define the semantic domain \mathbb{C} .

Definition 4.2 (Semantic Domain) *A rule-based interpretation \mathcal{I} is a zipped set of semantic rules.*

The semantic domain \mathbb{C} is the set of rule-based interpretations $\{\text{zip}(I) \mid I \in \mathbb{S}\}$ ordered by

$$A \sqsubseteq B := \text{unzip}(A) \subseteq \text{unzip}(B). \tag{4.1}$$

Note that $\mathbb{C} = \{\text{zip}(I) \mid I \subseteq \mathbb{W}\}$ ⁵ and, by idempotence of zip, $\forall I \in \mathbb{C}$, $\text{zip}(I) = I$. Thus \mathbb{C} is the set of zipped sets in \mathbb{W} .

The proof that \sqsubseteq is an order is straightforward. Moreover note that, by monotonicity of unzip, \sqsubseteq is implied by \subseteq , i.e., for all $A, A' \in \mathbb{C}$, if $A \subseteq A'$ then $A \sqsubseteq A'$.

⁵Given $I \subseteq \mathbb{W}$, consider $I' = \text{zip}(I)$ and $U := \text{unzip}(I)$. By Points 5 and 1 of Proposition 3.11, $\text{zip}(U) = I'$. Thus $I' \in \mathbb{C}$. The vice versa is straightforward, as $\mathbb{S} \subseteq \mathcal{P}(\mathbb{W})$.

4.2 Denotational (Fixpoint) Semantics

We can give a fixpoint characterization of our semantics by means of the following narrowing-based, immediate consequence operator.

Definition 4.3 (Immediate Consequence Operator) *Let $\mathcal{R} := (\Sigma, R)$ be a TRS, $BT \in \{\text{eval}_{\mathcal{R}}, \text{rnf}_{\mathcal{R}}\}$, and $\mathcal{I} \in \mathbb{C}$. The immediate consequence operator is defined as:*

$$T_{BT, \mathcal{R}}(\mathcal{I}) := \text{zip}(\{l\theta \mapsto u \mid l \rightarrow r \in R, r \rightsquigarrow_{\theta, \mathcal{I}}^* u, u \in BT\}) \quad (4.2)$$

Let us explain the meaning of this definition. Equation (4.2) “unfolds” (by using narrowing) the RHS r of a rule $l \rightarrow r$ with the interpretation \mathcal{I} and then zip takes care of removing inessential new contributes⁶. This plays several roles.

1. For the case when r is unarrowable, it provides the initial blocks for constructing the semantics. For instance, for $BT = \text{rnf}$ and the TRS \mathcal{R}_{ID} of Example 1.3, we obtain the semantic rule $\text{id}(\mathbf{X}) \mapsto \mathbf{X}$ as an initial semantic block. Note that the rule $\text{id}(\mathbf{0}) \mapsto \mathbf{0}$ is also obtained as an initial semantic block but is dropped by the zip operation.
2. It allows us to obtain semantic rules from program rules containing nested calls in their RHSs. Following the example, we obtain the semantic rule $\text{id}(\mathbf{s}(\mathbf{X})) \rightarrow \mathbf{s}(\mathbf{X})$ by unfolding the program rule $\text{id}(\mathbf{s}(\mathbf{X})) \rightarrow \mathbf{s}(\text{id}(\mathbf{X}))$ w.r.t. $\text{id}(\mathbf{X}) \mapsto \mathbf{X}$. Then, incidentally, this rule is also removed by the zip operation.
3. It also speeds up the process of generating consequences, as we (potentially) use all previously computed semantic rules collected in \mathcal{I} to “unfold” the right-hand sides of the rules.⁷

It is worth noting that we perform narrowing w.r.t. \mathcal{I} , but the test for membership in BT is done with \mathcal{R} instead of \mathcal{I} . This is very important, since we do not want to include useless temporary semantic rules while producing the semantics. This is also important in another sense: we do not need to care about the termination and completeness of narrowing for \mathcal{R} but for \mathcal{I} instead. The advantage is that the rules of \mathcal{I} have the very beneficial shape of a rigid normal form in their right-hand sides, which allows us

⁶Note that, as we will prove in the following, the set of unfoldings produced by narrowing is always finite. Thus the zip operator can be effectively implemented by just checking each equation against all the others.

⁷This is *particularly* relevant for 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.

to apply the results in [3] to guarantee that narrowing terminates and is reachability-complete w.r.t. \mathcal{I} . This essentially requires the following two conditions: right-rnf and left-plain TRSs.

Definition 4.4 (Right-rnf TRS [3]) *A TRS is called right-rnf if the right-hand side of every rule in \mathcal{R} is a rnf.*

Example 4.5

The TRS $\mathcal{R} = \{\text{pk}(K, \text{sk}(K, X)) \rightarrow X, \text{sk}(K, \text{pk}(K, X)) \rightarrow X\}$, which contains the protocol cancellation rules for public encryption/decryption, is trivially right-rnf; the symbol pk is used for public key encryption and the symbol sk for private key encryption.

Definition 4.6 (left-plain TRS [3]) *A TRS \mathcal{R} is called left-plain if every non-ground strict subterm of the left-hand side of every rule of \mathcal{R} is a rigid normal form.*

Roughly speaking, left-plain TRSs [3] are a generalization of the left-flat TRSs of [9] (i.e., each argument of the left-hand side of a rewrite rule is either a variable or a ground term).

Example 4.7

The TRS $\mathcal{R} = \{X + X \rightarrow 0, X + 0 \rightarrow X, (0 + 0) + h(X) \rightarrow h(X)\}$, defining a specialized version of the xor operator used in many security protocols [13, 14], is left-plain. The symbol h is constructor; it might represent e.g. the hash of a message.

Example 4.8

The semantics given in Examples 1.3 and 3.1, seen as a TRS, are left-plain. The rule $\text{pk}(K, \text{sk}(K, X)) \mapsto X$ of Example 4.5 is not left-plain, since the non-ground subterm $\text{sk}(K, X)$ is not a rnf.

Corollary 4.9 (Termination of narrowing [3]) *Let \mathcal{I} be a right-rnf TRS which is either*

1. *right-linear;*
2. *confluent and left-plain; or*
3. *topmost.*

Then, every narrowing derivation issuing from any term in \mathcal{I} terminates. In the case of Point 1, the termination only holds for linear input terms.

Example 4.10

Consider the following TRS $\mathcal{R}_{INS} := \{\text{insertNC}(X, \text{cons}(Y, Z)) \rightarrow \text{cons}(Y, \text{insertNC}(X, Z)), \text{insertNC}(X, Z) \rightarrow \text{cons}(X, Z)\}$. Note that \mathcal{R}_{INS} is right-linear and non-confluent. The zipped semantics is

$$\begin{aligned} \text{zip}(\mathcal{B}_{\text{eval}}^{\mathcal{V}}(\mathcal{R}_{INS})) = \\ \{ & \text{insertNC}(X, Z) \mapsto \text{cons}(X, Z), \\ & \text{insertNC}(X, \text{cons}(Y, Z)) \mapsto \text{cons}(Y, \text{cons}(X, Z)), \\ & \text{insertNC}(X, \text{cons}(Y1, \text{cons}(Y2, Z))) \mapsto \text{cons}(Y1, \text{cons}(Y2, \text{cons}(X, Z))), \\ & \dots \} \end{aligned}$$

By Corollary 4.9, every narrowing derivation in any finite subset of $\text{zip}(\mathcal{B}_{\text{eval}}^{\mathcal{V}}(\mathcal{R}_{INS}))$ terminates. Note that narrowing does not terminate in \mathcal{R}_{INS} , but we do not require this property for our results.

4.2.1 Denotation-compact TRSs

Termination of narrowing in a particular class of rule-based interpretations, together with the completeness of narrowing w.r.t. rewriting in such a class, are essential for ensuring that the $T_{BT, \mathcal{R}}$ transformation is effectively computable. The following definition formalizes these requirements. Note that the right-hand sides of the semantic rules that are obtained by the successive applications of the immediate consequence operator are rigid normal forms (or values), which implies that these rules are terminating (w.r.t. rewriting). Hence, the zip operator is computable, since only finite sets of terminating rules are generated.

Definition 4.11 (Narrowing-wise Interpretation) *Given a rule set $\mathcal{J} \in \mathbb{C}$, we say that \mathcal{J} is narrowing-wise if the following two conditions hold:*

1. *(strong reachability-completeness of narrowing) Narrowing is strongly reachability-complete for \mathcal{J} .*
2. *(narrowing termination) There are no infinite narrowing sequences in \mathcal{J} issued from any term.*

Definition 4.12 (Denotation-compact TRS) *A TRS \mathcal{R} is called denotation-compact if $T_{BT, \mathcal{R}}(\mathcal{I})$ is narrowing-wise, for any narrowing-wise interpretation $\mathcal{I} \in \mathbb{C}$.*

The following result is the basis for a useful characterization of denotation-compact TRSs.

Lemma 4.13 *Let \mathcal{R} be a TRS, and $BT \in \{\text{eval}, \text{rnf}\}$. Let $\mathcal{I} \in \mathbb{C}$ s.t. $\mathcal{I} \subseteq \mathcal{B}_{BT}^{\mathcal{V}}(\mathcal{R})$. If \mathcal{I} and \mathcal{R} are both either*

1. *topmost*,
2. *confluent*,
3. *right-linear*, or
4. *left-plain and left-linear*;

then $T_{BT,\mathcal{R}}(\mathcal{I})$ is, respectively,

1. *topmost*,
2. *confluent*,
3. *right-linear*, or
4. *left-plain and left-linear*.

Proof. For topmost TRSs, the proof is trivial, since any narrowing step is performed at the top position of every term.

For confluent TRSs, we prove the claim by contradiction. Assume that \mathcal{I} and \mathcal{R} are confluent and $T_{BT,\mathcal{R}}(\mathcal{I})$ is not confluent. Given that the RHS of every rule in $T_{BT,\mathcal{R}}(\mathcal{I})$ is a rnf, there must be two equations $t \mapsto s$ and $t \mapsto s'$ in $T_{BT,\mathcal{R}}(\mathcal{I})$ s.t. $s \neq s'$. Then, there are two rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ in \mathcal{R} and two substitutions σ_1 and σ_2 such that $t = l_1\sigma_1$, $t = l_2\sigma_2$, $r_1\sigma_1 \rightarrow_{\mathcal{I}}^* s$, and $r_2\sigma_2 \rightarrow_{\mathcal{I}}^* s'$. Therefore, we have $t \rightarrow_{\mathcal{R}} r_1\sigma_1 \rightarrow_{\mathcal{R}}^* s$, and $t \rightarrow_{\mathcal{R}} r_2\sigma_2 \rightarrow_{\mathcal{R}}^* s'$, which yields to a contradiction, since s and s' are different rnf's.

For right-linear TRSs, we just prove that each narrowing application from a linear term provides a linear term. Let us consider $t \rightsquigarrow_{\sigma,\mathcal{R}} s$ using rule $l \rightarrow r$ at position p . Since t is linear because is the RHS of a rule in \mathcal{R} , $\sigma(x) = x$ for each $x \in \text{Var}(t) - \text{Var}(t|_p)$. Then, $s = (t[r]_p)\sigma = t[r\sigma]_p$ and, since r is linear, s is also linear.

For left-plain, left-linear TRSs, every semantic rule in $T_{BT,\mathcal{R}}(\mathcal{I})$ has the form $l\theta \mapsto u$. Since l is left-plain and \mathcal{I} is left-linear, for each binding $(x \mapsto u) \in \theta$, u is either ground or a rigid normal form, and the claim follows straightforwardly. ■

Lemma 4.14 *The following classes of TRSs are denotation-compact:*

- (i) *topmost*;
- (ii) *right-linear*;
- (iii) *confluent, left-linear and left-plain*.

Proof. Cases (i) and (ii) and straightforward by Lemma 4.13, Corollary 4.9, and the strong reachability-completeness of TRSs proved in [33]. For case (iii), we must also consider that, by Definition 4.3, $T_{BT,\mathcal{R}}(\mathcal{I})$ is a right-rnf TRS for $\mathcal{I} \subseteq \mathcal{B}_{BT}^{\vee}(\mathcal{R})$ and $BT \in \{\text{eval}, \text{rnf}\}$. Hence $T_{BT,\mathcal{R}}(\mathcal{I})$ is terminating and the claim follows from the fact that confluent and terminating TRSs are trivially strong reachability-complete. ■

Lemma 4.14 implies that, for the considered TRSs, all the iterations built with $T_{BT,\mathcal{R}}$ are narrowing-wise. Note that the previous result is very

handy, since it applies to many TRSs that are commonly used in rewriting logic and functional programming:

- topmost TRSs and right-linear TRSs, which fulfill the soundness conditions for narrowing-based reachability analysis [33], and
- (almost) orthogonal TRSs (a subclass of confluent, left-linear and left-plain TRSs), which fulfill the design conditions of many functional programming languages such as Haskell.

Example 4.15

The TRSs \mathcal{R}_{ID} , \mathcal{R}_{SUM} , and \mathcal{R}_{INS} of Examples 1.3, 2.1 and 4.10 are right-linear. Thus, by Lemma 4.14, are denotation-compact.

4.2.2 Properties of the Immediate Consequence Operator

In the following, we characterize the properties of the immediate consequence operator in narrowing-wise (rule-based) interpretations and denotation-compact TRSs.

We prove continuity of the immediate consequence operator in Theorem 4.18 below. First, let us demonstrate two auxiliary results. We write $t \rightsquigarrow^! s$ to denote a narrowing computation from t to a rnf s .

Lemma 4.16 *Let $\mathcal{I}_1, \mathcal{I}_2 \in \mathbb{C}$ be narrowing-wise interpretations such that $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2$. Let $r, u \in \mathcal{T}(\Sigma, \mathcal{V})$. If $r \rightsquigarrow_{\rho, \mathcal{I}_1}^! u$, then there are substitutions η, ρ', θ and a term $u' \in \mathcal{T}(\Sigma, \mathcal{V})$ such that $r \rightsquigarrow_{\eta, \mathcal{I}_2}^* u'$, $\rho|_r \rightarrow_{\mathcal{I}_2}^* \rho'$, $\rho' = \eta\theta|_r$, and $u = u'\theta$.*

Proof. $r \rightsquigarrow_{\rho, \mathcal{I}_1}^! u$ implies $r\rho \rightarrow_{\mathcal{I}_1}^! u$ and, since $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2$, $r\rho \rightarrow_{\mathcal{I}_2}^! u$. Since \mathcal{I}_2 is a narrowing-wise interpretation, i.e., by strong-completeness of narrowing, there are substitutions η, ρ', θ and a term $u' \in \mathcal{T}(\Sigma, \mathcal{V})$ such that $r \rightsquigarrow_{\eta, \mathcal{I}_2}^* u'$, $\rho|_r \rightarrow_{\mathcal{I}_2}^* \rho'$, $\rho' = \eta\theta|_r$, and $u = u'\theta$. ■

Lemma 4.17 *Let $\mathcal{I}_1, \mathcal{I}_2 \in \mathbb{C}$ be narrowing-wise interpretations. If $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2$, then $T_{BT, \mathcal{R}}(\mathcal{I}_1) \sqsubseteq T_{BT, \mathcal{R}}(\mathcal{I}_2)$.*

Proof. Consider $t \mapsto s \in T_{BT, \mathcal{R}}(\mathcal{I}_1)$. By Lemma 4.16, there are $t' \mapsto s' \in T_{BT, \mathcal{R}}(\mathcal{I}_2)$ and a substitution ρ such that $t = t'\rho$ and $s = s'\rho$. Thus, $T_{BT, \mathcal{R}}(\mathcal{I}_2) \vdash t \mapsto s$ and the conclusion follows. ■

Theorem 4.18 *Let \mathcal{R} be a denotation-compact TRS and $BT \in \{\text{eval}, \text{rnf}\}$. The $T_{BT, \mathcal{R}}$ operator is continuous w.r.t. \sqsubseteq .*

Proof. To prove that $T_{BT, \mathcal{R}}$ is continuous we can prove that it is monotone and finitary. It is finitary because of termination of narrowing in narrowing-wise interpretations. Monotonicity follows by Lemma 4.17. ■

Finally, from the soundness of narrowing, the soundness of the $T_{BT,\mathcal{R}}$ operator follows straightforwardly.

Theorem 4.19 (Soundness) *Let \mathcal{R} be a TRS and $BT \in \{\text{eval}, \text{rnf}\}$. Let $\mathcal{I} \in \mathbb{C}$ s.t. $\mathcal{I} \subseteq \mathcal{B}_{BT}^{\mathcal{V}}(\mathcal{R})$. For each $t \mapsto s \in T_{BT,\mathcal{R}}(\mathcal{I})$, $t \rightarrow_{\mathcal{R}}^* s$.*

For interesting classes of TRSs, the fixpoint of the $T_{BT,\mathcal{R}}$ transformation characterizes the meaning in \mathcal{R} of all input calls, which allows us to define a complete compressed fixpoint semantics in the next section.

4.2.3 Fixpoint Compressed Semantics

We are ready to formalize our notion of compressed semantics for TRSs in the fixpoint style. As usual, we consider the chain of iterations of $T_{BT,\mathcal{R}}$ starting from the bottom, by defining $T_{BT,\mathcal{R}}^0 := \emptyset$; $T_{BT,\mathcal{R}}^{k+1} := T_{BT,\mathcal{R}}(T_{BT,\mathcal{R}}^k)$, for $k \geq 0$; and $T_{BT,\mathcal{R}}^{\omega} := \sqcup_{k \geq 0} T_{BT,\mathcal{R}}^k$.

Definition 4.20 (Fixpoint Compressed Semantics) *The least fixpoint compressed semantics of a program \mathcal{R} is defined as $\mathcal{F}_{BT}(\mathcal{R}) := T_{BT,\mathcal{R}}^{\omega}$.*

Example 4.21

Consider again the TRS \mathcal{R}_{ID} of Example 1.3. The transformation $T_{\text{rnf},\mathcal{R}_{ID}}$ computes the following interpretations: $T_{\text{rnf},\mathcal{R}_{ID}}^1 = T_{\text{rnf},\mathcal{R}_{ID}}^2 = \{\text{id}(X) \mapsto \mathbf{X}\}$, which is, thus, the least fixpoint.

We are able to characterize some broad classes of TRSs where the soundness and completeness of our compact fixpoint semantics can be proved. The following definition extends the notion of definedness of a TRS to (possibly) non-confluent TRSs.

Definition 4.22 (BT-defined TRS) *We say that $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is BT-defined in the TRS \mathcal{R} if there exists at least one $t' \in BT$ such that $t \rightarrow_{\mathcal{R}}^! t'$ and, for all t'' such that $t \rightarrow_{\mathcal{R}}^! t''$, $t'' \in BT$.*

We say that \mathcal{R} is BT-defined (resp. BT-ground-defined) if, for all $t \in \mathcal{T}(\Sigma, \mathcal{V})$ (resp. $t \in \mathcal{T}(\Sigma)$) t is BT-defined in \mathcal{R} .

It is immediate to see that BT-definedness is much more demanding than BT-ground-definedness. BT-(ground-)definedness has been studied in the literature for different semantics:

- For the ground value semantics $\mathcal{B}_{\text{eval}}(\mathcal{R})$, eval-ground-definedness implies that $\text{nf} \cap \mathcal{T}(\Sigma) = \text{eval} \cap \mathcal{T}(\Sigma)$. Therefore, eval-ground-definedness is equivalent to the condition that \mathcal{R} is weakly normalizing and completely defined (CD); see [5]. A TRS \mathcal{R} is *weakly normalizing* if every term has a normal form in \mathcal{R} , though infinite sequences from t may

exist. A TRS \mathcal{R} is *completely defined* if each defined symbol of the signature is completely defined. In other words, it does not occur in any ground term in normal form, i.e., function symbols are reducible on all ground terms.

- For the non-ground values semantics $\mathcal{B}_{\text{eval}}^{\mathcal{V}}(\mathcal{R})$, eval-definedness is much more demanding, since it requires functions to be reducible on all terms, not only ground terms.
- For the normalization semantics $\mathcal{B}_{\text{nf}}(\mathcal{R})$ and $\mathcal{B}_{\text{nf}}^{\mathcal{V}}(\mathcal{R})$, both nf-ground-definedness and nf-definedness are simply equivalent to the notion of weakly normalizing TRS, which is defined for terms with variables. It is the same for the semantics of ground rigid normal forms $\mathcal{B}_{\text{rnf}}(\mathcal{R})$, since $\mathcal{B}_{\text{rnf}}(\mathcal{R}) = \mathcal{B}_{\text{nf}}(\mathcal{R})$. However it is more demanding for $\mathcal{B}_{\text{rnf}}^{\mathcal{V}}(\mathcal{R})$ than weakly normalizing, since it requires every term (not only ground terms) to reach a rigid normal form by rewriting.

Example 4.23

For $BT \in \{\text{eval}, \text{rnf}, \text{nf}\}$, the TRSs \mathcal{R}_{ID} , \mathcal{R}_{SUM} , and \mathcal{R}_{INS} of Examples 1.3, 2.1 and 4.10 are BT -ground-defined. Only the TRS \mathcal{R}_{ID} is BT -defined.

Let us also define the class of BT -based TRSs, which generalizes the class of left-linear constructor systems as follows.

Definition 4.24 (BT -based TRS) *Let \mathcal{R} be a TRS and $BT \in \{\text{eval}_{\mathcal{R}}, \text{rnf}_{\mathcal{R}}\}$. A substitution σ is BT -based if, for each $X \in \mathcal{V}$, $X\sigma \in BT$. Given a TRS \mathcal{R} , we call it BT -based if every substitution computed by narrowing in \mathcal{R} is BT -based, i.e., for $t, t' \in \mathcal{T}(\Sigma, \mathcal{V})$ such that $t \rightsquigarrow_{\theta, \mathcal{R}} t'$, $\theta|_t$ is BT -based.*

Popular classes of rnf-based TRSs are:

- (i) left-linear constructor systems,
- (ii) almost orthogonal TRSs, and
- (iii) topmost TRSs.

Actually, (i) and (ii) are typical functional programs and the class of left-linear constructor systems is exactly the eval-based TRSs, a subclass of rnf-based TRSs. On the other hand, weakly normalizing left-linear constructor systems are rnf-based as well as rnf-ground-defined. It is worth noting that rnf-based TRSs are left-plain but not vice versa, as shown in the following example.

Example 4.25

For $BT \in \{\text{eval}, \text{rnf}\}$, the TRSs \mathcal{R}_{ID} , \mathcal{R}_{SUM} , and \mathcal{R}_{INS} of Examples 1.3, 2.1 and 4.10 are BT -based, since they are left-linear constructor systems. However, the TRS of Example 4.5 is not BT -based (nor left-plain). Furthermore,

the TRS of Example 4.7 is left-plain but not BT -based, since given the term $(0 + 0) + Y$, we have the narrowing step $(0 + 0) + Y \rightsquigarrow_{\sigma} 0$ using the rule $X + X \rightarrow 0$, where the substitution $\sigma = \{Y \mapsto 0 + 0\}$ is not BT -based.

Now we can prove the correctness of the fixpoint semantics w.r.t. the ordinary, big-step collecting semantics. The proof of Theorem 4.26 is in Appendix A.

Theorem 4.26 (Ground Soundness and Completeness) *Let $BT \in \{\text{rnf}, \text{eval}\}$. Let \mathcal{R} be either a*

- *BT -ground-defined, terminating, denotation-compact TRS that is either confluent or BT -based; or*
- *topmost TRS.*

Then, $\mathcal{B}_{BT}(\mathcal{R}) = \text{unzip}(\mathcal{F}_{BT}(\mathcal{R})) \cap (\mathcal{T}(\Sigma) \times \mathcal{T}(\Sigma))$.

Corollary 4.27 (Correctness w.r.t. Behavior) *Let $BT \in \{\text{rnf}, \text{eval}\}$. Let $\mathcal{R}_1, \mathcal{R}_2$ be either*

- *BT -ground-defined, terminating, denotation-compact TRSs that are either confluent or BT -based; or*
- *topmost TRSs.*

Then, $\mathcal{F}_{BT}(\mathcal{R}_1) = \mathcal{F}_{BT}(\mathcal{R}_2)$ implies $\mathcal{B}_{BT}(\mathcal{R}_1) = \mathcal{B}_{BT}(\mathcal{R}_2)$.

Note that the conditions that \mathcal{R} is BT -based (or confluent), BT -ground-defined, and terminating in the previous result are all necessary. Recall that denotation-compactness is required to ensure that narrowing terminates in the denotation.

Example 4.28

Let us consider the following denotation-compact, terminating, eval -based TRS $\mathcal{R} := \{g \rightarrow f(h(a)), h(a) \rightarrow s(h(b)), f(s(X)) \rightarrow a\}$. \mathcal{R} is not eval -ground-defined, since e.g. $h(a)$ cannot be rewritten to a value. Now we have $T_{\text{eval}, \mathcal{R}}^1 = T_{\text{eval}, \mathcal{R}}^2 = \{f(s(X)) \mapsto a\}$. Then, $g \mapsto a$ cannot be obtained from $\mathcal{F}_{\text{eval}}(\mathcal{R})$.

Example 4.29

Let us consider the following denotation-compact, terminating, rnf -ground-defined TRS $\mathcal{R} := \{g \rightarrow f(h), h \rightarrow s(i), i \rightarrow a, f(s(a)) \rightarrow a, f(s(i)) \rightarrow b\}$. \mathcal{R} is not rnf -based nor confluent due to the left-hand side $f(s(i))$. Now we have $T_{\text{rnf}, \mathcal{R}}^1 = \{i \mapsto a, f(s(a)) \mapsto a, f(s(i)) \mapsto b\}$, $T_{\text{rnf}, \mathcal{R}}^2 = T_{\text{rnf}, \mathcal{R}}^1 \cup \{h \mapsto s(a)\}$, $T_{\text{rnf}, \mathcal{R}}^3 = T_{\text{rnf}, \mathcal{R}}^2 \cup \{g \mapsto a\}$, and $T_{\text{rnf}, \mathcal{R}}^4 = T_{\text{rnf}, \mathcal{R}}^3$. Then, $g \mapsto b$ cannot be obtained from $\mathcal{F}_{\text{rnf}}(\mathcal{R})$, whereas g rewrites to b in \mathcal{R} .

Example 4.30

Let us consider the following denotation-compact, eval-ground-defined, eval-based TRS $\mathcal{R} := \{\mathbf{g} \rightarrow \mathbf{f}(\mathbf{h}), \mathbf{h} \rightarrow \mathbf{s}(\mathbf{g}), \mathbf{f}(\mathbf{s}(\mathbf{X})) \rightarrow \mathbf{a}\}$. \mathcal{R} is not terminating, since $\mathbf{g} \rightarrow \mathbf{f}(\mathbf{h}) \rightarrow \mathbf{f}(\mathbf{s}(\mathbf{g})) \rightarrow \dots$. Since \mathbf{h} and \mathbf{g} depend on each other, we have $\mathcal{F}_{\text{eval}}(\mathcal{R}) = \{\mathbf{f}(\mathbf{s}(\mathbf{X})) \mapsto \mathbf{a}\}$, and thus the computations $\mathbf{g} \mapsto \mathbf{a}$ and $\mathbf{h} \mapsto \mathbf{s}(\mathbf{a})$ cannot be obtained from $\mathcal{F}_{\text{eval}}(\mathcal{R})$.

The criteria given in Theorem 4.26 and Corollary 4.27 are reasonable⁸ for many programming languages such as Maude, Haskell, Scheme, etc. For instance, programs in Haskell are defined as left-linear constructor systems that can be understood as confluent and terminating systems (i.e., the Haskell evaluation strategy always provides, for each input term, one and only one finite rewriting sequence to a normal form). In Maude, functional (or equational) programs are usually described as (almost) orthogonal, terminating TRSs, which is a subclass of confluent, terminating, left-linear, left-plain TRSs; see [10]. Note that much work has been done recently to prove termination of functional programs automatically in Maude and Haskell; see [17, 21]. On the other hand, many concurrent systems of interest, including the vast majority of distributed algorithms and the operational semantics of many programming languages (Java, JVM bytecode, C, Haskell, Prolog, etc.), admit fairly natural (order-sorted) topmost specifications in Maude; see [32, 33].

Thus we believe that our compression methodology and the results that we have proved are quite powerful and practical. In Section 5 we also show more evidence to support this claim by showing, on various benchmarks, that the fixpoint computation of our semantics produces dramatically fewer semantic rules at each step w.r.t. the big-step semantics.

4.3 Operational (Compressed) Semantics

In this section, we study an operational semantics and compare it with the denotational semantics provided in Section 3.1.

Definition 4.31 *Let \mathcal{R} be a TRS and $BT \in \{\text{rnf}, \text{eval}\}$. The Operational Denotation of \mathcal{R} is defined as:*

$$\mathcal{O}_{BT}(\mathcal{R}) := \text{zip}(\{f(x_1, \dots, x_n)\theta \mapsto t \mid f(x_1, \dots, x_n) \rightsquigarrow_{\theta, \mathcal{R}}^* t, t \in BT\})$$

where x_1, \dots, x_n are pairwise distinct variables.

It is easy to prove that this operational semantics captures the observables of rigid normal forms and values.

⁸Obviously, these classes of programs can only be considered as the basis of functional programming languages, since many features are not addressed in this paper, such as evaluation strategies, strategy annotations, type systems; or algebraic properties such as associativity and commutativity, etc.

Corollary 4.32 *Let \mathcal{R} be a strongly reachability-complete TRS and $BT \in \{\text{rnf}, \text{eval}\}$. Then, $\text{unzip}(\mathcal{O}_{BT}(\mathcal{R})) = \mathcal{B}_{BT}^{\vee}(\mathcal{R})$.*

Corollary 4.33 (Correctness and Full Abstraction) *Let \mathcal{R} be a strongly reachability-complete TRS and $BT \in \{\text{rnf}, \text{eval}\}$. Then, $\mathcal{B}_{BT}^{\vee}(\mathcal{R}_1) = \mathcal{B}_{BT}^{\vee}(\mathcal{R}_2)$ if and only if $\mathcal{O}_{BT}(\mathcal{R}_1) = \mathcal{O}_{BT}(\mathcal{R}_2)$.*

The optimality of the operational semantics is also straightforward.

Corollary 4.34 (Optimality) *Let \mathcal{R} be a strongly reachability-complete TRS and $BT \in \{\text{rnf}, \text{eval}\}$. For each $S \subseteq \mathcal{B}_{BT}^{\vee}(\mathcal{R})$ s.t. $\text{unzip}(S) = \mathcal{B}_{BT}^{\vee}(\mathcal{R})$, then $\text{zip}(S) = \mathcal{O}_{BT}(\mathcal{R})$.*

The operational semantics in Definition 4.31 is actually equivalent to the fixpoint version, for BT -defined TRSs. The proof of Theorem 4.35 is in Appendix A.

Theorem 4.35 (Equivalence with Denotational Semantics) *Let $BT \in \{\text{rnf}, \text{eval}\}$. Let \mathcal{R} be either a*

- *BT -defined, terminating, denotation-compact TRS that is either confluent or BT -based; or*
- *topmost TRS.*

Then, $\mathcal{F}_{BT}(\mathcal{R}) = \mathcal{O}_{BT}(\mathcal{R})$.

Here we would like to justify why we have given a bottom-up formalization for our compressed semantics instead of a simpler top-down operational semantics, such as the one in Definition 4.31:

- First, the computation of $\mathcal{O}_{BT}(\mathcal{R})$ requires that narrowing terminates *in \mathcal{R}* , which is more demanding than narrowing termination *in the interpretations* obtained from \mathcal{R} .
- Second, the set of narrowing sequences in \mathcal{R} is not generally finite, which implies that the zip of this set is not effectively computable.
- Third, but not least important, our motivation for the compressed semantics comes from a previous work of ours [1] which aimed to extend to TRSs the technique of Abstract Diagnosis [12], originally developed for Logic Programs, to check the correctness of a TRS. This technique is *inherently* based on the use of an immediate consequence operator and [12] demonstrated that the resulting methodology is superior than top-down approaches.

Example 4.36

Consider again the TRS \mathcal{R}_{ID} of Example 1.3, which satisfies the conditions of Theorem 4.35. We have that $\mathcal{F}_{\text{eval}}(\mathcal{R}_{ID}) = \mathcal{O}_{\text{eval}}(\mathcal{R}_{ID}) = \{\text{id}(X) \mapsto X\}$.

However, note that the term $\text{id}(\mathbf{X})$ has an infinite number of narrowing derivations, which are only successively collapsed by the zip operation. Thus the computation of the semantics $\mathcal{O}_{\text{eval}}(\mathcal{R}_{ID})$ cannot be effectively implemented. On the contrary, the computation of $\mathcal{F}_{\text{eval}}(\mathcal{R}_{ID})$ is finite, as shown in Example 4.21.

4.4 Relations with the semantics of [1]

Finally, let us characterize the relationship of our $T_{BT, \mathcal{R}}$ transformation with the more traditional immediate consequence operator given in [1]. Given a set of final/blocking terms BT , in [1] we defined the following naïve immediate consequence operator.

Definition 4.37 ([1]) *Let \mathcal{R} be a TRS, $\mathcal{I} \in \mathbb{S}$, and BT a set of final/blocking state pairs. Then,*

$$T_{BT, \mathcal{R}}^{\text{unzip}}(\mathcal{I}) := BT^2 \cup \{s \mapsto t \in \mathbb{W} \mid r \mapsto t \in \mathcal{I}, s \rightarrow_{\mathcal{R}} r\}$$

where $BT^2 := \{t \mapsto t \mid t \in BT\}$.

This definition is sensible from a model-theoretic point of view, but it clearly lacks conciseness properties that are critical for analysis and debugging. Thus, one could think of using *zipped* sets as follows:

Definition 4.38 *Let \mathcal{R} be a TRS, $\mathcal{I} \in \mathbb{C}$, and BT a set of final/blocking state pairs. Then,*

$$T_{BT, \mathcal{R}}^{\text{zip}}(\mathcal{I}) := \text{zip}(BT^2 \cup \{s \mapsto t \in \mathbb{W} \mid r \mapsto t \in \text{unzip}(\mathcal{I}), s \rightarrow_{\mathcal{R}} r\})$$

Recall that, in Observation 3.12, we noted that $(\text{zip}, \text{unzip})$ is an isomorphism between compactable unzipped sets and \mathbb{C} . Then, it is clear that the immediate consequence operator of Definition 4.38 is the operator that corresponds to the one of Definition 4.37 by this isomorphism, as formally stated in the following lemma.

Lemma 4.39 *Let \mathcal{R} be a TRS and $BT \in \{\text{rnf}, \text{eval}\}$. Then, $T_{BT, \mathcal{R}}^{\text{zip}} = \text{zip} \circ T_{BT, \mathcal{R}}^{\text{unzip}} \circ \text{unzip}$.*

Now, we are able to prove the following result.

Corollary 4.40 *Let \mathcal{R} be a strongly reachability-complete TRS and $BT \in \{\text{rnf}, \text{eval}\}$. Then, $\mathcal{F}_{BT}(\mathcal{R}) = \mathcal{O}_{BT}(\mathcal{R}) = (T_{BT, \mathcal{R}}^{\text{zip}})^{\omega}$.*

Actually, the operator $T_{BT, \mathcal{R}}^{\text{zip}}$ of [1] has a much closer relationship to operational semantics than to denotational semantics. Indeed, at each application, it simulates one transition of the rewriting system. In other terms,

when applied n times to the bottom element, it will produce (equations corresponding to) derivations of length *exactly* n . From a denotational point of view, this is undesirable, since we would like to take advantage of the fact that, in the interpretation, we already have the information regarding several derivations (of different length) of various terms all at once. That is why we have preferred a “true denotational” definition such as Definition 4.3, where, at each iteration, we can produce all rewritings that are possible using “one step” from the program and all reductions that we already know from the current interpretation.

5 Experimental Results

A proof-of-concept implementation of the compression technique proposed in this paper has been developed, and used to conduct a number of experiments that demonstrate the practicality of our approach. The prototype is written in Haskell using the GHC compiler version 6.8.2, and is publicly available⁹. The tool accepts TRSs that can be written either in TPDB format¹⁰, TTT form¹¹, or a subset of the syntax of Maude functional modules.

Since there are many factors that may impact performance and effectiveness when comparing two different implementations, in order to guarantee a fair comparison w.r.t. [1], we have developed a unique fixpoint infrastructure that is parametric on the immediate consequence operator, and we have evaluated both operators (the compact one proposed in Section 4, and the one in [1]) within this single framework. In particular, both implementations share the same underlying machinery (unification, narrowing, etc). Furthermore, in order to obtain finite approximations of (possibly) infinite semantics, we use the depth- k abstraction described in [1], which essentially cuts the terms in both sides of each semantic rule down to a maximum depth bound, which is fixed by the k parameter; e.g. for depth bound $k = 3$, the term `plus(0, s(s(s(0))))` is cut down to `plus(0, s(s(X)))`.

In order to assess the practicality of our approach, we have benchmarked both the size of the (abstract) fixpoint semantics for a given depth bound and the corresponding computation times. We consider a set of benchmark programs that satisfy the conditions for the correctness and completeness of our approach. The benchmarks used for the comparison are: `risers` and `tails`, which are two almost-orthogonal and terminating Haskell programs that are commonly used for analyzing the safety of pattern matching in Haskell [35]; `doubleisone`, an almost-orthogonal and terminating TRS for doubling and unity checking borrowed from [24]; `bertconc`, a right-linear, canonical TRS for concatenating lists borrowed from [7]; `vending`, a topmost

⁹At <http://safe-tools.dsic.upv.es/zipit>

¹⁰See <http://www.lri.fr/~marche/tpdb/format.html>

¹¹See <http://colo6-c703.uibk.ac.at/ttt/trs.html>

TRS & k	1	2	3	4	5	10	15	20	30
id	4/3	11/4	27/5	63/6	143/7	-	-	-	-
incplus2	3/4	9/6	27/8	79/10	229/12	-	-	-	-
pksk	7/3	13/5	395/9	-	-	-	-	-	-
plusH	4/3	27/5	713/8	-	-	-	-	-	-
insertNC	2/3	10/5	260/9	-	-	-	-	-	-
addouble	8/4	55/7	2316/12	-	-	-	-	-	-
tails	25/5	-	-	-	-	-	-	-	-
doubleisone	27/4	635/7	-	-	-	-	-	-	-
bertconc	8/4	615/10	-	-	-	-	-	-	-
risers	22/4	-	-	-	-	-	-	-	-
insertsort	39/5	-	-	-	-	-	-	-	-
vending	70/9	70/9	70/9	70/9	70/9	70/9	70/9	70/9	70/9

Table 1: Experimental results for the semantics of [1]

TRS adapted from [10]; and `insertsort`, a standard encoding of insertion sort borrowed from the Termination Problem DataBase¹². The source code of these benchmark programs is available at the prototype web site. Finally, programs `incplus2`, `id`, `pksk`, `plusH`, and `insertNC` correspond to the TRSs given in Examples 1.1, 1.3, 4.5, 4.7 and 4.10, respectively.

Tables 1, 2 and 3 summarize our experiments. For a given depth bound, each cell in Tables 1 and 2 contains a pair n/i , where n is the number of rules in the (abstract) fixpoint semantics and i is the number of iterations needed to reach the least fixpoint. The symbol “-” indicates that the tool was not able to compute the abstract fixpoint semantics because the timeout¹³ was exceeded, or the tool exceeded the available memory. Both these circumstances imply the generation of a huge and unmanageable number of rules. In Table 3, we compare the time necessary to compute our compressed semantics versus the fixpoint semantics of [1], for each benchmark program and a given depth bound. Times are expressed in 1/100 of seconds and are the average of 10 executions. The experiments were performed on a Linux machine with an Intel Core Duo and 6 Gigabytes of memory, running Ubuntu server 8.04.

Let us analyze our results. When we compare the results of Table 2 with the results of Table 1, we confirm that our compression technique computes fewer elements than the fixpoint semantics of [1] in all programs except for `vending`, whose big-step semantics contains no redundant rules. Indeed, the impact of compressing the semantics is impressive. For instance, for program `addDouble` with depth $k = 3$, our compression technique generated only 7

¹²Available at <http://www.lri.fr/~marche/tpdb/>

¹³We have considered the same, sufficiently large timeout for all benchmarks.

TRS & k	1	2	3	4	5	10	15	20	30
id	2/3	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2
incplus2	2/3	2/3	2/3	2/3	2/3	2/3	2/3	2/3	2/3
pksk	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
plusH	3/2	3/2	3/2	3/2	3/2	3/2	3/2	3/2	3/2
insertNC	1/2	2/3	3/4	4/5	5/6	10/11	15/16	20/21	30/31
adddouble	4/4	5/4	7/5	9/6	11/7	21/12	31/17	41/22	61/32
tails	9/7	9/7	12/8	15/9	18/10	33/15	48/20	63/25	-
doubleisone	10/4	12/5	14/6	17/7	20/8	35/13	50/18	65/23	95/33
bertconc	4/4	8/5	14/6	22/7	32/8	112/13	-	-	-
risers	8/3	13/4	21/6	102/10	-	-	-	-	-
insertsort	8/4	14/5	34/7	116/9	-	-	-	-	-
vending	70/8	70/8	70/8	70/8	70/8	70/8	70/8	70/8	70/8

Table 2: Experimental results for the compressed semantics

semantic rules, as opposed to the 2316 semantic rules generated by the big-step fixpoint operator, which gives a reduction of 99.7%. We note that the original fixpoint infrastructure implemented in the DEBUSSY debugger of [1] was generally unable to generate approximations of the fixpoint semantics for a depth bound greater than two. When we compare in Table 3 the computation times of our compressed semantics versus the fixpoint semantics of [1], we also confirm that our compressed semantics can be computed much faster than the previous one. For instance, for $k = 2$, the fixpoint semantics of [1] cannot be computed for some benchmark programs, whereas the compressed one can be computed in less than 230 milliseconds. Note that, in the case of program `vending`, the big-step semantics contains no redundant rules and the compression technique only introduces a negligible overhead.

6 Conclusions

In the natural big-step rewriting semantics, there are many “semantically useless” elements that can be retrieved from a smaller set of terms. This becomes a serious issue when this semantics is used as the basis for an automated tool because the algorithms of the tool have to use and produce all this redundant information at each stage. In the best case, this reduces performance and, in the worst case, it ends in ineffective methods. We have presented a compact fixpoint semantics that models the observables of ground values (or rigid normal forms) of Term Rewriting Systems. It is both correct and complete w.r.t. the operational big-step collecting semantics of rewritings for the particular classes of TRSs as given by Theorem 4.26 but, unlike the big-step semantics, our semantics is goal-independent and collects

TRS & k	1		2		3		4		5	
	Zipped	[1]	Zipped	[1]	Zipped	[1]	Zipped	[1]	Zipped	[1]
id	0	0	0	0	0	3	0	15	0	83
incplus2	0	0	0	0	0	3	0	22	0	187
pksk	0	0	0	2	0	2913	0	-	0	-
plusH	0	0	0	9	0	7518	0	-	0	-
insertNC	0	0	0	1	0	689	1	-	1	-
adddouble	0	0	0	18	1	-	1	-	2	-
tail	3	5	3	-	6	-	11	-	24	-
doubleisone	1	4	1	2966	6	-	4	-	6	-
bertconc	0	0	3	6036	10	-	29	-	74	-
risers	2	3	6	-	77	-	8761	-	-	-
insertsort	2	16	23	-	295	-	9168	-	-	-
vending	23	17	23	17	23	17	23	17	23	17

Table 3: Computation times of the compressed semantics versus [1]

just the operational behavior of the minimal set of terms that are strictly necessary. This information is computed by narrowing, which provides only the most general rewriting sequences, and which is sufficient to describe, by semantic closure, the operational behavior of all other terms.

We presented the experimental results obtained by a proof-of-concept implementation to show that our semantics produces dramatically fewer semantic rules at each step w.r.t. the big-step semantics.

Encouraged by these benchmarks, as future work, by mixing the methodology of [1] with the idea of compressing semantics given in this paper, we plan to develop efficient, bottom-up analyzers of Maude and Haskell programs as well as the rewriting-based components of Curry and TOY, which are non-deterministic in contrast to Haskell. While top-down analyses may be more efficient for large, goal-oriented problems such as the analysis of call patterns [6], the bottom-up approach is suitable for goal-independent properties as addressed in Abstract Diagnosis [11, 12].

We would like to note that future improvements in the results for narrowing termination and strong reachability completeness could eventually lead to better conditions for our compressed semantics.

Acknowledgments

We would like to thank the anonymous referees for their very careful reviews and helpful suggestions that greatly improved the results in the paper.

References

- [1] 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.
- [2] M. Alpuente, F. J. Correa, and M. Falaschi. A Declarative Debugging Scheme for Functional Logic Programs. In M. Hanus, editor, *Proceedings of 10th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, volume 64 of *Electronic Notes in Theoretical Computer Science*, North Holland, 2002. Elsevier Science Publishers.
- [3] M. Alpuente, S. Escobar, and J. Iborra. Termination of narrowing revisited. *Theoretical Computer Science*, 410(46):4608–4625, 2009.
- [4] M. Alpuente, M. Falaschi, M. J. Ramis, and G. Vidal. A Compositional Semantics for Conditional Term Rewriting Systems. In H. E. Bal, editor, *Proc. of 6th IEEE Int’l Conf. on Computer Languages, ICCL’94*, pages 171–182, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [5] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK, 1998.
- [6] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1993.
- [7] D. Bert, R. Echahed, and B. M. Østvold. Abstract Rewriting. In *Proceedings of Third Int’l Workshop on Static Analysis (WSA’93)*, volume 724 of *Lecture Notes in Computer Science*, pages 178–192, Berlin, 1993. Springer-Verlag.
- [8] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19–20:149–197, 1994.
- [9] J. Christian. Some Termination Criteria for Narrowing and E-Narrowing. In *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 582–588. Springer-Verlag, Berlin, 1992.

- [10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2007.
- [11] 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.
- [12] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract Diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.
- [13] H. Comon-Lundh. Intruder Theories (Ongoing Work). In *Proceedings FoSSaCS 2004*, volume 2987 of *Lecture Notes in Computer Science*, pages 1–4. Springer-Verlag, 2004.
- [14] V. Cortier, S. Delaune, and P. Lafourcade. A Survey of Algebraic Properties used in Cryptographic Protocols. *Journal of Computer Security*, 14(1):1–43, 2006.
- [15] 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.
- [16] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 6, pages 244–320. Elsevier, 1990.
- [17] F. Durán, S. Lucas, and J. Meseguer. MTT: The Maude Termination Tool (System Description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 313–319. Springer-Verlag, 2008.
- [18] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A new Declarative Semantics for Logic Languages. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of Fifth Int'l Conf. on Logic Programming*, pages 993–1005, Cambridge, Mass., 1988. The MIT Press.

- [19] 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.
- [20] M. Fay. First-Order Unification in an Equational Theory. In *Fourth Int'l Conf. on Automated Deduction*, pages 161–167, 1979.
- [21] 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.
- [22] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
- [23] M. Hanus. Multi-paradigm Declarative Languages. In V. Dahl and I. Niemelä, editors, *Logic Programming, 23rd International Conference, ICLP 2007, Proceedings*, volume 4670 of *Lecture Notes in Computer Science*, pages 45–75. Springer-Verlag, 2007.
- [24] M. Hanus. Call Pattern Analysis for Functional Logic Programs. In *Proceedings 10th Int'l ACM SIGPLAN Conf. on Principle and Practice of Declarative Programming (PPDP'08)*, pages 67–78. ACM Press, 2008.
- [25] Haskell Debugging Technologies. At <http://www.haskell.org/debugging/>, October 2008.
- [26] J.-M. Hullot. Canonical Forms and Unification. In *Proceedings of the 5th International Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334, Berlin, 1980. Springer-Verlag.
- [27] J.-M. Hullot. Canonical Forms and Unification. In *Proceedings of the 5th International Conference on Automated Deduction* [26], pages 318–334.
- [28] 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.
- [29] D. E. Knuth and P. B. Bendix. Simple Word Problems in Universal Algebras. In *Computational Problems in Abstract Algebra*, pages 263–297, 1970.

- [30] F.J. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of 10th International Conference on Rewriting Techniques and Applications, RTA 1999*, volume 1631 of *Lecture Notes in Computer Science*, pages 244–247. Springer-Verlag, 1999.
- [31] S. Marlow, J. Iborra, B. Pope, and A. Gill. A Lightweight Interactive Debugger for Haskell. In G. Keller, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2007)*, pages 13–24. ACM Press, 2007.
- [32] J. Meseguer and G. Rosu. The Rewriting Logic Semantics Project. *Theoretical Computer Science*, 373(3):213–237, 2007.
- [33] J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1-2):123–160, 2007.
- [34] A. Middeldorp and E. Hamoen. Completeness Results for Basic Narrowing. *Journal of Applicable Algebra in Engineering, Communication and Computing*, 5:313–353, 1994.
- [35] N. Mitchell and C. Runciman. A Static Checker for Safe Pattern Matching in Haskell. In M. Van Eekelen, editor, *Trends in Functional Programming*, volume 6, pages 15–30. Intellect, 2007.
- [36] 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.
- [37] S. Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.
- [38] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, Reading, MA, 1993.
- [39] TeReSe, editor. *Term Rewriting Systems*. Cambridge University Press, Cambridge, UK, 2003.
- [40] P. Wadler. An angry half-dozen. *SIGPLAN Notices*, 33(2):25–30, 1998.

A Proof of main theorems

The following result is the basis for the main theorems below. We use notation $t \xrightarrow{n}_{\mathcal{R}} s$ to represent a rewrite sequence from t to s consisting of n steps.

Lemma A.1 *Let $BT \in \{\text{rnf}, \text{eval}\}$. Let \mathcal{R} be a BT -defined, terminating, denotation-compact TRS that is either confluent or BT -based. Let $t, s \in \mathcal{T}(\Sigma, \mathcal{V})$. If $t \mapsto s \in \mathcal{B}_{BT}^{\mathcal{V}}(\mathcal{R})$, then $\mathcal{F}_{BT}(\mathcal{R}) \vdash t \mapsto s$.*

Proof. First, consider the maximal context $C[]$ with m holes at disjoint positions p_1, \dots, p_m such that $t = C[u_1, \dots, u_m]$, $s = C[v_1, \dots, v_m]$, and $v_1, \dots, v_m \in BT$. Since $C[]$ is maximal, each sequence $u_j \xrightarrow{*}_{\mathcal{R}} v_j$ contains at least one rewriting step at the top position. In the following, we prove that $\mathcal{F}_{BT}(\mathcal{R}) \vdash u_j \mapsto v_j$ for each $j = 1, \dots, m$, and thus the conclusion follows.

Let $u, v \in \mathcal{T}(\Sigma, \mathcal{V})$ and $v \in BT$ such that $u \xrightarrow{*}_{\mathcal{R}} v$ contains at least one rewriting step at the top position. Since \mathcal{R} is terminating, we consider the longest rewrite derivation from u to v , i.e., $u = t_0 \xrightarrow{p_1}_{\mathcal{R}} t_1 \dots \xrightarrow{p_n}_{\mathcal{R}} t_n = v$. Let $k \in \{1, \dots, n\}$ be such that p_k is the first top position in the sequence p_1, \dots, p_n . The derivation can be split into three parts $t_0 \xrightarrow{n_1}_{\mathcal{R}} t_{k-1} \xrightarrow{\epsilon}_{\mathcal{R}} t_k \xrightarrow{n_2}_{\mathcal{R}} t_n$. Then, we prove $\mathcal{F}_{BT}(\mathcal{R}) \vdash t_0 \mapsto t_n$ by induction on $n = n_1 + n_2 + 1$.

$n_1 = 0$ & $n_2 = 0$) Here we have that $t_0 \xrightarrow{\epsilon}_{l \rightarrow r} t_n$ where $l \rightarrow r \in \mathcal{R}$ and ρ is the applied substitution, i.e., $t_0 = l\rho$ and $t_n = r\rho$. Note that $r \in BT$, since $r\rho \in BT$. Thus, by definition, $l \mapsto r$ has been eventually added to the fixpoint (though possibly removed after the zip compression); hence, $\mathcal{F}_{BT}(\mathcal{R}) \vdash l \mapsto r$, which implies $\mathcal{F}_{BT}(\mathcal{R}) \vdash t_0 \mapsto t_n$.

$n_1 > 0$ or $n_2 > 0$) Here we have $t_0 \xrightarrow{n_1}_{\mathcal{R}} t_{k-1} \xrightarrow{\epsilon}_{l \rightarrow r} t_k \xrightarrow{n_2}_{\mathcal{R}} t_n$ where $l \rightarrow r \in \mathcal{R}$ and $t_n \in BT$. Following the same reasoning as in the general case given at the beginning of this proof, there is a maximal context $C[]$ with m holes at disjoint positions q_1, \dots, q_m such that $t_0 = C[u_1, \dots, u_m]$ and $t_{k-1} = C[u'_1, \dots, u'_m]$. Now, since p_k is the first reduced top position, $C[]$ is different from the empty context.

Note that u'_1, \dots, u'_m do not necessarily belong to BT . Let $\rho = \{x_1 \mapsto w_1, \dots, x_{n_\rho} \mapsto w_{n_\rho}\}$ be such that $t_{k-1} = l\rho$ and $t_k = r\rho$. Assume ρ is not BT -based. Since \mathcal{R} is BT -defined, there is a BT -based substitution $\hat{\rho} = \{x_1 \mapsto \hat{w}_1, \dots, x_{n_\rho} \mapsto \hat{w}_{n_\rho}\}$ such that for each $i \in \{1, \dots, n_\rho\}$, $w_i \xrightarrow{h_i}_{\mathcal{R}} \hat{w}_i$ and, since n is the length of the longest derivation from t_0 , then $h_i < n$. Then, by the induction hypothesis, $\mathcal{F}_{BT}(\mathcal{R}) \vdash w_i \mapsto \hat{w}_i$ for each $i \in \{1, \dots, n_\rho\}$. Let $\hat{t}_{k-1} = l\hat{\rho}$. Thus, $\mathcal{F}_{BT}(\mathcal{R}) \vdash t_{k-1} \mapsto \hat{t}_{k-1}$, since the maximal context $C[]$ of t_{k-1} and \hat{t}_{k-1} is the same.

Now, we consider the cases when \mathcal{R} is confluent and \mathcal{R} is BT -based separately.

\mathcal{R} is confluent) Then, $r\widehat{\rho} \rightarrow_{\mathcal{R}}^* t_n$. By the induction hypothesis, $\mathcal{F}_{BT}(\mathcal{R}) \vdash \widehat{t}_{k-1} \mapsto t_n$. Since $\mathcal{F}_{BT}(\mathcal{R})$ is narrowing-wise and $\widehat{\rho}$ is BT -based, there are substitutions η, θ and a term $t'_n \in \mathcal{T}(\Sigma, \mathcal{V})$ such that $r \rightsquigarrow_{\eta, \mathcal{F}_{BT}(\mathcal{R})}^* t'_n$, $\widehat{\rho}|_r = (\eta\theta)|_r$, and $t_n = t'_n\theta$. Thus, by definition, $l\eta \mapsto t'_n$ has eventually been added to the fixpoint (though possibly removed) and $\mathcal{F}_{BT}(\mathcal{R}) \vdash l\eta \mapsto t'_n$. Finally $\mathcal{F}_{BT}(\mathcal{R}) \vdash t_0 \mapsto t_n$, since $t_0 \rightarrow_{\mathcal{F}_{BT}(\mathcal{R})}^* \widehat{t}_{k-1} \rightarrow_{\mathcal{F}_{BT}(\mathcal{R})} r\widehat{\rho} \rightarrow_{\mathcal{F}_{BT}(\mathcal{R})}^* t'_n\theta = t_n$.

\mathcal{R} is BT -based) Since $\mathcal{F}_{BT}(\mathcal{R})$ is narrowing-wise, there are substitutions η, ρ', θ and a term $t'_n \in \mathcal{T}(\Sigma, \mathcal{V})$ such that $r \rightsquigarrow_{\eta, \mathcal{F}_{BT}(\mathcal{R})}^* t'_n$, $\rho|_r \rightarrow_{\mathcal{R}}^* \rho'$, $\rho'|_r = (\eta\theta)|_r$, and $t_n = t'_n\theta$. Thus, by definition, $l\eta \mapsto t'_n$ has eventually been added to the fixpoint (though possibly removed) and $\mathcal{F}_{BT}(\mathcal{R}) \vdash l\eta \mapsto t'_n$. Since we do not require confluence, there may be several possible terms $\widehat{w}_{i,1}, \dots, \widehat{w}_{i,n_{i,\rho}}$ for each $i \in \{1, \dots, n_\rho\}$. Note that ρ', η, θ are BT -based, since $t_n \in BT$. Therefore, we arbitrarily choose a $\widehat{w}_{i,j}$ that is compatible with the substitution ρ' , i.e., we choose $\widehat{w}_{i,j}$, $j \in \{1, \dots, n_{i,\rho}\}$, for each $i \in \{1, \dots, n_\rho\}$ such that $\widehat{\rho}|_r = \rho'|_r$. Finally, $\mathcal{F}_{BT}(\mathcal{R}) \vdash t_0 \mapsto t_n$, since $t_0 \rightarrow_{\mathcal{F}_{BT}(\mathcal{R})}^* \widehat{t}_{k-1} \rightarrow_{\mathcal{F}_{BT}(\mathcal{R})} r\widehat{\rho} \rightarrow_{\mathcal{F}_{BT}(\mathcal{R})}^* t'_n\theta = t_n$. ■

Lemma A.2 *Let $BT \in \{\text{rnf}, \text{eval}\}$. Let \mathcal{R} be a topmost TRS. Let $t, s \in \mathcal{T}(\Sigma, \mathcal{V})$. If $t \mapsto s \in \mathcal{B}_{BT}^{\mathcal{V}}(\mathcal{R})$, then $\mathcal{F}_{BT}(\mathcal{R}) \vdash t \mapsto s$.*

Proof. Let $u, v \in \mathcal{T}(\Sigma, \mathcal{V})$ and $v \in BT$ such that $u \rightarrow_{\mathcal{R}}^* v$. Since \mathcal{R} is topmost, every rewriting step is performed at the top position, i.e., we consider $u = t_0 \xrightarrow{\epsilon}_{\mathcal{R}} t_1 \dots \xrightarrow{\epsilon}_{\mathcal{R}} t_n = v$. Then, we prove $\mathcal{F}_{BT}(\mathcal{R}) \vdash t_0 \mapsto t_n$ by induction on n .

$n = 0$) Immediate, since $t_0 = t_n \in BT$.

$n > 0$) Here we have $t_0 \xrightarrow{\epsilon}_{l \rightarrow r} t_1 \rightarrow_{\mathcal{R}}^{n-1} t_n$ where $l \rightarrow r \in \mathcal{R}$, $t_n \in BT$, and $t_1 = r\rho$ for some substitution ρ . Since $\mathcal{F}_{BT}(\mathcal{R})$ is narrowing-wise by Lemma 4.14, there are substitutions η, θ and a term $t'_n \in \mathcal{T}(\Sigma, \mathcal{V})$ such that $r \rightsquigarrow_{\eta, \mathcal{F}_{BT}(\mathcal{R})}^* t'_n$, $\rho|_r = (\eta\theta)|_r$, and $t_n = t'_n\theta$. Thus, by definition, $l\eta \mapsto t'_n$ has eventually been added to the fixpoint (though possibly removed) and $\mathcal{F}_{BT}(\mathcal{R}) \vdash l\eta \mapsto t'_n$. Finally, $\mathcal{F}_{BT}(\mathcal{R}) \vdash t_0 \mapsto t_n$, since $t_0 = l\rho \rightarrow_{\mathcal{F}_{BT}(\mathcal{R})} t_1 = r\rho \rightarrow_{\mathcal{F}_{BT}(\mathcal{R})}^* t'_n\theta = t_n$. ■

In the following, we exchange the order of appearance of Theorems 4.26 and 4.35 for simplicity.

Theorem 4.35 (Equivalence with Denotational Semantics) *Let $BT \in \{\text{rnf}, \text{eval}\}$. Let \mathcal{R} be either a*

- *BT-defined, terminating, denotation-compact TRS that is either confluent or BT-based; or*
- *topmost TRS.*

Then, $\mathcal{F}_{BT}(\mathcal{R}) = \mathcal{O}_{BT}(\mathcal{R})$.

Proof. Consider the case when \mathcal{R} is a *BT-defined*, terminating, denotation-compact TRS that is either confluent or *BT-based*. First note that, by Corollary 4.32, $\mathcal{O}_{BT}(\mathcal{R}) = \text{zip}(\mathcal{B}_{BT}^\vee(\mathcal{R}))$. If $t \mapsto s \in \mathcal{F}_{BT}(\mathcal{R})$, then $t \mapsto s \in \text{zip}(\mathcal{B}_{BT}^\vee(\mathcal{R}))$ by Theorem 4.19. If $t \mapsto s \in \text{zip}(\mathcal{B}_{BT}^\vee(\mathcal{R}))$, then $t \mapsto s \in \mathcal{F}_{BT}(\mathcal{R})$ by Lemma A.1.

If \mathcal{R} is a topmost TRS, then the proof is similar but using Lemma A.2 instead of Lemma A.1. ■

Theorem 4.26 (Soundness and Completeness) *Let $BT \in \{\text{rnf}, \text{eval}\}$. Let \mathcal{R} be either a*

- *BT-ground-defined, terminating, denotation-compact TRS that is either confluent or BT-based; or*
- *topmost TRS.*

Then, $\mathcal{B}_{BT}(\mathcal{R}) = \text{unzip}(\mathcal{F}_{BT}(\mathcal{R})) \cap (\mathcal{T}(\Sigma) \times \mathcal{T}(\Sigma))$.

Proof. The proof is perfectly analogous to the proof of Theorem 4.35 by replacing *BT-definedness* with the less restrictive *BT-ground-definedness* condition. This is because ground terms can be rewritten only to ground terms when no extra variables are allowed in right-hand sides of rules. ■

B More properties of zip/unzip

We show here, in more details, what happens in the general case of dealing with interpretations that are not necessarily compactable. Given $I \subseteq \mathbb{W}$, we say I is *meaningless* if it contains only semantic rules of the form $t \mapsto t$; and *meaningful* otherwise. Note that meaningless sets generate, by rewriting consequences, only meaningless sets and meaningful sets cannot be obtained by rewriting consequences from meaningless sets.

First, we provide a generic notion of semantic generator.

Definition B.1 (Generator Subset) *Given $I \subseteq \mathbb{W}$, we say that $G \subseteq I$ is a generator subset of I , written $G \sqsubseteq I$, if*

1. $G \vdash I$ (i.e., $I \subseteq \text{unzip}(G)$), and
2. $\forall r \in G, G - \{r\} \not\vdash r$ (i.e., $\text{zip}(G) = G$).

By $\text{allgens}(I)$ we denote the set of all generators, i.e., $\text{allgens}(I) = \{G \mid G \sqsubseteq I\}$.

When $G \sqsubseteq I$, all elements in $I - G$ are (roughly speaking) “redundant elements” of I that can be reconstructed from G . Note that, excluded the degenerate case of meaningless interpretations, generators are non-empty (i.e., $G = \emptyset$ cannot be a generator of a meaningful I).

Moreover, note that

- for any $G \sqsubseteq I$, $\text{unzip}(G) = \text{unzip}(I)$
because $G \subseteq I$ implies $\text{unzip}(G) \subseteq \text{unzip}(I)$, by Point 2 of Proposition 3.11, and $I \subseteq \text{unzip}(G)$ implies $\text{unzip}(I) \subseteq \text{unzip}(G)$, by Points 2 and 1 of Proposition 3.11.
- $G \sqsubseteq I$ if and only if $G \subseteq I$, $\text{zip}(G) = G$, $\text{unzip}(G) = \text{unzip}(I)$
the “only if” comes from Definition B.1 and the previous point and the vice versa since $\text{unzip}(G) = \text{unzip}(I)$ implies $G \vdash I$.
- $\text{allgens}(I) = \{X \subseteq I \mid \text{zip}(X) = X, \text{unzip}(X) = \text{unzip}(I)\}$
immediate from the previous point.
- a generator G cannot be properly contained into another generator G'
otherwise, for $r \in G' - G$, $G \vdash r$ and, since $G \subseteq G' - \{r\}$, $G' - \{r\} \vdash r$, contradicting that G' is a generator

Generator subsets may not be unique and all possible generators are not necessarily of the same cardinality, as shown in the following example.

Example B.2

Consider again the set I of Example 3.9. According to Definition B.1 there are several possible generators of I :

1. $\{a \mapsto b, b \mapsto c, c \mapsto a\}$,
2. $\{b \mapsto a, c \mapsto b, a \mapsto c\}$,
3. $\{a \mapsto b, b \mapsto a, a \mapsto c, c \mapsto a\}$,
4. $\{b \mapsto c, c \mapsto b, a \mapsto c, c \mapsto a\}$,
5. $\{a \mapsto b, b \mapsto a, b \mapsto c, c \mapsto b\}$,

Note that, e.g. $\{a \mapsto b, b \mapsto c, c \mapsto a\} \vdash b \mapsto a$.

Nevertheless, there is no “internal” redundancy in any such a generator subset and all the original elements can be reconstructed from it. In this sense, the actual choice of a generator subset can be considered irrelevant. Moreover, in Corollary B.8 below we prove that a unique, minimal generator subset exists for any meaningful $I \subseteq \mathcal{B}_{\text{rnf}}^{\mathcal{V}}(\mathcal{R})$ (and thus $I \subseteq \mathcal{B}_{\text{eval}}^{\mathcal{V}}(\mathcal{R})$). Actually, for these semantics, Definition B.1 boils down to the operator zip as given in Definition 3.7.

Note that, in general, $\text{zip}(I)$ is not necessarily a generator of I , as we already showed in Example 3.9. However, if for any $r \in I$ we have $I - \{r\} \not\vdash r$, then $A - \{r\} \not\vdash r$ for each $A \subseteq I$. This implies that $\text{zip}(I)$ is *contained* in any generator of I . However, the fact that $\text{zip}(I)$ is not a generator of I does not generally imply $\text{zip}(I) = \emptyset$. For instance, if we consider the set $I' := I \cup \{a \mapsto d\}$ for the set I of Example 3.9, we have that $\text{zip}(I') = \{a \mapsto d\}$ and $\text{zip}(I')$ is not a generator of I' .

The interesting thing is that, even when zip fails to work, generator subsets may exist (as shown by Examples B.2 and 3.9).

Moreover it is possible to define an increasing succession of subsets of I whose limit is exactly one generator of I . Unfortunately we are not guaranteed that the limit always exist.

Proposition B.3 *For any meaningful $I \subseteq \mathbb{W}$ it is possible to define a succession of subsets of I , such that its limit (if it exists) is one generator subset of I .*

Proof. Let us consider two meaningful sets I_1, I_2 , we say that $I_1 \ll I_2$ iff $\text{unzip}(I_1) \subset \text{unzip}(I_2)$ or $\text{unzip}(I_1) = \text{unzip}(I_2)$ and $I_1 \subseteq I_2$. Let us note that \ll is a partial order and that $I_1 \ll I_2 \implies I_2 \vdash I_1$.

Thus, let us choose any arbitrary total ordering of $I := \{e_1, e_2, \dots\}$ (this is possible since \mathbb{W} is countably infinite). Let us define inductively $G_1 := \{e_1\}$ and

$$G_{i+1} := \begin{cases} G_i & \text{if } G_i \vdash e_{i+1} \\ \text{reduce}(G_i \cup \{e_{i+1}\}) & \text{otherwise} \end{cases}$$

where, for a finite set $F = \{e'_1, \dots, e'_m\}$, $\text{reduce}(F) := F_m$ where the F_i are iteratively defined as $F_0 := F$ and

$$F_{i+1} := \begin{cases} F_i & \text{if } F_i - \{e'_{i+1}\} \not\vdash e'_{i+1} \\ F_i - \{e'_{i+1}\} & \text{otherwise} \end{cases}$$

It is easy to see that for $F' = \text{reduce}(F)$, $F' \subseteq F$, $F' \vdash F$, and for all $r \in F'$, $F' - \{r\} \not\vdash r$, i.e., $\text{reduce}(F) \subseteq F$. By induction hypothesis, G_i is non empty, $G_i \vdash G_{i-1}$, $G_i \vdash \{e_1, \dots, e_i\}$ and $\forall r \in G_i, G_i - \{r\} \not\vdash r$. Now we can conclude that G_{i+1} is non empty, $G_{i+1} \vdash G_i$, $G_{i+1} \vdash \{e_1, \dots, e_{i+1}\}$ and $\forall r \in G_{i+1}, G_{i+1} - \{r\} \not\vdash r$. Now, in case that I is finite, i.e. $I = \{e_1, \dots, e_n\}$ we can conclude that G_{n+1} is a generator for I , since $G_n \vdash \{e_1, \dots, e_n\}$ and since $\forall r \in G_n, G_n - \{r\} \not\vdash r$. Let us thus consider the case in which I is infinite. Clearly the sequence G_1, G_2, \dots will also be infinite. Let us note that this sequence is a chain, i.e. $\forall i \in \omega. G_i \ll G_{i+1}$. We will now show that if this chain has a least upper bound, such element is one generator of I . In fact, let $G := \bigsqcup_i G_i$. Then, since G is an upper bound, $\forall i \in \omega. G_i \ll G$, and hence $G \vdash G_i \vdash \{e_1, \dots, e_i\}$, and hence $G \vdash I$. Furthermore, G is not redundant, i.e. $\forall r \in G. G - \{r\} \not\vdash r$. In fact, there are only two possibilities: either (1) there exists $j < \omega$ such that $\text{unzip}(G_j) = \text{unzip}(G)$, and hence clearly $G_j = G$, and G_j is not redundant by construction or (2) $\forall i < \omega. \text{unzip}(G_i) \subset \text{unzip}(G)$. Then, let us assume by contradiction that G is redundant, and that there exists $e_j \in G$ such that $G - \{e_j\} \vdash \{e_j\}$. Then, $\forall i < \omega. \text{unzip}(G_i) \subset \text{unzip}(G) = \text{unzip}(G - \{e_j\})$. Hence $G - \{e_j\}$ would be an upper bound strictly smaller than the least upper bound G , which is absurd. Finally, let us prove that $G \subseteq I$. In fact, I is an upper bound for the set $\{G_1, G_2, \dots\}$, and hence the least upper bound $G \ll I$, and $I \vdash G$. Since we proved that $G \vdash I$, then $\text{unzip}(G) = \text{unzip}(I)$, and hence $G \ll I \implies G \subseteq I$.

Clearly depending on the chosen total ordering on I we can have different generators.

Finally, let us observe that the existence of the least upper bound is not guaranteed for any possible input set I . ■

The following proposition shows that the construction presented in Proposition B.3 is a generalization of the notion of zip, which is retrieved as a particular case when the conditions of Lemma 3.16 holds.

Proposition B.4 *Let $BT \in \{\text{eval}_{\mathcal{R}}, \text{rnf}_{\mathcal{R}}\}$, and $I \subseteq \mathcal{B}_{BT}^{\vee}(\mathcal{R})$ then $\text{zip}(I)$ is the least upper bound of the sequence of sets constructed for the set I in Proposition B.3.*

Proof. Let us first note that By Lemma 3.16, $\text{zip}(I) \vdash I$, $\text{zip}(I)$ is an upper bound for the sequence G_1, G_2, \dots , constructed in Proposition B.3. Then let us prove that it is the least upper bound. In Proposition B.3 we showed that the least upper bound, if it exists, is a subset of I , and that for any upper bound G , $G \vdash I$. Thus, let us consider an upper bound $G \subseteq I$. Now we prove that $\text{zip}(I) \subseteq G$, and hence that $\text{zip}(I)$ is the least upper bound. By contradiction, assume that there exists $r \in \text{zip}(I)$ such that $r \notin G$. By definition of $\text{zip}(I)$, $I - \{r\} \not\vdash r$ and clearly since $G \subseteq I$, $G = G - \{r\} \not\vdash r \in I$, which is absurd. ■

Let us now restrict our attention to meaningful sets that have a generator, which we call *weakly compactable*. Clearly meaningful compactable sets are weakly compactable and (as shown by Examples B.2 and 3.9) the viceversa is not true.

The following proposition shows that the notion of generator of Definition B.1 is a generalization of the notion of zip, because when $\text{zip}(I) \vdash I$ then $\text{zip}(I)$ is the unique generator of I .

Proposition B.5 *For any weakly compactable $I \subseteq \mathbb{W}$,*

1. $\text{zip}(I) \subseteq \bigcap \text{allgens}(I)$ ($= \bigcap \{G \mid G \subseteq I\}$).
2. *If $\text{zip}(I)$ is a generator of I then it is the unique generator of I .*

Proof. First of all note that, since I is weakly compactable, $I \neq \emptyset$ and $\text{allgens}(I) \neq \emptyset$.

For Point 1, the case where I is a singleton is straightforward.

Otherwise, let us recall that if for any $r \in I$ we have that $I - \{r\} \not\vdash r$, then for any $G \in \text{allgens}(I)$ we have that $G - \{r\} \not\vdash r$, $r \in \text{zip}(I)$, and $\text{zip}(I) \subseteq G$. Thus $\text{zip}(I) \subseteq \bigcap \text{allgens}(I)$.

For Point 2, if $\text{zip}(I)$ is a generator, by the fact that generators cannot be properly contained, then $\text{zip}(I)$ has to be the unique generator. ■

The following results are straightforward consequences of the two previous propositions.

Corollary B.6 *If I is compactable, then $\text{zip}(I)$ is the unique generator of I .*

Proof. It follows from Proposition B.5 by the fact that $\text{zip}(I) \vdash I$. ■

Corollary B.7 *If I is zipped, then it is the unique generator of itself.*

Finally, we show that zip is the unique generator for $BT \in \{\text{eval}_{\mathcal{R}}, \text{rnf}_{\mathcal{R}}\}$.

Corollary B.8 *For $BT \in \{\text{eval}_{\mathcal{R}}, \text{rnf}_{\mathcal{R}}\}$ and a set $I \subseteq \mathcal{B}_{BT}^{\mathcal{V}}(\mathcal{R})$, $\text{zip}(I)$ is the unique generator subset of I according to Definition B.1.*

Proof. By Corollary B.6 it suffices to prove that $\text{zip}(I) \vdash I$. But this follows from Lemma 3.16. ■

Note that Corollary B.8 only holds for $BT \in \{\text{eval}_{\mathcal{R}}, \text{rnf}_{\mathcal{R}}\}$, as it is witnessed by the following counter-example.

Example B.9

Consider again the interpretation I of Example B.2, which consider semantics $\mathcal{B}_{\text{red}}^{\mathcal{V}}(\mathcal{R})$. According to Definition B.1 there are two minimal generators of I : $\{a \mapsto b, b \mapsto c, c \mapsto a\}$ and $\{b \mapsto a, c \mapsto b, a \mapsto c\}$. However, $\text{zip}(I) = \emptyset$, which is not a generator subset of I .

As we saw from examples the $\cdot \sqsubseteq I$ relation is not, in general, a function. However in order to define a concrete domain which is a complete lattice, the actual choice of the generator subset is irrelevant. Any generator of I is able to reconstruct I and this is the only property which matters. Hence let gen be a function such that $\text{gen}(I) \sqsubseteq I$. Moreover let us denote the equivalence $\text{unzip}(A) = \text{unzip}(B)$ by $A \equiv_{\vdash} B$ (which is equivalent to $A \vdash B \wedge B \vdash A$). Then, independently of the chosen possibility, gen satisfies the following similar, but more regular properties than those of zip .

Proposition B.10 *Let $I \subseteq \mathbb{W}$ be a weakly compactable set of rules.*

1. gen is idempotent, i.e., $\text{gen}(\text{gen}(I)) = \text{gen}(I)$.
2. $\text{unzip} \circ \text{gen}$ is extensive w.r.t. \sqsubseteq , i.e., $I \subseteq \text{unzip}(\text{gen}(I))$.
3. $\text{unzip}(\text{gen}(I)) = \text{unzip}(I)$
4. $\forall U \in \mathbb{S}. \text{unzip}(\text{gen}(U)) = U$.
5. gen is monotone w.r.t. \sqsubseteq on unzipped sets, i.e., $\forall U_1, U_2 \in \mathbb{S}. U_1 \subseteq U_2 \implies \text{gen}(U_1) \sqsubseteq \text{gen}(U_2)$.
6. $\text{gen}(\text{unzip}(I)) \equiv_{\vdash} I$.

Proof. Point 1 is straightforward.

For Point 2, by Equation (3.3), $\text{unzip}(\text{gen}(I)) = \{e \mid \text{gen}(I) \vdash e\}$. By Definition B.1, $\text{gen}(I) \vdash I$ and thus $I \subseteq \text{unzip}(\text{gen}(I))$.

For Point 3, since $\text{gen}(I) \subseteq I$, by Point 2 of Proposition 3.11, we have $\text{unzip}(\text{gen}(I)) \subseteq \text{unzip}(I)$. Moreover, by Point 2 and Point 2 of Proposition 3.11, $\text{unzip}(I) \subseteq \text{unzip}(\text{unzip}(\text{gen}(I)))$. By Point 1 of Proposition 3.11, $\text{unzip}(I) \subseteq \text{unzip}(\text{gen}(I))$. Thus the thesis follows.

For Point 4, since $U = \text{unzip}(U)$, by Point 3, $\text{unzip}(\text{gen}(U)) = \text{unzip}(U) = U$.

For Point 5, by Point 3 and since $U_i = \text{unzip}(U_i)$, $\text{unzip}(\text{gen}(U_i)) = \text{unzip}(U_i) = U_i$. Thus from $U_1 \subseteq U_2$ follows $\text{unzip}(\text{gen}(U_1)) \subseteq \text{unzip}(\text{gen}(U_2))$, which is $\text{gen}(U_1) \sqsubseteq \text{gen}(U_2)$.

For Point 6, by definition of equivalence \equiv_{\vdash} , the thesis is equivalent to $\text{unzip}(\text{gen}(\text{unzip}(I))) = \text{unzip}(I)$. But, by Point 3, $\text{unzip}(\text{gen}(\text{unzip}(I))) = \text{unzip}(\text{unzip}(I))$. Then the thesis follows from Point 1 of Proposition 3.11. ■

Our semantic domain \mathbb{C} is not a complete lattice, as it is not closed under least upper bound. Consider for example $\{a \mapsto b, b \mapsto c\}$ and $\{c \mapsto a\}$: it is not possible to find a compactable set bigger than both.

If we use gen instead of zip we can “extend” \mathbb{C} to a complete lattice, in the sense that we can define a semantic domain into which \mathbb{C} can be embedded.

Definition B.11 (Extended Semantic Domain) *The extended semantic domain $\bar{\mathbb{C}}$ is the set $\bar{\mathbb{C}} := \{\text{gen}(I)/_{\equiv_{\perp}} \mid I \subseteq \mathbb{W}, I \text{ weakly compactable}\}$ ordered by*

$$[A]_{\equiv_{\perp}} \sqsubseteq [B]_{\equiv_{\perp}} := \text{unzip}(A) \subseteq \text{unzip}(B). \quad (\text{B.1})$$

Note that, by idempotence of gen and the relation between zip and gen , (all canonical representatives of) the elements of $\bar{\mathbb{C}}$ are zipped sets. Thus $\mathbb{C}/_{\equiv_{\perp}} \subseteq \bar{\mathbb{C}}$.

$\bar{\mathbb{C}}$ is a “faithful” representation of \mathbb{S} , which contains also (the representations of) the pathological sets which contain mutually recursive rewriting dependencies, as proved by the following result.

Proposition B.12 *($\text{gen}/_{\equiv_{\perp}}$, unzip) is an order preserving isomorphism between \mathbb{S} and $\bar{\mathbb{C}}$, and thus $\bar{\mathbb{C}}$ is a complete lattice, where*

$$\sqcap_i [A_i]_{\equiv_{\perp}} = \text{gen}(\cap_i \text{unzip}(A_i))/_{\equiv_{\perp}} \quad (\text{B.2})$$

$$\sqcup_i [A_i]_{\equiv_{\perp}} = \text{gen}(\cup_i A_i)/_{\equiv_{\perp}} \quad (\text{B.3})$$

Proof. The isomorphism follows from Points 4 and 6 of Proposition B.10. Order preservation, from \mathbb{S} to $\bar{\mathbb{C}}$, comes from Point 5 of Proposition B.10; while from $\bar{\mathbb{C}}$ to \mathbb{S} comes by Equation (B.1).

Equations (B.2) and (B.3) follow by defining into $\bar{\mathbb{C}}$ the dual (by the isomorphism) of \cup and \cap of \mathbb{S} . ■

Recall, anyhow, that for what concerns (our present) semantics of interest, the much simpler domain \mathbb{C} is sufficient (all semantics properties we are interested in have been proved for domain \mathbb{C}). Nevertheless, the domain $\bar{\mathbb{C}}$ can be more interesting for Abstract Interpretation purposes, where (many) powerful results hold for complete lattices.