Automatic Inference of Term Equivalence in Term Rewriting Systems

Marco Comini[1] and Luca Torella[2]

[1] DIMI, University of Udine, Italy
marco.comini@uniud.it
[2] DIISM, University of Siena, Italy
luca.torella@unisi.it

Abstract

In this paper we propose a method to automatically infer algebraic property-oriented specifications from Term Rewriting Systems. Namely, having three semantics with suitable properties, given the source code of a TRS we infer a specification which consists of a set of *most general* equations relating terms that rewrite, for all possible instantiations, to the same set of constructor terms.

The semantic-based inference method that we propose can cope with non-constructor-based TRSs, and considers non-ground terms. Particular emphasis is put on avoiding the generation of "redundant" equations that can be a logical consequence of other ones.

1 Introduction

In the last years there has been a growing interest in the research on automatic inference of high-level specifications from an executable or the source code of a system. This is probably due to the fact that the size of software systems is growing over time and certainly one can greatly benefit from the use of automatic tools. There are several proposals, like [1, 7, 6], which have proven to be very helpful.

Specifications have been classified by their characteristics [8]. It is common to distinguish between *property-oriented* specifications and *model-oriented* or *functional* specifications. Property-oriented specifications are of higher description level than other kinds of specifications: they consist of an indirect definition of the system's behavior by stating a set of properties, usually in the form of axioms, that the system must satisfy [11, 10]. In other words, a specification does not represent the functionality of the program (the output of the system) but its properties in terms of relations among the operations that can be invoked in the program (i.e., identifies different calls that have the same behavior when executed). This kind of specifications is particularly well suited for program understanding: the user can realize non-evident information about the behavior of a given function by observing its relation to other functions. Moreover, the inferred properties can manifest potential symptoms of program errors which can be used as input for (formal) validation and verification purposes.

We can identify two mainstream approaches to perform the inference of specifications: glass-box and black-box. The glass-box approach [1] assumes that the source code of the program is available. In this context, the goal of inferring a specification is mainly applied to document the code, or to understand it. Therefore, the specification must be more succinct and comprehensible than the source code itself. The inferred specification can also be used to automate the testing process of the program or to verify that a given property holds [1]. The black-box approach [7, 6] works only by running the executable. This means that the only information used during the inference process is the input-output behavior of the program. In this setting,

the inferred specification is often used to discover the functionality of the system (or services in a network) [6]. Although black-box approaches work without any restriction on the considered language – which is rarely the case in a glass-box approach – in general, they cannot *guarantee* the correctness of the results (whereas indeed semantics-based glass-box approaches can).

For this work, we developed a (glass-box) *semantic-based* algebraic specification synthesis method for the Term Rewriting Systems formalism that is *completely automatic*, i.e., needs only the source TRS to run. Moreover, *the outcomes are very intuitive* since specifications are sets of equations of the form $e_1 = e_2$, where e_1, e_2 are generic TRS expressions, so the user does not need any kind of extra knowledge to interpret the results. The underpinnings of our proposal are radically different from other works for functional programming, since the presence of *non-confluence* or *non-constructor-based* TRSs poses several additional problems.

1.1 Notations

We assume that the reader is familiar with the basic notions of term rewriting. For a thorough discussion of these topics, see [9]. In the paper we use the following notations. $\mathcal{V}$ denotes a (fixed) countably infinite set of variables and $\mathcal{T}(\Sigma, \mathcal{V})$ denotes the terms built over signature Σ and variables $\mathcal{V}$. $\mathcal{T}(\Sigma, \varnothing)$ are ground terms. Substitutions over $\mathcal{T}(\Sigma, \varnothing)$ are called ground substitutions. Σ is *partitioned* in $\mathcal{D}$, the *defined* symbols, and $\mathcal{C}$, the *constructor* symbols. $\mathcal{T}(\mathcal{C}, \mathcal{V})$ are *constructor terms*. Substitutions over $\mathcal{T}(\mathcal{C}, \mathcal{V})$ are said constructor substitutions. $C[t_1, \ldots, t_n]$ denotes the replacement of $t_1, \ldots, t_n$ in context C. A TRS $\mathcal{R}$ is a set of rules $l \to r$ where $l = f(t_1, \ldots, t_n)$, $l, r \in \mathcal{T}(\Sigma, \mathcal{V})$, $var(r) \subseteq var(l)$ and $f \in \mathcal{D}$. $t_1, \ldots, t_n$ are the argument patterns of $l \to r$ and need not necessarily be in $\mathcal{T}(\mathcal{C}, \mathcal{V})$, unlike in functional programming, where only *constructor-based* TRSs are considered (with $t_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$).

2 Many notions of equivalence

In the functional programming paradigm an equation $e_1 = e_2$ is typically *interpreted* as a property that holds for any *well-typed constructor ground* instance of the variables occurring in the equation. Namely, *for all bindings of variables with well-typed (constructor ground) terms* ϑ the constructor term computed for the calls $e_1\vartheta$ and $e_2\vartheta$ is the same. In functional programming we can consider only constructor instances because, by having constructor-based confluent TRSs, the set of values for non-constructor instances is the same as for constructor ones.

Differently from the functional programming case, the TRS formalism admits variables in initial terms and defined symbols in the patterns of the rules; moreover rules are evaluated non-deterministically and can rewrite to many constructor terms. Thus an equation can be interpreted in many different ways. We will discuss the key points of the problem by means of a (very simple) illustrative example.

Example 2.1 Consider the following (non constructor based) TRS R where we provide a pretty standard definition of the arithmetic operations `+`, `-` and (modulo) `%`:

```
0 + x -> x              x - 0 -> x              x % s(y) -> (x - s(y)) % s(y)
s(x) + y -> s(x+y)      s(x) - s(y) -> x - y    (0 - s(x)) % s(y) -> y - x
```

Note that, since the TRS formalism is untyped, a term like `0 + a` is admissible and is evaluated to constructor term `a`. ∎

For this TRS, one *could* expect to have in its property-oriented specification equations like:

$$\texttt{(x + y) + z} = \texttt{x + (y + z)} \tag{2.1}$$
$$\texttt{x - (y + z)} = \texttt{(x - y) - z} \tag{2.2}$$
$$\texttt{(x - y) - z} = \texttt{(x - z) - y} \tag{2.3}$$
$$\texttt{0 + x} = \texttt{x} \tag{2.4}$$
$$\texttt{x + 0} = \texttt{x} \tag{2.5}$$
$$\texttt{x + y} = \texttt{y + x} \tag{2.6}$$

These equations, of the form $e_1 = e_2$, can be read as *the (observable) outcomes of* e_1 *are the same of* e_2. The first essential thing is to formalize the meaning of "observable outcomes" of the evaluation of an expression e (which contains variables). In the TRS formalism we have several possible combinations. First it is technically possible to *literally* interpret variables in e and decide to observe either the set of normal forms of e or the set of constructor terms of e. However, as can be quickly verified, only (2.4) is valid in either of these forms. Indeed, consider, for example, terms `x - (y + z)` and `(x - y) - z`. For both terms no rule can be applied, hence they are both (non constructor) normal forms and thus (2.2) is not literally valid. Clearly this is quite a radical choice. An interesting alternative would be to request that, for any possible interpretation of variables with any term, the expressions rewrite to the same set of constructor terms. Formally, by defining the rewriting behavior of a term t as

$$\mathcal{B}[\![t;\mathcal{R}]\!] := \left\{ \vartheta \cdot s \;\middle|\; t\vartheta \underset{\mathcal{R}}{\longrightarrow}^{*} s, s \in \mathcal{T}(\mathcal{C},\mathcal{V}),\ \vartheta{:}\,\mathcal{V} \to \mathcal{T}(\Sigma,\mathcal{V}),\ dom(\vartheta) \subseteq var(t) \right\} \tag{2.7}$$

we can interpret $e_1 = e_2$ as $e_1 =_{RB} e_2 :\Longleftrightarrow \mathcal{B}[\![e_1;\mathcal{R}]\!] = \mathcal{B}[\![e_2;\mathcal{R}]\!]$. In the following, we call this equivalence *rewriting behavior equivalence*. Actually, eqs. (2.1) to (2.4) are valid w.r.t. $=_{RB}$.

Note that if we would have chosen to use normal forms instead of constructor terms in (2.7) (i.e., $s \not\to$ instead of $s \in \mathcal{T}(\mathcal{C},\mathcal{V})$), then we would still have the same situation described before where only (2.4) holds.

The other equations (eqs. (2.5) and (2.6)) are not valid in this sense. For instance, $\mathcal{B}[\![\texttt{x + 0}]\!] = \{\{\texttt{x}/t\} \cdot \texttt{s}^i(\texttt{0}) \,|\, t \to^{*} \texttt{s}^i(\texttt{0})\}$ [1] while $\mathcal{B}[\![\texttt{x}]\!] = \{\{\texttt{x}/t\} \cdot v \,|\, t \underset{\mathcal{R}}{\longrightarrow}^{*} v, v \in \mathcal{T}(\mathcal{C},\mathcal{V})\}$ (and then $\varepsilon \cdot \texttt{x} \in \mathcal{B}[\![\texttt{x}]\!] \smallsetminus \mathcal{B}[\![\texttt{x + 0}]\!]$). These equations are not valid essentially because we have variables which cannot be instantiated, but, if we consider ground constructor instancies which "trigger" in either term an evaluation to constructor terms, then we actually obtain the same constructor terms. For example for $t_1 = \texttt{x}$ and $t_2 = \texttt{x + 0}$, for all $\vartheta_i = \{\texttt{x}/\texttt{s}^i(\texttt{0})\}$, we have $\mathcal{B}[\![t_1\vartheta_i]\!] = \mathcal{B}[\![t_2\vartheta_i]\!] = \{\varepsilon \cdot \texttt{s}^i(\texttt{0})\}$. Thus (2.5) holds in this sense.

Decidedly, also this notion of equivalence is interesting for the user. We can formalize it as $t_1 =_{G} t_2 :\Longleftrightarrow \forall\vartheta$ ground constructor. $\mathcal{B}[\![t_1\vartheta]\!] \cup \mathcal{B}[\![t_2\vartheta]\!] \subseteq \{\varepsilon \cdot t \,|\, t \in \mathcal{T}(\mathcal{C},\mathcal{V})\} \Rightarrow \mathcal{B}[\![t_1\vartheta]\!] = \mathcal{B}[\![t_2\vartheta]\!]$ We will call it *ground constructor equivalence*. Note that $=_G$ is the only possible notion in the pure functional paradigm where we can just have evaluations of ground terms and where we have only confluent constructor-based TRSs. In this case the latter definition boils down to: two expressions are equivalent if all its ground constructor instances rewrite to the same constructor term. This fact allows one to have an intuition of the reason why the problem of specification synthesis is definitively more complex in the full TRS paradigm.

Since we do not consider only constructor-based TRSs, there is another very relevant difference w.r.t. the pure functional case. For instance, let us consider the TRS Q obtained by adding

[1] Here by $\texttt{s}^i(\texttt{0})$ we mean i repeated applications of `s` to `0`, including the degenerate case for $i = 0$.

the rule `g((x - y) - z) -> x - (y + z)` to TRS R of Example 2.1. Let t_1 = `x - (y + z)` and t_2 = `(x-y)-z`. We have $\mathcal{B}[\![t_1;Q]\!] = \mathcal{B}[\![t_2;Q]\!] = \{\{x/0,y/0,z/0\}\cdot 0, \{x/s(0),y/0,z/0\}\cdot s(0), \{x/s(0),y/s(0),z/0\}\cdot 0, \{x/s(0),y/0,z/s(0)\}\cdot 0,\dots\}$. While the term `g(`$t_2$`)` has this same behavior, $\mathcal{B}[\![g(t_1);Q]\!] = \varnothing$.

In general, in the case of the TRS formalism, terms embedded within a context do not necessarily manifest the same behavior. Thus, it is also interesting to *additionally* ask to the equivalence notion $=_{RB}$ that the behaviors must be the same also when the two terms are embedded within any context. Namely, $e_1 =_C e_2 :\Longleftrightarrow \forall\,\text{context}\, C.\ \mathcal{B}[\![C[e_1];\mathcal{R}]\!] = \mathcal{B}[\![C[e_2];\mathcal{R}]\!]$ We call this equivalence *contextual equivalence*. We can see that $=_C$ is (obviously) stronger than $=_{RB}$, which is in turn stronger than $=_G$. Actually they are *strictly* stronger. Indeed, only eqs. (2.1) and (2.4) are valid w.r.t. $=_C$ (while eqs. (2.2) and (2.3) are not).

We believe that *all* these notions are interesting for the user, thus we formalize our notion of (algebraic) specification as follows.

Definition 2.2 *A specification $\mathcal{S}$ is a set of (sequences of) equations of the form $t_1 =_K t_2 =_K \dots =_K t_n$, with $K \in \{C, RB, G\}$ and $t_1, t_2, \dots, t_n \in \mathcal{T}(\Sigma,\mathcal{V})$.*

Thus, for TRS Q we would get the following (partial) specification:

```
(x + y) + z =C x + (y + z)                    0 + x =C x
x - (y + z) =RB (x - y) - z                   (x - y) - z =RB (x - z) - y
x + y =G y + x                                x + 0 =G x
(x + y) % z =G ((x % z) + (y % z)) % z        (x + y) % y =G x % y
(x - y) % z =G (((x + z) % z) - (y % z)) % z
```

In the following we present a first proposal of a semantics-based method that infers such specifications for TRSs and tackles the presented issues (and discuss about its limitations). It is an adaptation for the TRS paradigm of ideas from [2] for the Functional Logic paradigm. This adaptation is not straightforward, since the Functional Logic paradigm is quite similar to the Functional paradigm, but considerably different from the TRS paradigm. Moreover, this work significantly extends the inference process by tackling equations like $f(x,y) = f(y,x)$ which are really important.

3 Deriving specifications from TRSs

The methodology we are about to present is parametric w.r.t. three semantics evaluation functions which need to enjoy some properties. Namely,

$\mathcal{E}^{RB}[\![t;\mathcal{R}]\!]$ gives the *rewriting behavior* (RB) semantics of term t with (definitions from) TRS $\mathcal{R}$. This semantics has to be fully abstract w.r.t. rewriting behavior equivalence. Namely, we require that $\mathcal{E}^{RB}[\![t_1;\mathcal{R}]\!] = \mathcal{E}^{RB}[\![t_2;\mathcal{R}]\!] \Longleftrightarrow t_1 =_{RB} t_2$.

$\mathcal{E}^{C}[\![t;\mathcal{R}]\!]$ gives the *contextual* (C) semantics of the term t with the TRS $\mathcal{R}$. This semantics has to be fully abstract w.r.t. contextual equivalence. Namely, $\mathcal{E}^{C}[\![t_1;\mathcal{R}]\!] = \mathcal{E}^{C}[\![t_2;\mathcal{R}]\!] \Longleftrightarrow t_1 =_C t_2$.

$\mathcal{E}^{G}[\![t;\mathcal{R}]\!]$ gives the *ground* (G) semantics of the term t with the TRS $\mathcal{R}$. This semantics has to be fully abstract w.r.t. ground constructor equivalence. Namely, $\mathcal{E}^{G}[\![t_1;\mathcal{R}]\!] = \mathcal{E}^{G}[\![t_2;\mathcal{R}]\!] \Longleftrightarrow t_1 =_G t_2$.

The idea underlying the process of inferring specifications is that of computing the semantics of various terms and then identify all terms which have the same semantics. However, not all equivalences are as important as others, given the fact that many equivalences are simple logical consequences of others. For example, if $t_i =_C s_i$ then, for all constructor contexts C, $C[t_1, \ldots, t_n] =_C C[s_1, \ldots, s_n]$, thus the latter *derived* equivalences are uninteresting and should be omitted. Indeed, it would be desirable to synthesize the *minimal* set of equations from which, by deduction, all valid equalities can be derived. This is certainly a complex issue in testing approaches. With a semantics-based approach it is fairly natural to produce just the relevant equations. The (high level) idea is to proceed bottom-up, by starting from the evaluation of simpler terms and then newer terms are constructed (and evaluated) by using only semantically different arguments. Thus, by construction, only non-redundant equations are produced.

There is also another source of redundancy due to the inclusion of relations $=_K$. For example, since $=_C$ is the finer relation, $t =_C s$ implies $t =_{RB} s$ and $t =_G s$. To avoid the generation of coarser redundant equations, a simple solution is that of starting with $=_C$ equivalences and, once these are all settled, to proceed *only* with the evaluation of the $=_{RB}$ equivalences of *non* $=_C$ equivalent terms. Thereafter, we can evaluate the $=_G$ equivalences of *non* $=_{RB}$ equivalent terms.

Clearly, the *full* computation of a programs' semantics is not feasible in general. For the moment, for the sake of comprehension, we prefer to present the conceptual framework leaving out of the picture the issues related to decidability. We will show a possible decidable instance of the method in Section 3.1.

Let us describe in more detail the specification inference process. The input of the process consists of a TRS to be analyzed and two additional parameters: a *relevant* API, Σ^r, and a maximum term size, *max_size*. The *relevant* API allows the user to choose the operations in the program that will be present in the inferred specification, whereas the maximum term size limits the size of the terms in the specification. As a consequence, these two parameters tune the granularity of the specification, both making the process terminating and allowing the user to keep the specification concise and easy to understand.

The output consists of a set of equations represented by equivalence classes of terms (with distinct variables). Note that inferred equations may differ for the same program depending on the considered API and on the maximum term size. Similarly to other property-oriented approaches, the computed specification is complete up to terms of size *max_size*, i.e., it includes all the properties (relations) that hold between the operations in the relevant API and that are expressible by terms of size less or equal than *max_size*.

Terms are classified by their semantics into a data structure, which we call *classification*, consisting (conceptually) of a set of *equivalence classes* (*ec*) formed by

- $sem(ec)$: the semantics of (all) the terms in that class;
- $rep(ec)$: the *representative term* of the class ($rep(ec) \in terms(ec)$);
- $terms(ec)$: the set of terms belonging to that equivalence class;
- $epoch(ec)$: an integer to record the moment of introduction of that equivalence class.

The *representative term* is the term which is used in the construction of nested expressions when the equivalence class is considered. To output smaller equations it is better to choose the smallest term in the class (w.r.t. the function *size*), but any element of $terms(ec)$ can be used.

The inference process consists of successive phases, one for each kind of equality (in order of discriminating power, i.e., C, RB, G), as depicted in Figure 1.

Computation of the initial classification (epochs 0 and 1). The first phase of the algorithm, is the computation of the initial classification that is needed to compute the classification w.r.t. $=_C$. We initially create a classification which contains:

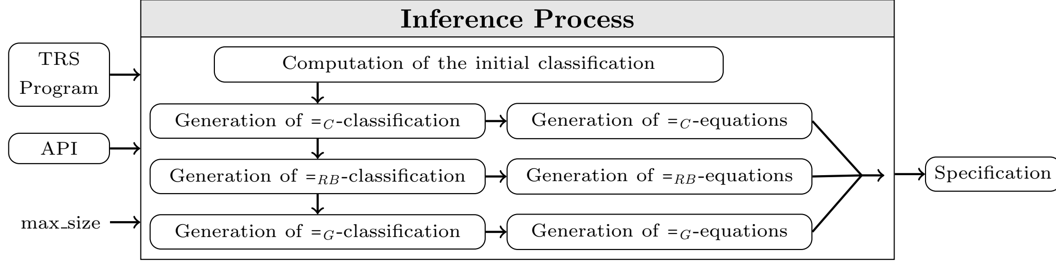

Figure 1: A general view of the inference process.

- one class for a free (logical) variable $\langle \mathcal{E}^C[\![x]\!], x, \{x\}, 0\rangle$;
- the classes for any constructor symbol, i.e., for all $c_{/n}$[2] $\in \mathcal{C}$ and $x_1, \dots, x_n$ distinct variables, $\langle \mathcal{E}^C[\![t]\!], t, \{t\}, 0\rangle$, where $t = c(x_1, \dots, x_n)$.

Then, for all symbols $f_{/n}$ of the relevant API, Σ^r, and distinct variables $x_1, \dots, x_n$, we *add to classification* the term $t = f(x_1, \dots, x_n)$ with semantics $s = \mathcal{E}^C[\![t; \mathcal{R}]\!]$ and epoch $m = 1$. This operation, denoted $addEC(t, s, m)$ is the most delicate part of the method. If we would not consider the generation of "permuted" equations like $f(x, y) = f(y, x)$, then the whole activity would just boil down to look for the presence of s in some equivalence class and then updating the classification accordingly (like in [2]). Handling of permutations cannot be done by naïvely adding all (different) permutations of s. In this way we would generate many redundant equations. Consider for instance the TRS Q

```
f(a,b,c) -> 0          f(a,c,b) -> 0          f(c,a,b) -> 0
f(b,a,c) -> 0          f(b,c,a) -> 0          f(c,b,a) -> 0
```

From the minimal set $f(x,y,z) =_C f(y,x,z) =_C f(x,z,y)$ we can generate all other valid permutations, like $f(x,y,z) =_C f(y,z,x) =_C f(z,x,y)$. Thus *in this case* we should only generate permutated equations where we just swap two variables. However, for the TRS R

```
f(a,b,c) -> 0          f(b,c,a) -> 0          f(c,a,b) -> 0
```

$f(x,y,z) \neq_C f(y,x,z) \neq_C f(x,z,y)$ but $f(x,y,z) =_C f(y,z,x)$. Moreover $f(x,y,z) =_C f(z,x,y)$ can be deduced by this. Thus *in this case* we should only generate permutated equations with a rotation of three variables, since all other rotations of three variables are just a consequence.

Thus, not all permutations have to be considered while adding a semantics to a classification, and it is not necessary to look for all permutations within the semantics already present in the classification. To generate only a minimal set of necessary permutated equations we need to consider, for each k variables, a set Π^n_k of *generators* of the permutations of n variables which do not move $n-k$ variables (note that $\Pi^n_1 = \{id\}$). Then, for a term $t = f(x_1, \dots, x_n)$, we start, sequentially for i from 1 to n, and look if, for some $\pi \in \Pi^n_i$, we have an equivalence class ec in the current classification whose semantics coincides with $s\pi$ (i.e., $ec = \langle s\pi, t', T, m'\rangle$). If it is found, then the term $t\pi^{-1}$ is added to the set of terms in ec (i.e., ec is transformed in $ec' = \langle s\pi, t', T \cup \{t\pi^{-1}\}, m'\rangle$) and we stop. Otherwise, we iterate with next i. If all fails, a new equivalence class $\langle s, t, T, m\rangle$ has to be created, but we have to determine the right term set

[2]Following the standard notation $f_{/n}$ denotes a function f of arity n.

T (to possibly generate equations like $f(x,y) = f(y,x)$). We initially start with $T = \{t\}$ and $j = n$, and sequentially for i from 2 to j, we check if, for some $\pi \in \Pi_i^j$, we have $s = s\pi$. If we find it, we add $t\pi$ to T, and decrement j by $i-1$ (i variables are now used in a valid equation and thus from this moment on $i-1$ variables have no longer to be considered). Otherwise we continue with next i. The final T is used for the new equivalence class.

For example, for $t = f(x,y,z)$ in TRS Q, we start with variables $\{x,y,z\}$ and $T = \{t\}$. Now consider, for instance, $\pi = (x\,y) \in \Pi_2^3$. Since $\mathcal{E}^C[\![t]\!] = \mathcal{E}^C[\![t\pi]\!]$ then $T = \{f(x,y,z), f(y,x,z)\}$ and then we drop x and remain with variables $\{y,z\}$. Now we have only $\pi' = (y\,z) \in \Pi_2^2$, and since $\mathcal{E}^C[\![t]\!] = \mathcal{E}^C[\![t\pi']\!]$ then $T = \{f(x,y,z), f(y,x,z), f(x,z,y)\}$, we drop y and (thus) finish. Instead, for $t = f(x,y,z)$ in TRS R we start with variables $\{x,y,z\}$ and $T = \{t\}$. Now consider, for instance, $\pi_1 = (x\,y) \in \Pi_2^3$. Since $\mathcal{E}^C[\![t]\!] \neq \mathcal{E}^C[\![t\pi_1]\!]$ then we consider $\pi_2 = (y\,z) \in \Pi_2^3$, and since $\mathcal{E}^C[\![t]\!] \neq \mathcal{E}^C[\![t\pi_2]\!]$ we increase i. Now we have only $\pi_3 = (x\,y\,z) \in \Pi_3^3$, and since $\mathcal{E}^C[\![t]\!] = \mathcal{E}^C[\![t\pi_3]\!]$ then $T = \{f(x,y,z), f(y,z,x)\}$, we drop x, y and (thus) finish.

Generation of $=_C$ classification (epochs 2 and above). The second phase of the algorithm, is the (iterative) computation of the successive epochs, until we complete the construction of the classification of terms w.r.t. $=_C$. At each iteration (with epoch k), for all symbols $f_{/n}$ of the relevant API Σ^r, we select from the current classification all possible combinations of n equivalence classes $ec_1, \dots, ec_n$ such that at least one ec_i was newly produced in the previous iteration (i.e., whose epoch is $k-1$). We build the term $t = f(rep(ec_1), \dots, rep(ec_n))$ which, by construction, has surely not been considered yet. Then, if $size(t) \leq max_size$, we compute the semantics $s = \mathcal{E}^C[\![t; \mathcal{R}]\!]$ and update the current classification by *adding to classification* the term t and its semantics s ($addEC(t,s,k)$) as described before.

If we have produced new equivalence classes then we continue to iterate. This phase eventually terminates because at each iteration we consider, by construction, terms which are different from those already existing in the classification and whose size is strictly greater than the size of its subterms (but the size is bounded by max_size).

The following example illustrates how the iterative process works:

Example 3.1 Let us use the program R of Example 2.1 and choose as relevant API the functions `+`, `-` and `%`. In the first step, the terms

$t_{1.1}$ = `x + y` $t_{1.2}$ = `x - y` $t_{1.3}$ = `x % y`

are built. Since (all permutations of) the semantics of all these terms are different, and different from the other semantics already in the initial classification, three new classes are added to the initial classification.

During the second iteration, the following two terms (among others) are built:

- the term $t_{2.1}$ = `(x' + y') + y` is built as the instantiation of `x` in $t_{1.1}$ with (a renamed apart variant of) $t_{1.1}$, and
- the term $t_{2.2}$ = `x + (x' + y')` as the instantiation of `y` in $t_{1.1}$ with $t_{1.1}$.

The semantics of these two terms is the same s, but it is different from the semantics of the existing equivalence classes. Thus, during this iteration (at least) the new equivalence class $ec' := \langle s, t_{2.1}, \{t_{2.1}, t_{2.2}\}, n\rangle$ is computed. Hereafter, only the representative of the class will be used for constructing new terms. Since we have chosen $t_{2.1}$ instead of $t_{2.2}$ as the representative, terms like `(x + (x' + y')) % z` will never be built. ■

Thanks to the closedness w.r.t. context of the semantics, this strategy for generating terms is *safe*. In other words, when we avoid to build a term, it is because it is not able to produce a

behavior different from the behaviors already included by the existing terms, thus we are not losing completeness.

Generation of the $=_C$ specification Since, by construction, we have avoided much redundancy thanks to the strategy used to generate the equivalence classes, we now have only to take each equivalence class with more than one term and generate equations for these terms.

Generation of $=_{RB}$ equations The third phase of the algorithm works on the former classification by first transforming each equivalence class ec by replacing the C-semantics $sem(ec)$ with $\mathcal{E}^{RB}[\![rep(ec); \mathcal{R}]\!]$ and $terms(ec)$ with the (singleton) set $\{rep(ec)\}$. After the transformation, some of the previous equivalence classes which had different semantic constructor terms may now have the same RB-semantics and then we merge them, making the union of the term sets $terms(ec)$.

Thanks to the fact that, before merging, all equivalence classes were made of just singleton term sets, we cannot generate (again) equations $t_1 =_{RB} t_2$ when an equation $t_1 =_C t_2$ had been already issued. Let us clarify this phase by an example.

Example 3.2 Assume we have a classification consisting of three equivalence classes with C-semantics s_1, s_2 and s_3 and representative terms t_{11}, t_{22} and t_{31}:

$$ec_1 = \langle s_1, t_{11}, \{t_{11}, t_{12}, t_{13}\}\rangle \qquad ec_2 = \langle s_2, t_{22}, \{t_{21}, t_{22}\}\rangle \qquad ec_3 = \langle s_3, t_{31}, \{t_{31}\}\rangle$$

We generate equations $t_{11} =_C t_{12} =_C t_{13}$ and $t_{21} =_C t_{22}$.

Now, assume that $\mathcal{E}^{RB}[\![t_{11}]\!] = w_0$ and $\mathcal{E}^{RB}[\![t_{22}]\!] = \mathcal{E}^{RB}[\![t_{31}]\!] = w_1$. Then (since t_{12}, t_{13} and t_{21} are removed) we obtain the new classification

$$ec_4 = \langle w_0,\, t_{11},\, \{t_{11}\},\, n\rangle \qquad ec_5 = \langle w_1,\, t_{22},\, \{t_{22}, t_{31}\},\, n\rangle$$

Hence, the only new equation is $t_{22} =_{RB} t_{31}$. Indeed, equation $t_{11} =_{RB} t_{12}$ is uninteresting, since we already know $t_{11} =_C t_{12}$ and equation $t_{21} =_{RB} t_{31}$ is redundant (because $t_{21} =_C t_{22}$ and $t_{22} =_{RB} t_{31}$). ■

The resulting (coarser) classification is then used to produce the $=_{RB}$ equations, as done before, by generating equations for all non-singletons term sets.

Generation of the $=_G$ equations In the last phase, we transform again the classification by replacing the RB-semantics with the G-semantics (and $terms(ec)$ with the set $\{rep(ec)\}$). Then we merge eventual equivalence classes with the same semantics and, finally, we generate $=_G$ equations for non singleton term sets.

Theorem 3.3 (Correctness) *For all equations $e_1 = e_2$ generated in the second, third and forth phase we have that $e_1 =_C e_2$, $e_1 =_{RB} e_2$ and $e_1 =_G e_2$, respectively.*

Proof. By construction of equivalence classes, in the second phase an equation $t_1 = t_2$ is generated if and only if the semantics $\mathcal{E}^C[\![t_1; \mathcal{R}]\!] = \mathcal{E}^C[\![t_2; \mathcal{R}]\!]$. Then, since $\mathcal{E}^C[\![t_1; \mathcal{R}]\!] = \mathcal{E}^C[\![t_2; \mathcal{R}]\!] \iff t_1 =_C t_2$, the first part of thesis follows immediately. Then, by the successive trasformation and reclassification, in the third phase we issue an equation $t_1 = t_2$ if and only if $\mathcal{E}^{RB}[\![t_1; \mathcal{R}]\!] = \mathcal{E}^{RB}[\![t_2; \mathcal{R}]\!]$. Then, since $\mathcal{E}^{RB}[\![t_1; \mathcal{R}]\!] = \mathcal{E}^{RB}[\![t_2; \mathcal{R}]\!] \iff t_1 =_{RB} t_2$, we have the second part of thesis. The proof of the third part of thesis, since $\mathcal{E}^G[\![t_1; \mathcal{R}]\!] = \mathcal{E}^G[\![t_2; \mathcal{R}]\!] \iff t_1 =_G t_2$, is analogous. □

Example 3.4 Let us recall (again) the program R of Example 2.1. The terms `x + y` and `y + x`, before the transformation, belong to two different equivalence classes (since their rewriting behavior is different). Anyway, after the transformation, the two classes are merged since their G-semantics is the same, namely $\big\{\{x/0,y/0\}\cdot 0,\{x/s(0),y/0\}\cdot s(0),\{x/0,y/s(0)\}\cdot s(0),\dots\big\}$. ■

We now present some examples to show how our proposal deals with non-contructor based or non-confluent TRS, plus some to show that we can infer several non-trivial equations.

Example 3.5 Consider the definition for the boolean data type with constructor terms `true` and `false` and boolean operations `and`, `or`, `not` and `imp`:

```
and(true,x) -> x                    not(true) -> false
and(false,x) -> false               not(false) -> true
or(true,x) -> true                  imp(false,x) -> true
or(false,x) -> x                    imp(true,x) -> x
```

This is a pretty standard "short-cut" definition of boolean connectives. With our method we get the following equations:

`not(or(x,y))` $=_C$ `and(not(x),not(y))`	`imp(x,y)` $=_C$ `or(not(x),y)`
`not(and(x,y))` $=_C$ `or(not(x),not(y))`	`not(not(x))` $=_G$ `x`
`and(x,and(y,z))` $=_C$ `and(and(x,y),z)`	`and(x,y)` $=_G$ `and(y,x)`

■

Example 3.6 Let us consider the following *non-constructor based* TRS implementing some operations over the naturals in Peano notation.

```
x - 0 -> x                          chk(0) -> 0
s(x) - s(y) -> x - y                chk(s(x)) -> s(x)
g(x) -> chk(x - s(x))               chk(0 - s(x)) -> err
```

The definition of `-` is a standard definition of the minus operation over naturals. The `chk` function simply returns the natural passed as argument, or returns `err` if the argument is a not defined subtraction. It is easy to see that the TRS is not constructor-based because of the presence of the `-` in the pattern of a rule. The artificial function `g`, which checks if the subtraction of a number by its successor is a natural number, is doomed to return `err`. With our classification we actually derive equation `g(x)` $=_C$ `err`. ■

Example 3.7 Let us consider the following (artificial) *non-orthogonal* TRS.

```
coin -> 0              d(x) -> g(x,x)          t(x) -> k(x,x,x)
coin -> 1              g(0,1) -> true          k(1,0,1) -> true
```

The `coin` function can return both `0` and `1`. The functions `d` and `t` call an auxiliary function, duplicating and triplicating (respectively) the variable received as argument. Functions `f` and `g` require a specific combination of `0` and `1` to return the constructor term `true`. Notice that, to reach the constructor term `true` from `d` and `t` respectively, it is necessary to use a non deterministic function able to rewrite to both `0` and `1`. Some of the inferred equations for this TRS are `t(x)` $=_C$ `d(x)` and `t(coin)` $=_C$ `g(coin,coin)` $=_C$ `k(coin,coin,coin)` $=_C$ `d(coin)`.
■

Example 3.8 Let us consider the following TRS that computes the double of numbers in Peano notation:

```
double(0) -> 0                          plus(0,y) -> y
double(s(x)) -> s(s(double(x)))         plus(s(x),y) -> s(plus(x,y))
dbl(x) -> plus(x,x)
```

Some of the inferred equations for the TRS are:

$$\texttt{dbl(dbl(double(x)))} =_G \texttt{dbl(double(dbl(x)))} \quad (3.1)$$

$$\texttt{double(x)} =_G \texttt{dbl(x)} \quad (3.2)$$

$$\texttt{dbl(dbl(x))} =_G \texttt{dbl(double(x))} =_G \texttt{double(dbl(x))} =_G \texttt{double(double(x))} \quad (3.3)$$

$$\texttt{plus(double(x),y)} =_G \texttt{plus(dbl(x),y)} \quad (3.4)$$

We can observe that all equations hold with the $=_G$ relation, and not with the $=_{RB}$ relation. This is due to the fact that the two functions `dbl` and `dbl`, even if at a first sight could seem to evaluate the same constructor terms, can behave differently. For instance, if we add to the TRS the two rules `coin -> 0` and `coin -> s(0)` we can notice that the terms `double(coin)` and `dbl(coin)` return different constructor terms. This is due to the non determinism of the `coin` function that exploits the right non-linearity of function `dbl`. While `double(coin)` evaluates to `0` and `s(s(0))`, the term `dbl(coin)` can be reduced even to `s(0)` (by `dbl(coin)` $\to$ `plus(coin,coin)` $\to^2$ `plus(0,s(0))` $\to$ `s(0)`).

This characteristic of the program is not easy to realize by just looking at the code. ■

Example 3.9 Let us consider the following TRS defining two functions over an extension of the Peano notation able to handle negative integers:

```
abs(-(x)) -> abs(x)                     f(-(-(x))) -> f(x)
abs(s(x)) -> s(x)                       f(0) -> 0
abs(0) -> 0                             f(s(x)) -> s(x)
                                        f(-(s(x))) -> 0
```

Function `abs` is a standard definition of the absolute value; function `f` returns its input if it is a positive number, and `0` if it is not. Some of the inferred equations are `f(f(x))` $=_C$ `abs(f(x))` and `f(abs(x))` $=_C$ `abs(abs(x))`. ■

Example 3.10 Let us consider the following program which implements a two-sided queue in a (non-trivial) efficient way. The queue is implemented as two lists where the first list corresponds to the first part of the queue and the second list is the second part of the queue reversed. The `inl` function adds the new element to the head of the first list, whereas the `inr` function adds the new element to the head of the second list (the last element of the queue). The `outl` (`outr`) function drops one element from the left (right) list, unless the list is empty, in which case it reverses the other list and then swaps the two lists before removal.

```
new -> Q(Nil,Nil)                       outr(Q(xs,Nil)) ->
inl(x,Q(xs,ys)) -> Q(:(x,xs),ys)                Q(Nil,tail(rev(xs)))
inr(x,Q(xs,ys)) -> Q(xs,:(x,ys))        outr(Q(xs,:(y,ys))) -> Q(xs,ys)
outl(Q(Nil,ys)) ->                      null(Q(:(x,xs),ys)) -> False
        Q(tail(rev(ys)),Nil)            null(Q(Nil,:(x,xs))) -> False
outl(Q(:(x,xs),ys)) -> Q(xs,ys)         null(Q(Nil,Nil)) -> True
```

```
tail(:(x,xs)) -> xs                    rv'(Nil,ys) -> ys
rev(xs) -> rv'(xs,Nil)                 rv'(:(x,xs),ys) -> rv'(xs,:(x,ys))
```

With our method (amongst others) we derive:

$$\texttt{null(new)} =_C \texttt{True} \tag{3.5}$$
$$\texttt{new} =_C \texttt{outl(inl(x,new))} =_C \texttt{outr(inr(x,new))} \tag{3.6}$$
$$\texttt{outl(inl(x,q))} =_C \texttt{outr(inr(x,q))} \tag{3.7}$$
$$\texttt{inr(x,inl(y,q))} =_C \texttt{inl(y,inr(x,q))} \tag{3.8}$$
$$\texttt{inl(x,outl(inl(y,q)))} =_C \texttt{outr(inl(x,inr(y,q)))} \tag{3.9}$$
$$\texttt{null(inl(x,new))} =_C \texttt{null(inr(x,new))} =_C \texttt{False} \tag{3.10}$$

We can see different kinds of non-trivial equations: eqs. (3.6) and (3.7) state that adding and removing one element produces always the same result independently from the side in which we add and remove it. Equations (3.8) and (3.9) show a sort of *restricted* commutativity between functions. ■

3.1 An effective instance of the presented method

In a semantics-based approach, one of the main problems to be tackled is effectiveness. The semantics of a program is in general infinite and thus some approximation has to be used in order to have a terminating method. To experiment on the validity of our proposal we started by using a novel (condensed) fixpoint semantics which we have developed for left-linear TRSs that is fully abstract w.r.t. $=_C$ [3]. Such semantics is defined as the fixpoint of an immediate consequences operator $\mathcal{P}[\![\mathcal{R}]\!]$. We have opted for this semantics because it has some properties which are very important from a pragmatical point of view:

- it is condensed, meaning that denotations are the smallest possible (between all those semantics which induce the same program equivalence). This is a very relevant (if not essential) feature to develop a semantic-based tool which has to *compute* the semantics.
- The semantics $\mathcal{E}^{RB}$ can be obtained directly by transforming the $\mathcal{E}^{C}$ semantics, concretely just by loosing internal structure. Therefore, no (costly) computation of $\mathcal{E}^{RB}$ is needed.

We have implemented the basic functionality of the proposed methodology in a prototype written in Haskell, TRSynth, available at http://safe-tools.dsic.upv.es/trsynth (for a detailed description see [4]). The implementation of $=_G$ equality is still ongoing work, because we are lacking of a suitable implementation of the G-semantics.

To achieve termination, the prototype computes a fixed number k of steps of $\mathcal{P}[\![\mathcal{R}]\!]$. Then, it proceeds with the classification as described in Section 3. Clearly, in presence of terms with infinite solutions, with such a rough approximation we may loose both correctness (by mistakenly equating terms which become semantically different after k iterates) and completeness (by mistakenly not equating terms which will become semantically equivalent). Nevertheless the results are encouraging. For instance TRSynth detects all $=_C$ and $=_{RB}$ which we showed in examples (except of Example 3.8 because of a bug in the computation of the semantics).

[3]The writing of articles related to the formal definition of this semantics is in progress [3].

4 Conclusions and future work

This paper discusses about the issues that arise for the automatic inference of high-level, property-oriented (algebraic) specifications because of non-confluent or non-constructor based TRS. Then, a first proposal which overcomes these issues is presented.

Our method computes a concise specification of program properties from the source code of a TRS. We hope to have convinced the reader (with all examples) that we reached our main goal, that is, to get a concise and clear specification (that is useful for the programmer in order to discover unseen properties or detect possible errors).

We have developed a prototype that implements the basic functionality of the approach. We are aware that many other attempts to guarantee termination could be used in other instancies of the presented method. Certainly, given our know-how, in the future we will experiment with abstractions obtained by abstract interpretation [5] (our novel semantics itself has been obtained as an abstract interpretation). Actually we already have an ongoing work to implement the $depth(k)$ abstract version of our semantics. In the $depth(k)$ abstraction, terms (occurring in the nodes of the semantic trees) are "cut" at depth k by replacing them with *cut variables*, distinct from program variables. Hence, for a given signature Σ, the universe of abstract semantic trees is finite (although it increases exponentially w.r.t. k). Therefore, the finite convergence of the computation of the abstract semantics is guaranteed.

References

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'02)*, pages 4–16, New York, NY, USA, 2002. Acm. 1

[2] G. Bacci, M. Comini, M. A. Feliú, and A. Villanueva. Automatic Synthesis of Specifications for First Order Curry Programs. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, pages 25–34, New York, NY, USA, 2012. ACM. 2, 3

[3] M. Comini and L. Torella. A Condensed Goal-Independent Fixpoint Semantics Modeling the Small-Step Behavior of Rewriting. Technical Report DIMI-UD/01/2013/RR, Dipartimento di Matematica e Informatica, Università di Udine, 2013. 3

[4] M. Comini and L. Torella. TRSynth: a Tool for Automatic Inference of Term Equivalence in Left-linear Term Rewriting Systems. In E. Albert and S.-C. Mu, editors, *PEPM '13, Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 67–70. Acm, 2013. 3.1

[5] 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. 4

[6] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *31st International Conference on Software Engineering (ICSE'09)*, pages 430–440, 2009. 1

[7] J. Henkel, C. Reichenbach, and A. Diwan. Discovering Documentation for Java Container Classes. *IEEE Transactions on Software Engineering*, 33(8):526–542, 2007. 1

[8] A. A. Khwaja and J. E. Urban. A property based specification formalism classification. *The Journal of Systems and Software*, 83:2344–2362, 2010. 1

[9] TeReSe, editor. *Term Rewriting Systems*. Cambridge University Press, Cambridge, UK, 2003. 1.1

[10] H. van Vliet. *Software Engineering–Principles and Practice*. John Wiley, 1993. 1

[11] J. M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):10–24, 1990. 1