Automatic Synthesis of Specifications for First Order Curry Programs

Giovanni Bacci  Marco Comini
DIMI, University of Udine (Italy)
giovanni.bacci@uniud.it, marco.comini@uniud.it

Marco A. Feliú  Alicia Villanueva
DSIC, Universitat Politècnica de València (Spain)
mfeliu@dsic.upv.es, villanue@dsic.upv.es

Abstract
This paper presents a technique to automatically infer algebraic property-oriented specifications from first-order Curry programs. Curry is a lazy functional logic language and the interaction between laziness and logical variables raises some additional difficulties with respect to other proposals for functional languages. Our technique statically infers from the source code of a Curry program a specification which consists of a set of equations relating (nested) operation calls that have the same behavior. We propose a (glass-box) semantic-based inference method which relies on a fully-abstract (condensed) semantics for achieving, to some extent, the correctness of the inferred specification, differently from other (black-box) approaches based on testing techniques.

Categories and Subject Descriptors F.3.1 [Logics and meaning of programs]: Specifying and Verifying and Reasoning about programs—Specification techniques; D.3.2 [Programming Languages]: Constraint and logic languages

General Terms Documentation, Languages, Verification

Keywords Curry, property-oriented specifications, semantic-based inference methods

1. Introduction
Specifications have been widely used for several purposes: they can be used to aid (formal) verification, validation or testing, to instrument software development, as summaries in program understanding, as documentation of programs, to discover components in libraries or services in a network context, etc. [2, 8, 11, 12, 15, 17, 19, 22]. Depending on the context and the use of specifications, they can be defined, either manually or automatically, before coding (e.g. for validation purposes), during the program coding (e.g. for testing or understanding purposes), or after the code has been written (for verification or documentation). We can find several proposals of (automatic) inference of high-level specifications (from an executable or from the source code) of a system, like [2, 8, 12, 15], which have proven to be very helpful.

In the literature, specification formalisms have been classified through some common characteristics [16]. It is frequent to distinguish between property-oriented specifications and model-oriented or functional specifications. It can be said that property-oriented specifications are at a higher description level than other kinds of specifications: they consist in an indirect definition of the system’s behavior by means of stating a set of properties, usually in the form of axioms, that the system must satisfy [20, 21]. 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 with 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.

Clearly, the task of automatically inferring program specifications is in general undecidable and, given the complexity of the problem, there exists a large number of different proposals which impose several restrictions. Many aspects vary from one solution to another: the kind of specifications that are computed (e.g., model-oriented vs. property-oriented specifications), the kind of programs considered, the correctness or completeness of the method, etc.

We can identify two mainstream approaches to perform the inference of specifications: glass-box and black-box. The glass-box approach [2, 8] 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 [8]. Therefore, the specification must be more succinct and comprehensible than the source code itself. The inferred specification can also be used to automatize the testing process of the program [8] or to verify if a given property holds [2]. The black-box approach [12, 15] 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) [12]. Although black-box approaches work without any restriction on the considered language—which is rarely the case in a glass-box approach—in general, they cannot guarantee the correctness of the results (whereas indeed semantics-based glass-box approaches can).

For this work, we took inspiration from QuickSpec [8], which is an (almost) black-box inference approach for Haskell programs [18] based on testing. QuickSpec automatically infers program specifications as sets of equations of the form \( e_1 = e_2 \), where \( e_1 \) and \( e_2 \) are generic program expressions that (should) have the same computational behavior. This approach has two properties that we like:
it is completely automatic as it needs only the program to run, plus some indications on target functions and generic values to be employed in equations, and the outcomes are very intuitive since they are expressed only in terms of the program components, so the user does not need any kind of extra knowledge to interpret the results.

However, our proposal ended up being radically different from QuickSpec:

• First, we aim to infer correct (algebraic) property-oriented specifications. To this end, instead of a testing-based approach, we propose a glass-box based approach.

• Second, we consider the functional logic language Curry [13, 14]. Curry is a multi-paradigm programming language that combines in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions) and logic programming (logical variables, partial data structures, built-in search). Due to lazy evaluation in presence of (free) logical variables, the problem of inferring specifications for this kind of languages poses several additional problems w.r.t. other paradigms. We discuss these issues in Section 2.

In the rest of the paper, we first introduce the problem of generating useful specifications for the functional logic paradigm by discussing a simple, illustrative example. In Section 3, we define our notion of specification, which is composed of equations of different kinds. Thereafter, in Section 4, we explain how the specifications are computed in detail. In Section 5 we show some examples of specifications computed by the prototype implementing the technique. Finally, Section 6 discusses the most related work and Section 7 concludes.

2. Analysis of the issues posed by the logical features of Curry

Curry is a lazy functional logic language which admits free (logical) variables in expressions and whose program rules are evaluated non-deterministically. Differently from the functional case, due to the logical features, an equation $e_1 = e_2$ can be interpreted in many different ways. We will discuss the key points of the problem by means of a (very simple) illustrative example.

The syntax of Curry is very similar to that of Haskell. Variables and function names start with a character in lower case, whereas data constructors and type names start with a letter in upper case. For a complete description of the Curry language, the interested reader can consult [14]. In this work, we assume that the reader is familiar with the syntax and basic semantic notions of Haskell.

Example 2.1 (Boolean Logic Example) Consider the definition of the boolean data type with values True and False and operations and, or, not and imp:

\[
\begin{align*}
\text{and} & \quad True \times x = x \\
\text{and} & \quad False \_ = False \\
\text{or} & \quad True \_ = True \\
\text{or} & \quad False \_ = x \\
\text{not} & \quad True = False \\
\text{not} & \quad False = True \\
\text{imp} & \quad False \_ = True \\
\text{imp} & \quad True \_ = x
\end{align*}
\]

This is a pretty standard “short-cut” definition of boolean connectives. For example, the definition of and states that whenever the first argument is equal to False, the function returns the value False, regardless of the value of the second argument. Since the language is lazy, in this case the second argument will not be evaluated.

For example, one could expect a (property-oriented) specification with equations like

\[
\begin{align*}
\text{imp} & \quad x \_ y = or \_ (not \_ x) \_ y \\
\text{not} & \quad (or \_ x \_ y) = and \_ (not \_ x) \_ (not \_ y) \\
\text{not} & \quad (and \_ x \_ y) = or \_ (not \_ x) \_ (not \_ y) \\
\text{not} & \quad (not \_ x) = x \\
\text{and} & \quad (x \_ y \_ z) = and \_ (x \_ y) \_ z \\
\text{and} & \quad x \_ y = and \_ y \_ x
\end{align*}
\]

which are well-known among the (theoretical) boolean operators. This comprises specification aids the user to learn the properties of the program. In addition, the specification can be useful to detect bugs in the program by observing both, properties (equations) that occur in the specification but were not expected, and expected equations that are missing. These equations, of the form $e_1 = e_2$, can be read as

all possible outcomes for $e_1$ are also outcomes for $e_2$, and vice versa.

In the following, we call this notion of equivalence computed result equivalence and we denote it by $\equiv_C$.

Actually, Equations (2.1), (2.2), (2.3), (2.5) and (2.6) are literally valid in this sense since, in Curry, free variables are admitted in expressions, and the mentioned equations are valid as they are. This is quite different from the pure functional case where equations have to be interpreted as properties that hold for any ground instance of the variables occurring in the equation.

On the contrary, Equation (2.4) is not literally valid. Let us first introduce the notation for evaluations. The expression $\{x/True\} \cdot True$ denotes that the normal form True (at the right of the · symbol) has been reached with computed answer substitution $\{x/True\}$ (at the left of the · symbol). Now we are ready to discuss Equation (2.4). The goal on the left hand side of the equation $not \_ (not \_ x)$ evaluates to two normal forms: $\{x/True\} \cdot True$ and $\{x/False\} \cdot False$, whereas the right hand side of the equation $x$ evaluates just to $\{\} \cdot x$. Note however that any ground instance of the two goals evaluates to the same results, namely both True and not (not False) evaluate to $\{\} \cdot True$, and both False and not (not False) evaluate to $\{\} \cdot False$.

This fact motivates the use of an additional notion of equivalence, called ground equivalence, which can be helpful for the user since the equations that hold under this equivalence represent, in general, interesting properties of the program. We denote it by $\equiv_G$.

This notion coincides with the (only possible) equivalence notion used in the pure functional paradigm: two terms are ground equivalent if, for all ground instances, the outcomes of both terms coincide.

Because of the presence of logical variables, there is another very relevant difference w.r.t. the pure functional case, concerned with contextual equivalence: given a valid equation $e_1 = e_2$, is it true that, for any context $C$, the equation $C[e_1] = C[e_2]$ still holds? Curry is not referentially transparent w.r.t. its operational behavior, i.e., an expression can produce different computed values.

2 In this section, our main goal is to give the intuition of the specification computed by our approach, thus we show just a subset of the equations satisfied by the program.

3 The concept of referential transparency of a language can be stated in terms of a formal semantics as: the semantics equivalence of two expres-
In this section, we formally present all the kinds of term equivalence. Since the number of needed terms to be evaluated grows exponentially w.r.t. the depth of nestings, the addition of a further outer context would dramatically alter the performance. Moreover, if we try to mitigate this problem by reducing the number of terms used within any context \(C[\cdot]\), the semantics is not.

**Example 2.2**

Given a program with the following rules

\[
\begin{align*}
g \ x &= C \ (h \ x) \\
g' \ A &= C \ A \\
h \ A &= A \\
f (C \ x) \ B &= B
\end{align*}
\]

the expressions \(g \ x\) and \(g' \ x\) compute the same result, namely \(\{x/A\} \cdot C\ A\). However, the expression \(f \ (g \ x) \ x\) computes one result, namely \(\{x/B\} \cdot B\), while expression \(f' \ (g' \ x) \ x\) computes none.

Thus, in the Curry case, becomes mandatory to additionally ask in the equivalence notion of (2.7) that the outcomes must be equal also when the two terms are embedded within any context. We call this equivalence contextual equivalence and we denote it by \(\equiv\). Actually, Equations (2.1), (2.2), (2.3) and (2.5) are valid w.r.t. this equivalence notion.

We can see that \(\equiv\) is (obviously) stronger than \(\equiv_{\text{cr}}\), which is in turn stronger than \(=\). As a conclusion, for our example we would get the following (partial) specification.

\[
\begin{align*}
\text{imp} \ x \ y &= C \text{ or } (\text{not} \ x) \ y \\
\text{not} \ (\text{or} \ x \ y) &= C \text{ and } (\text{not} \ x) \ (\text{not} \ y) \\
\text{not} \ (\text{and} \ x \ y) &= C \text{ or } (\text{not} \ x) \ (\text{not} \ y) \\
\text{and} \ x \ (\text{and} \ y \ z) &= C \text{ and } (\text{and} \ x \ y) \ z \\
\text{not} \ (\text{not} \ x) &= C \ x \\
\text{and} \ x \ y &= C \text{ and } y \ x
\end{align*}
\]

This example has shown, first, the kind of property-oriented specifications that we want to compute from the program, and second, the need to consider different kinds of equivalences between terms in order to get a useful specification. It is worth noticing that adopting only a notion of equivalence based on the referentially transparent semantics (the \(=\_C\) equivalence) can be too restrictive: we may lose important properties. However, by using the just the weaker notions we cannot know if two equivalent expressions are also equivalent within any context.

The need of determining \(\equiv\) equalities can explain the reason because we believe that, in the case of Curry, the use of a semantics-based approach can be more suited than testing-based approaches. In a test-based approach expressions should have to be computed in some outer context in order to establish their equivalence notion. Since the number of needed terms to be evaluated grows exponentially w.r.t. the depth of nestings, the addition of a further outer context would dramatically alter the performance. Moreover, if we try to mitigate this problem by reducing the number of terms/tests to be checked, the quality of the produced equations degrades sensibly. On the contrary, a semantics-based approach, based on a fully abstract semantics, achieves the \(\equiv\) equivalence by construction.

### 3. Formalization of equivalence notions

In this section, we formally present all the kinds of term equivalence notions that are used to compute equations of the specifications \(e, e'\) implies the semantics equivalence of \(e\) and \(e'\) when used within any context \(C[\cdot]\). Namely, \(\forall e, e', C. [e] = [e'] \implies [C[e]] = [C[e']]\).

\[\text{as we will show later, our technique computes a complete specification for a specific size of terms in equations.}\]

We need first to introduce some basic formal notions that are used in the rest of the paper.

We say that a first order Curry program is a set of rules \(P\) built over a signature \(\Sigma\) which is partitioned in \(C\), the constructor symbols, and \(D\), the defined symbols. \(V\) denotes a (fixed) countably infinite set of variables and \(\mathcal{T}(\Sigma, V)\) denotes the terms built over signature \(\Sigma\) and variables \(V\). A fresh variable is a variable that appears nowhere else.

**The semantics.** We evaluate first order Curry programs on the condensed, goal-independent semantics recently defined in [3, 4] for functional logic programs. We preferred this semantics instead of the established (small-step) operational and I/O semantics [1, 4] because they do not fulfill referential transparency, whereas the former does. This fact makes the (more elaborated) semantics of [3, 4] an appropriate base semantics for computing specifications w.r.t. \(\equiv\). Moreover, this semantics has another property which is very important from a pragmatical point of view: it is condensed, meaning that denotations are the smallest possible (between all those semantics which are fully abstract). This is an almost essential feature in order to develop a semantic-based tool which has to compute the semantics. In particular, with this semantics it is reasonable to compute a finite number of iterations of the program’s denotation itself, while the computation of the other mentioned semantics is not.

The denotation \(\mathcal{F}[P]\) of a program \(P\) is the least fixed-point of an immediate consequence operator \(\mathcal{P}[P]\). This operator is based on a term evaluation function \(\mathcal{E}[\cdot]\) which, for any term \(t \in \mathcal{T}(\Sigma, V)\), gives the semantics of \(t\) as \(\mathcal{E}[t]_{\mathcal{F}[P]}\). Intuitively, the evaluation \(\mathcal{E}[t]_{\mathcal{F}[P]}\) computes a tree-like structure collecting the “relevant history” of the computation of all computed results of \(t\), abstracting from function calls and focusing only on the way in which the result is built. In particular, every leaf of the tree represents a normal form of the initial term. Nodes are pairs of the form \(\sigma \cdot s\), where \(\sigma\) is a substitution (binding variables of the initial expression with linear constructor terms), and \(s\) is a partially computed value, that is, a term in \(\mathcal{T}(C, V \cup V_o)\) which may contain special variables \(o_0, o_1, \ldots \in V_o\) (a set disjoint from \(V\)) indicating an unevaluated subterm. Leaves with no occurrences of special variables are computed results. We denote by \(\mathcal{cr}(T)\) the set of computed results of the semantic tree \(T\).

Full-abstraction w.r.t. the behavior and referential transparency of the semantics are proven in [3]. Thus, it holds that \(\mathcal{E}[\mathcal{E}[t]_{\mathcal{F}[P]}] = \mathcal{E}[t]_{\mathcal{F}[P]}\) corresponds to the set of computed outcomes of \(t\) using \(P\). Moreover, given two terms \(e, e'\), and a generic context \(C[\cdot]\), it holds that if \(\mathcal{E}[e]_{\mathcal{F}[P]} = \mathcal{E}[e']_{\mathcal{F}[P]}\), then \(\mathcal{E}[C[e]]_{\mathcal{F}[P]} = \mathcal{E}[C[e']]_{\mathcal{F}[P]}\).

The following states the correctness of the semantics.

**Theorem 3.1 ([3])** Let \(P\) be a first-order Curry program and \(t\) be a term in \(\mathcal{T}(\Sigma, V)\). Then \(\mathcal{cr}(\mathcal{E}[t]_{\mathcal{F}[P]})\) corresponds to the set of computed outcomes of \(t\) using \(P\).

Moreover, \(\mathcal{E}[t]_{\mathcal{F}[P]}\) fulfills referential transparency:

**Theorem 3.2** Let \(P\) be a first-order Curry program, \(e, e'\) terms in \(\mathcal{T}(\Sigma, V)\), and \(C[\cdot]\) be a context. If \(\mathcal{E}[e]_{\mathcal{F}[P]} = \mathcal{E}[e']_{\mathcal{F}[P]}\), then \(\mathcal{E}[C[e]]_{\mathcal{F}[P]} = \mathcal{E}[C[e']]_{\mathcal{F}[P]}\).

**Example 3.3** (Example 2.2 continued)

The computed semantics for the program \(P\) in Example 2.2 is the following:

\[
\mathcal{F}[P] = \left\{ \begin{array}{ll}
g \ x \mapsto e \cdot \varrho \cdot \varphi & g \ x \mapsto e \cdot \varrho \cdot \varphi \cdot \{x/A\} \cdot C \ A \\
g' \ x \mapsto e' \cdot \varrho & g' \ x \mapsto e' \cdot \varrho \cdot \{x/A\} \cdot C \ A \\
h \ x \mapsto e \cdot \varrho & h \ x \mapsto e \cdot \varrho \cdot \{x/C \ x', y/B\} \cdot B \\
f \ x \ y \mapsto e \cdot \varrho & f \ x \ y \mapsto e \cdot \varrho \cdot \{x/C \ x', y/B\} \cdot B 
\end{array} \right.
\]
The semantics of a program $P$ is a family of semantic trees indexed by most general expressions (a function symbol applied to distinct variables). Edges in the semantic tree are labeled with the special variable that is instantiated with an expression (that may contain another special variable). Below we show the evaluation of the two expressions that lead to different computed results in the example:

$$E[f \ (g \ x)]_{\mathcal{F}[P]} = \epsilon \cdot g \ \xrightarrow{\epsilon} \ x/B \cdot B$$

$$E[f \ (g' \ x)]_{\mathcal{F}[P]} = \epsilon \cdot g$$

Note that $\epsilon \cdot g$ is not a computed result due to the occurrence of the $g$ variable.

Even if this semantics is condensed, the trees in denotations can be infinite both in depth and in width, as we show with the following example.

**Example 3.4**

Consider the classical append function:

```plaintext
append \[\] y = y
append \((x:xs)\) y = x:(append xs y)
```

The semantics of the function `append` in $\mathcal{F}[P]$ is:

$$\begin{align*}
\text{append} & \ y & \equiv_0 \{x/\} \cdot y \\
\text{append} & \ (x:xs) \ y & \equiv_0 \{x/\} \cdot (x/\{x/\}) \cdot (x_1:y)
\end{align*}$$

The dotted triangle in the figure denotes that the semantics has not been completely computed.

In order to deal with infinite trees, we need to use an approximated (abstract) semantics (obtained by abstract interpretation [9, 10]) for the computation of our inference method (in Section 4). A discussion about effectiveness and precision regarding this issue is given in Section 4.1.

**The specification.** Formally, an algebraic specification $S$ is a set of (sequences of) equations of the form $t_1 =^K t_2 =^K \ldots =^K t_n$, with $K \in \{C, CR, G\}$ and $t_1, t_2, \ldots, t_n \in \mathcal{T}(\Sigma, V)$. $K$ distinguishes the kinds of computational equalities that we previously informally discussed, which we now present formally.

**Contextual Equivalence $=_C$.** This equivalence states that two terms $t_1$ and $t_2$ are equivalent if $C[t_1]$ and $C[t_2]$ have the same behavior for any context $C[\cdot]$. This is the most difficult equivalence to be established by testing approaches. However, by using the semantics $\mathcal{F}[P]$ it is really easy because the semantics is fully abstract w.r.t. the contextual program behavior equivalence [3]. Therefore, two terms $t_1$ and $t_2$ are related by the contextual relation $=_C$ if and only if their semantics coincide, namely

$$t_1 =_C t_2 \iff \mathcal{E}[t_1]_{\mathcal{F}[P]} = \mathcal{E}[t_2]_{\mathcal{F}[P]}$$

Intuitively, due to the definition of this semantics, this means that all the ways in which these two terms reach their normal forms coincide. Note that $=_C$ does not capture termination properties, which is out of our current scope. However, thanks to the abstraction of the semantics, the inference technique that we are now proposing can work even if we have a non-terminating function, a situation in which black-box approaches cannot work at all.

**Computed-result equivalence $=_\text{cr}$.** This notion of equivalence states that two terms are equivalent when the outcomes of their evaluation are the same. Therefore, the computed-result equivalence abstracts from the way in which the results evolve during computation.

It is important to note that we can determine $=_\text{cr}$ just by collecting the leaves of $\mathcal{E}[e]_{\mathcal{F}[P]}$. This means that if we define a function $\text{cr}$ that, given a semantic tree computed by the evaluation function, collects the leaves of the tree, then it holds that

$$t_1 =_\text{cr} t_2 \iff \text{cr}(\mathcal{E}[t_1]_{\mathcal{F}[P]}) = \text{cr}(\mathcal{E}[t_2]_{\mathcal{F}[P]})$$

The $=_{\text{cr}}$ equivalence is coarser than $=_C (=_C \subseteq =_{\text{cr}})$ as shown by Example 2.2.

**Ground Equivalence $=_G$.** This equivalence states that two terms are equivalent if all their possible ground instances have the same outcomes. This equivalence can be obtained by generating all ground instances of the leaves of $\mathcal{E}[e]_{\mathcal{F}[P]}$. We will discuss on possible effective implementations of this notion further ahead.

Note that the ground equivalence $=_G$ is the only possible notion in the pure functional paradigm. This fact allows one to have an intuition of the reason why the problem of specification synthesis is more complex in the functional logic paradigm.

To summarize, by construction, we have that $=_C \subseteq =_{\text{cr}} \subseteq =_G$ and only $=_C$ is referentially transparent (i.e., a congruence w.r.t. contextual embedding).

### 4. Deriving Specifications from Programs

Now we are ready to describe the process of inferring specifications. The input of the process consists of the Curry program to be analyzed and two additional parameters: a relevant API, denoted $\Sigma'$, and a maximum term size, $\text{max.size}$. 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. 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 like [8], the computed specification is complete up to terms of size $\text{max.size}$, 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 $\text{max.size}$.

The inference process consists of three phases, as depicted in Figure 1. First, (an approximation of) the semantics of the input program is computed. Second, a partition of terms, formed with functions from the relevant API of size less or equal to the provided maximum size is computed. In our implementation, the size of a term is determined by its depth; however, the inference process is parametric w.r.t. the size function, thus other notions for size are allowed (e.g., number of parameters, length, etc.). Each equivalence class of the partition contains terms that are equivalent w.r.t. the contextual equivalence $=_C$ defined in Section 3. Finally, the equations of the specification are generated: first, the equations of the contextual partition are computed, and then, the equations corresponding to the other two notions of equivalence are computed by transforming the semantics.

In the following, we explain in detail the phases of the computation process by referring to the pseudo-code given in Algorithm 1. For the sake of comprehension, we present an untyped version of the algorithm. The actual one is a straightforward modification conformant w.r.t. types.
Algorithm 1 Inference of an algebraic specification

Require: Program $P$;
Program’s relevant API $\Sigma'$;
Maximum term size $\text{max\_size}$

1. Compute $F[P]$; the (abstract) semantics of $P$
2. $\text{part} \leftarrow \text{initial\_part}(F[P])$
3. repeat
4. $\text{part}' \leftarrow \text{part}$
5. for all $f/n \in \Sigma'$ do
6. for all $ec_1, \ldots, ec_n \in \text{part}$ such that at least one $ec_i$ has been introduced in the previous iteration do
7. $t \leftarrow \text{f}(\text{rep}(ec_1), \ldots, \text{rep}(ec_n))$ where the $\text{rep}(ec_i)$ are renamed apart
8. if $t \notin \text{part}$ and $\text{size}(t) \leq \text{max\_size}$ then
9. $s \leftarrow \mathcal{E}[t]_F[p]$: Compute the (abstract) semantics of term $t$
10. $\text{add\_to\_partition}(t, s, \text{part}')$
11. end if
12. end for
13. end for
14. until $\text{part}' = \text{part}$
15. specification $\leftarrow \emptyset$
16. $\text{add\_equations}(\text{specification, part})$
17. for all $\text{kind} \in [\text{CR}, G]$ do
18. $\text{part} \leftarrow \text{transform\_semantics}(\text{kind, part})$
19. $\text{add\_equations}(\text{specification, part})$
20. end for
21. return specification

Computation of the abstract semantics (and initial classification).
The first phase of the algorithm, Lines 1 to 2 (in Algorithm 1), is the computation of the initial classification that is needed to compute the classification w.r.t. $\equiv_C$. It is based on the computation of an approximation of the semantics of the program (abstract semantics).

Terms are classified by their semantics into a data structure, which we call partition, consisting of a set of equivalence classes $(ec)$ formed by
- $\text{sem}(ec)$: the semantics of (all) the terms in that class
- $\text{rep}(ec)$: the representative term of the class, which is defined as the smallest term in the class (w.r.t. the function size), and
- $\text{terms}(ec)$: the set of terms belonging to that equivalence class.

The representative term is used in order to avoid much redundancy in the generation of equations. The generation process is iterative, thus we generate first equations of smaller size, and then we increment the size until the size limit is reached. Instead of using every term of an equivalence class to build new terms of greater size, we just use the representative term.

With the program’s semantics, the initial_part function builds the initial classification which contains:
- one class for a free (logical) variable $(\mathcal{E}[x]_F[p_1], x, \{x\})$,
- the classes for any built-in or user-defined constructor.

During the definition of the initial classification, it might occur that two terms, for instance $t_1 := f(x_1, \ldots, x_n)$ and $t_2 := g(y_1, \ldots, y_n)$, have the same semantics. If this happens, we don’t generate two different classes, one for each term, but the second generated term is added to the class of the first one. For this particular example, we would have $(\mathcal{E}[t_1]_F[p_2], t_1, \{t_1, t_2\})$.

Generation of $\equiv_C$ classification. The second phase of the algorithm, Lines 3 to 14, is the (iterative) computation of the classification of terms w.r.t. $\equiv_C$. As mentioned before, this classification is also the basis for the generation of the other kinds of equivalence classes.

We iteratively select all symbols $f/n$ of the relevant API $\Sigma'$ (Line 5) and $n$ equivalence classes $ec_1, \ldots, ec_n$ from the current partition (Line 6) such that at least one $ec_i$ was newly produced in the previous iteration. We build the term $t := f(t_1, \ldots, t_n)$, where each $t_i$ is the representative term of $ec_i$, i.e., $t_i = \text{rep}(ec_i)$. In this way, by construction, the term $t$ has surely not been considered yet; then, we compute the semantics $s = \mathcal{E}[t]_F[p_1]$ and update the current partition $\text{part}'$ by using the auxiliary method add_to_partition($t, s, \text{part}'$) (Lines 7 to 11). Here, the compositionality of the semantics makes possible to compute the semantics of terms (Line 9) efficaciously: since the semantics $s_i = \text{sem}(ec_i)$ for each term $t_i$ is already stored in $ec_i$, then the computation of the semantics of $t$ can be done in an efficient way just by nesting the semantics $s_i$ into the semantics of $f(x_1, \ldots, x_n)$. This semantics nesting operation is the core of the $\mathcal{E}$ operation.

The add_to_partition($t, s, \text{part}'$) function looks for an equivalent class $ec$ in the current classification $\text{part}$ whose semantics coincides with $s$. If it is found, then the term $t$ is added to the set of terms in $ec$. Otherwise, a new equivalence class $\langle s, t, \{t\} \rangle$ is created.

If the partition suffers any modification during the current iteration (i.e., any term is added to the partition), then the algorithm iterates. This phase terminates eventually because at each iteration we consider, by construction, terms which are different from those already existing in the partition and whose size is strictly greater than the size of its subterms (but the size is bounded by $\text{max\_size}$).

The following example illustrates how the iterative process works:

5 The typed version of the inference method uses one variable for each type.
6 Following the standard notation $f/n$ denotes a function $f$ of arity $n$.
7 The interested reader can see [3] for the technical details about the semantic operators.
**Example 4.1**

Let us recall the program of Example 2.1 and choose as relevant API the functions `and`, `or`, and `not`. The following are the terms considered during the first iteration:

\[ t_{1,1} := \neg x \]
\[ t_{1,2} := \text{and} \ x \ y \]
\[ t_{1,3} := \text{or} \ x \ y \]

Since the semantics of all these terms are different, three new classes are added to the initial partition. Thus, the partition at the end of the first iteration consists of four equivalence classes: the three corresponding to terms \( t_{1,1} \), \( t_{1,2} \) and \( t_{1,3} \) and the equivalence class for the boolean free variable.

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

\[ t_{2,1} := \text{and} \ (\neg x) \ (\neg x') \]
\[ t_{2,2} := \text{not} \ (\text{or} \ x \ y) \]

More specifically, the term \( t_{2,1} \) is built as the instantiation of \( t_{1,2} \) with \( t_{1,1} \) (in both arguments), and the term \( t_{2,2} \) is the instantiation of \( t_{1,1} \) with \( t_{1,3} \). The semantics of these two terms are the same, but it is different from the semantics of the existing equivalence classes. Therefore, during this iteration (at least) a new class \( ec_1 \) for this new semantics is added, having as representative the term \( t_{2,2} \) (i.e., \( \text{rep}(ec_1) = t_{2,2} \)).

From this point on, only the representative of the class will be used for constructing new terms. This means that terms like \( \text{not} \ (\text{and} \ (\neg x) \ (\neg x')) \), which is the instantiation of \( t_{1,1} \) with \( t_{1,2} \), will never be built since only \( t_{2,1} \) can be used.

We recall here that, 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.

Although the overall strategy has been organized in order to avoid much redundancy in equations, there is one additional issue that may introduce a little redundancy. In particular, it might occur that we generate and classify a term which introduces a property that can be deduced from other equations.

Let us discuss this with an artificial example that uses the terms in Example 4.1.

**Example 4.2**

The two terms \( t_{2,1} \) and \( t_{2,2} \) belong to the same equivalence class, \( t_{2,1}, t_{2,2} \in \text{terms}(ec_1) \), which means that, as we describe below, the equation and \( (\neg x) \ (\neg y) =_C \text{not} \ (\text{or} \ x \ y) \) can be generated.

Now, assume that there exists a second equivalence class, \( ec_2 \), that includes the terms \( \text{not} \ (\neg x) \) and \( x \), thus allowing the generation of the equation \( \text{not} \ (\neg x) =_C x \).

Let us move to the following iteration and assume that \( t_{2,1} \) is the representative of \( ec_1 \). Then, one of the built terms is \( \text{not} \ (\text{and} \ (\neg x) \ (\neg y)) \), which is the instantiation of \( \text{not} \ x \) with \( t_{2,1} \). Let us suppose that the semantics of this term is the same as that of the term \( \text{or} \ x \ y \), thus it is added to its equivalence class. This implies that the corresponding equation \( \text{not} \ (\text{and} \ (\neg x) \ (\neg y)) =_C \text{or} \ x \ y \) will be generated. However, this equation is redundant because it can be deduced from the other (smaller) two.

These redundant equations are not common. In fact, the example above is not a real one since in our example the generated equation is and \( (\neg x) \ (\neg y) =_C \text{not} \ (\text{or} \ x \ y) \). Moreover, as we will illustrate later, the eventual presence of these redundant equations does not propagate to other equations.

**Generation of the specification**

The third phase of the algorithm (Lines 15 to 20) constructs the specification for the provided Curry program. First, Line 16 computes the \( =_C \) equations from the current partition. Since we have avoided much redundancy thanks to the strategy used to generate the equivalence classes, the add_equations function needs only to take each equivalence class with more than one term and generate equations for these terms.

This function generates also a side effect on the equivalence classes that is needed in the successive steps. Namely, it modifies the third component of the classes so that it replaces the (non-singleton) set of terms with a singleton set containing just the representative term.

Then, Lines 17 to 20 compute the equations corresponding to the other equivalence notions defined in Section 3. Let us explain in detail the case for the computed result equations (kind \( CR \)). As already noted, from the (tree) semantics \( T \) in the equivalence classes computed during the second phase of the algorithm, it is possible to construct (by losing the tree internal structure and collecting just the computed result leaves \( cr(T) \)) the semantics that models the computed result behavior. Therefore, we apply this transformation to the semantic values of each equivalence class. After the transformation, some of the equivalence classes which had different semantic values may now collapse into the same class. This transformation and reclassification is performed by the transform_semantics function. The resulting (coarser) partition is then used to produce the \( =_{CR} \) equations by an application of add_equations.

Thanks to the fact that add_equations ends with a partition made of just singleton term sets, we cannot generate (again) equations \( t_1 =_{CR} t_2 \) when an equation \( t_1 =_C t_2 \) had been already issued.

Let us clarify this phase by an example.

**Example 4.3**

Assume we have a partition consisting of three equivalence classes with semantics \( s_1, s_2 \) and \( s_3 \) and representative terms \( t_{11}, t_{22} \) and \( t_{31} \):

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

The add_equations procedure generates the equations

\[ \{t_{11} =_C t_{12} =_C t_{13}, \]
\[ t_{21} =_C t_{22}\]}

and, as side effect, the partition becomes

\[ ec_1 = \langle s_1, t_{11}, \{t_{11}\} \rangle \]
\[ ec_2 = \langle s_2, t_{22}, \{t_{22}\} \rangle \]
\[ ec_3 = \langle s_3, t_{31}, \{t_{31}\} \rangle \]

Now, assume that \( cr(s_1) = x_0 \) and \( cr(s_2) = cr(s_3) = x_1 \). Then, after applying transform_semantics, we obtain the new partition

\[ ec_4 = \langle x_0, t_{11}, \{t_{11}\} \rangle \]
\[ ec_5 = \langle x_1, t_{22}, \{t_{22}, t_{31}\} \rangle \]

Hence, the only new equation is \( t_{22} =_{CR} t_{31} \). Indeed, equation \( t_{13} =_{CR} t_{12} \) is uninteresting, since we already know \( t_{11} =_C t_{12} \) and equation \( t_{21} =_{CR} t_{31} \) is redundant (because \( t_{21} =_C t_{22} \) and \( t_{22} =_{CR} t_{31} \)).

In summary, if \( t_1 =_C t_2 \) holds, then \( t_1 =_{(CR,C)} t_2 \) are not present in the specification.

The same strategy can be used to generate also the \( =_G \) kind of equations. Conceptually, this could be done with a semantic
transformation equations. Conceptually, this could be done with a semantic transformation which replaces each free variable in all computed result with all its ground instances. In practice, we can use a completely dual approach, were we use a variable to represent all its possible ground instances. The transformation which corresponds to this representation

- retains only the most general instances of the original semantics (removing computed results which are instances of others) and,
- replaces a set of computed results \(R\) with its common anti-instance \(r\) when appropriate. This happens when the set of all ground instances of \(r\) is the same as that of the union of all ground instances of all elements in \(R\). This can be implemented by checking if we have a set with all the constructors of a given type (applied to free variables), then, we replace the set of constructors by a free variable and then we repeat the process until we reach a fix point.

In this way the semantics are transformed by removing further internal structure and again classes may collapse and new equations (w.r.t. \(\alpha\)) are generated.

### 4.1 Effectivity and efficiency considerations

In a semantic-based approach, one of the main problems to be tackled is effectiveness. The semantics of a program is in general infinite and thus we use abstract interpretation [9] in order to have a terminating method. More specifically, in this work we use a correct abstraction of the semantics of \([3, 5]\) over the \(\text{depth}(k)\) abstract domain. In the \(\text{depth}(k)\) abstraction, terms (occurring in the nodes of the semantic trees) are “cut” at depth \(k\) by replacing them with \textit{cut variables}, distinct from program variables. Hence, for a given signature \(\Sigma\), 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.

The presence of cut variables in the nodes of the abstract semantics denotes that the (partial) computed result has been abstracted. However, if no cut variable occurs in a node, we know that it coincides with a node in the concrete semantics. Thanks to this structure, \(\text{depth}(k)\) semantics is technically an over approximation of the semantics, but simultaneously it can be very precise (concrete) when computed results show up without “cuts”.

Therefore, equations coming from equivalence classes whose \(\text{depth}(k)\) semantics does not contain cut variables are correct equations, while for the others we do not know (yet). If we use a bigger \(k\), the latter can definitively become valid or not. Thus, equations involving approximation are equations that \textit{have not been falsified up to that point}, analogously to what happens in the testing-based approach. We call these equations \textit{unfalsified}, analogously to what happens in the testing-based approach. We call these equations \textit{unfalsified}, analogously to what happens in the

\[\text{null \(Q\)} [] \text{[]} = \text{True} \]

\[\text{null \(Q\)} (_{-:}) \text{[]} = \text{False} \]

\[\text{null \(Q\)} [] (_{-:}) = \text{False} \]

\[\text{eq} \text{Q} (\text{Q} \text{xs} \text{ys}) (\text{Q} \text{xs}’ \text{ys}’) = (\text{xs}’\text{++} \text{reverse} \text{ys}) =:= (\text{xs}’\text{++} \text{reverse} \text{ys}) \]

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 \text{inl} function adds the new element to the head of the first list, whereas the \text{inr} function adds the new element to the head of the second list (the last element of the queue). The \text{outl} (\text{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. If we include all the functions in the API and by assuming \(k \geq 3\) for the abstraction, an extract of the inferred specification for this program is the following one:

\[\text{null new} =_C \text{True} \quad (5.1)\]

\[\text{new} =_C \text{outl} (\text{inl} x \text{new}) =_C \quad (5.2)\]

\[\text{null} =_C \text{null} (\text{inr} x \text{new}) =_C \quad (5.3)\]

\[\text{outl} (\text{inl} x \text{q}) =_C \text{outl} (\text{inr} x \text{q}) =_C \quad (5.4)\]

\[\text{null} =_C \text{null} (\text{inl} x \text{q}) =_C \quad (5.5)\]

\[\text{null} (\text{inl} x \text{new}) =_C \text{null} (\text{inr} x \text{new}) =_C \quad (5.6)\]

\[\text{eq} \text{Q} (\text{inl} x \text{new}) \text{y} =_C \text{equiv} \quad (5.7)\]

\[\text{eq} \text{Q} (\text{inr} x \text{new}) \text{y} =_C \text{equiv} \quad (5.8)\]

We can see different kinds of equations in the specifications. The asymmetry in the definition of the queue makes that Equation (5.6) holds only for ground instances. Moreover, the semantics for terms in Equations (5.6) and (5.7) is abstracted (in fact, the semantics tree is infinite for these cases). Equations (5.2), (5.3) and (5.4) state that adding and removing one element produces always the same result independently from the side in which we add and remove it. Equations (5.5), (5.6), (5.7) and (5.8) show a sort of \textit{restricted} commutativity between functions. Finally, Equa-
tion (5.10) shows that, w.r.t. the user defined predicate eqQ, that identifies queues which contain the same elements, \( \text{inl} \times \text{new} \) is equivalent to \( \text{inl} \times \text{new} \), although the internal structure of the queue differs.

In Section 4, we have shown how we avoid much redundancy by using a single representative for each equivalence class. However, there is a situation in which equations may show some redundancy. The Queue example allows us to better illustrate this fact:

**Example 5.1**

Assume we are computing the specification for the Queue example with relevant API \( \Sigma^\prime = \{\text{inl}, \text{outl}\} \) following our method. The initial partition computes four equivalence classes: one for each of the two terms from the API and one for each of the two free variables \( q \) and \( x \) of type \( \text{Queue} \ a \) and \( a \), respectively.

\[
\begin{align*}
\alpha_{c1.1} &= \langle s_{1.1}, \text{inl} \times q, \{\text{inl} \times q\} \rangle \\
\alpha_{c1.2} &= \langle s_{1.2}, \text{outl} \times q, \{\text{outl} \times q\} \rangle \\
\alpha_{c1.3} &= \langle s_{1.3}, q, \{q\} \rangle \\
\alpha_{c1.4} &= \langle s_{1.4}, x, \{x\} \rangle
\end{align*}
\]

During the second iteration, the term \( \text{inl} \times (\text{outl} \times q) \) is built as the instantiation of the representative of \( \alpha_{c1.1} \) with the representative of \( \alpha_{c1.2} \). This term adds a new equivalence class to the partition:

\[
\alpha_{c2.1} = \langle s_{2.1}, \text{inl} \times (\text{outl} \times q), \{\text{inl} \times (\text{outl} \times q)\} \rangle
\]

During the same iteration, also the term \( \text{outl} \times (\text{inl} \times q) \) is built. For the sake of this discussion, we assume that it has the same semantics of \( q \):

\[
\alpha_{c1.3} = \langle s_{1.3}, q, \{q, \text{outl} \times (\text{inl} \times q)\} \rangle
\]

This means that the equation \( \text{outl} \times (\text{inl} \times q) \) is generated from \( \alpha_{c1.3} \). Then, in the third iteration, the term \( \text{outl} \times (\text{inl} \times (\text{outl} \times q)) \) is built as the instance of the representative of \( \alpha_{c1.2} \) with the representative of \( \alpha_{c2.1} \). The semantics of this new term coincides with that of \( \alpha_{c1.3} \), thus \( \alpha_{c1.2} \) is updated:

\[
\alpha_{c1.2} = \langle s_{1.2}, \text{outl} \times q, \{\text{outl} \times q, \text{outl} \times (\text{inl} \times (\text{outl} \times q))\} \rangle
\]

As a consequence, the equation \( \text{outl} \times (\text{inl} \times q) \) is redundant because it is an instance of the equation already generated from \( \alpha_{c1.3} \). \( \text{outl} \times (\text{inl} \times q) \) is generated. However, this equation is redundant because it is an instance of the equation already generated from \( \alpha_{c1.3} \). \( \text{outl} \times (\text{inl} \times q) =_{CR} q \). We recall that we have used the representatives for building the term, thus we cannot avoid this kind of redundancy with the strategy of just using the representatives.

Nevertheless, as we have already mentioned, these redundant equations are not common. The example above is not a real one since in our example the generated equation is \( q =_{CR} \text{outl} \times (\text{inl} \times q) \). Moreover, the eventual presence of these redundant equations does not propagate to other equations since, when these (greater) terms are introduced, they are not used for building other terms, but only the representative of the class.

It is worth noticing that the unfalsified equations for the Queue example (Equations (5.6) and (5.7) above), represent properties that actually hold for the program. However, it might occur that unfalsified equations correspond to false properties of the program, as the following example shows.

Consider the following program that computes the double of numbers in Peano notation:

\[
\begin{align*}
data \text{Nat} &= \text{Z} \mid \text{S Nat} \\
\text{double}, \text{double'} : \text{Nat} \to \text{Nat} \\
\text{double} \ Z &= \text{Z} \\
\text{double} \ (\text{S} \ x) &= \text{S} \ (\text{S} \ (\text{double} \ x))
\end{align*}
\]

\[
\begin{align*}
double' \ x &= \text{plus} \ x \ x \\
\text{plus} : : \text{Nat} \to \text{Nat} \to \text{Nat} \\
\text{plus} \ Z \ y &= y \\
\text{plus} \ (\text{S} \ x) \ y &= \text{S} \ (\text{plus} \ x \ y)
\end{align*}
\]

Some of the inferred equations for the program are:

\[
\begin{align*}
double' (\text{double'} (\text{double} \ x)) &=_{CR} ^{\alpha} \text{double'} (\text{double'} (\text{double} \ x)) \quad (5.11) \\
\text{double} \ x &=_{CR} ^{\alpha} \text{double'} \ x \quad (5.12) \\
\text{double'} (\text{double'} \ x) &=_{CR} ^{\alpha} \text{double'} (\text{double} \ x) \quad (5.13) \\
\text{double} (\text{double} \ x) &=_{CR} ^{\alpha} \text{double} (\text{S} \ x) \quad (5.14) \\
\text{plus} (\text{double} \ x) \ y &=_{CR} ^{\alpha} \text{plus} (\text{double'} \ x) \ y \quad (5.15)
\end{align*}
\]

We can observe that, in this case, all the equations are unfalsified due to the nature of the example. Moreover, all equations hold with the \( =_{CR} \) relation. This is due to the asymmetry in the definition of the two versions of \( \text{double} \): although the computed results of both versions are the same, there exist contexts in which the terms behave differently. This characteristic of the program is not easy to realize by just looking at the code, thus this is an example of the usefulness of having different notions of equivalence.

Finally, Equation (5.14) is an unfalsified equation that states a property which is false for the program. This is due to the approximation of the abstraction. It is worth noting that we would need to completely compute the (infinite) concrete semantics in order to discard the equation from the specification.

We do not remove unfalsified equations from the specifications since they have their own interest. Although it might be unfeasible to guarantee correctness of some equations (as in the example above), unfalsified equations may nevertheless show behaviors of the program which are actually correct. This is the only possible situation that arises in testing-based approaches, where all the equations must be considered unfalsified since it is impossible to distinguish them from correct equations. In any case, we can try to prove correctness of these equations by using a complementary verification or validation technique.

### 5.1 AbsSpec: The prototype

We have implemented the basic functionality of this methodology in a prototype written in Haskell. The core of the AbsSpec 1.0 prototype\(^8\) consists of about 800 lines of code implementing the tasks of generating and classifying terms. The inference core of AbsSpec 1.0 is generic w.r.t. the abstract domain, i.e., the operations implementing the abstract domain are passed to the generic inference process. Note that the AbsSpec 1.0 prototype invokes the semantics’ prototype implementation, which consists of about 7500 additional lines of code. On top of the core part of the prototype, the interface module implements some functions that allow the user both to check if a specific set of equations hold, or to get the whole specification. It is worth noting that, although in this paper we consider as input Curry programs, the prototype also accepts programs written in (the first order fragment of) Haskell (which are automatically converted by orthogonalization into Curry equivalent programs).

Unfortunately, we do not know of sets of benchmarks in the literature to be used to evaluate the prototype. Hence, we wrote some examples as a proof of concept in order to get some impressions on the efficacy of our proposal. Since the prototype does not han-

---

\(^8\) Available at [http://safe-tools.dsic.upv.es/absspec](http://safe-tools.dsic.upv.es/absspec).


dle built-in arithmetic operators yet, we tested it on both Curry and Haskell programs which do not involve arithmetics (mainly implementation of abstract data structures like queues, binary trees, arrays, heaps, etc.).

The experiments were conducted on an Intel Core2 Quad CPU Q9300(2.50GHz) with 6 Gigabytes of RAM. AbsSpec 1.0 was compiled with version 6.12.3 of the Glorious Glasgow Haskell Compilation System (GHC). Table 1 shows the execution times for the inference of each program example with some additional information. Column Program shows the name of the example program. The first three cases correspond to the examples shown in this paper. The fourth example is a more elaborated logic example where a data structure representing formulas is defined; column Rules shows the number of rules defining the program; column API size shows the number of operations included in the relevant API for the experiment. For the Booleans example, the experiment includes the operator for the logic implication defined explicitly (not in terms of the other boolean operators); column Terms shows the number of terms generated (thus, whose semantics is computed) during the inference process; columns \(\mathit{eq}\), and \(\mathit{ineq}\) show, respectively, the number of \(\mathit{eq}\), and \(\mathit{ineq}\) equivalence classes with more than one term that have been generated.\(^9\)

Our preliminary experiments show that many interesting properties hold over the depth \(k\) domain with low \(k\) values (we run the prototype with depth 7 by default). Also, many interesting properties show up with \(\maxsize = 3\). For example, we can see that for the Queue example, with \(\maxsize = 2\), only one equivalence class is defined whereas for \(\maxsize = 3\), 13 (sequences of) equations belong to the specification. We have used also \(\maxsize = 4\), but specifications tend to be less comprehensible for the user (64 equivalence classes for the same Queue example). Hence, increasing this value should be done only when bigger terms make sense, being at the same time very careful in choosing a sufficiently small API. The Double example illustrates the usefulness of the \(\mathit{ineq}\) equations: with \(\maxsize = 2\) we have already five of these equations.

The last example illustrates the fact that with a complex data structure, the high number of generated terms penalizes the inference process. Not that the increase of the number of generated terms depends, not only on the number of elements included in the relevant API (and on the number of arguments of the functions in the API), but also on the semantics of the program. Intuitively, if we have a program in which many terms share the same semantics, then fewer terms will be generated.

We also made some experiments on programs which use arithmetics by simulating the arithmetical operations using Peano’s notation but, as one would expect, we got poor results.

A serious evaluation of the prototype should be done on a number of standard benchmarks (hopefully) not written by ourselves, and we hope to get several contributions in this sense by the international community.

6. Related Work

To the best of our knowledge, in the functional logic setting there are currently no proposals for specification synthesis. There is a testing tool, EasyCheck [7], in which specifications are used as the input for the testing process. Given the properties, EasyCheck executes ground tests in order to check whether the property holds.

This tool could be used as a companion tool of ours in order to check if the unfalsified \(\mathit{eq}\) equations can be actually falsified. However EasyCheck is not capable of checking the \(\mathit{eq}\) and \(\mathit{ineq}\) equations because it is based only on the execution of ground tests.

The mentioned tool QuickSpec [8] computes an algebraic specification for Haskell programs by means of (almost black-box) testing. Like our approach, its inferred specifications are complete up to a certain depth of the analyzed terms because of its exhaustiveness. However, the equations in their specification are all unfalsified due to the use of testing for the equation generation. Instead, we follow a (glass-box) semantic-based approach that allows us to compute specifications as complete as those of QuickSpec, but with correctness guarantees on part of them (depending on the abstraction).

We have not done an exhaustive comparison, but performance of QuickSpec is better than that of our prototype for similar programs. However, we have to recall that our purpose is more ambitious since, for the case of functional-logic languages, using just the ground equivalence is not enough: important behaviors regarding the loss of contextual closeness would not show up, as shown by the double example. Moreover, we can deal with non terminating programs.

Finally, a previous work ([6]) identifies the additional difficulties for the inference of high level (property-based) specifications for the functional-logic paradigm.

7. Conclusions and Future Work

This paper presents a method to automatically infer high-level, property-oriented (algebraic) specifications in a functional logic setting. A specification represents relations that hold between operations (nested calls) in the program.

The method computes a concise specification of program properties from the source code of the program. We hope to have convinced the reader that we reached our main goal, that is to get a concise and clear specification that is useful for the programmer in order to detect possible errors, or to check that the program corresponds to the intended behavior.

The computed specification is particularly well suited for program understanding since it allows to discover non-evident behaviors, and also to be combined with testing. In the context of (formal) verification, the specification can be used to ease the verification tasks, for example by using the correct equations as annotations, or unfalsified equations as candidate axioms to be proven.

The approach relies on the computation of the semantics. Therefore, to achieve effectiveness and good performance results, we use a suitable abstract semantics instead of the concrete one. This means that we may not guarantee correctness of all the equations in the specification, but we can nevertheless infer correct equations thanks to a good compromise between correctness and efficiency.

We have developed a prototype that implements the basic functionality of the approach. We are working on the inclusion of all the functionality described in this paper.

As future work, we plan to add another notion of equivalence class \(\equiv\). More specifically, when dealing with a user-defined data type, the user may have defined a specific notion of equivalence by means of an “equality” function. In this context, some interesting equations could show up. For instance, in the Queue example, the predicate \(\mathit{eq}\) identifies queues which contain the same elements and Equation (5.10) should be presented as \(\mathit{eq}\) for the relation \(\equiv\) and \(\mathit{ineq}\) for the relation \(\neq\). These equivalence classes would depend on how the user defines the equality notion, thus we will surely need some assumptions (to ensure that indeed the user-defined function induces an equality relation) in order to get useful specifications.

Acknowledgments

M. A. Felíu and A. Villanueva have been partially supported by the EU (FEDER), the Spanish MICINN/MINECO under grant TIN2010-21062-C02-02, the Spanish MEC FPU grant AP2008-00608, and by the Generalitat Valenciana, ref. PROMETEO2011/052.
<table>
<thead>
<tr>
<th>Program</th>
<th>Rules</th>
<th>API size</th>
<th>Term size</th>
<th>Terms</th>
<th>=c (# classes)</th>
<th>=CR (# classes)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Booleans</td>
<td>9</td>
<td>5</td>
<td>2</td>
<td>121</td>
<td>65</td>
<td>1</td>
<td>0m0.130s</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2549</td>
<td>4</td>
<td>1</td>
<td>410</td>
<td>1</td>
<td>0m6.350s</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6399</td>
<td>2</td>
<td>34</td>
<td>1378</td>
<td>1</td>
<td>0m42.320s</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>172</td>
<td>2</td>
<td>3</td>
<td>122</td>
<td>1</td>
<td>0m0.330s</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>47</td>
<td>5</td>
<td>56</td>
<td>8</td>
<td>0</td>
<td>0m0.580s</td>
</tr>
<tr>
<td>Queue</td>
<td>11</td>
<td>6</td>
<td>2</td>
<td>25</td>
<td>0</td>
<td>5</td>
<td>0m0.110s</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>676</td>
<td>2</td>
<td>676</td>
<td>55</td>
<td>15/</td>
<td>0m19.860s</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2638</td>
<td>4</td>
<td>2638</td>
<td>410</td>
<td>.344</td>
<td>0m43.710s</td>
</tr>
<tr>
<td>Double</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>42</td>
<td>0</td>
<td>0</td>
<td>3m53.980s</td>
</tr>
<tr>
<td>BooleanDataStruct</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1648</td>
<td>2</td>
<td>0</td>
<td>9m0.140s</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1. Inference process of example programs

References


