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 that 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, e_2 are generic program expressions that (should) have the same computational behavior. This approach has two properties that we like:

PPDP'12, September 19–21, 2012, Leuven, Belgium.

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 *semantic-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[1], 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`:

```
and True x = x
and False _ = False
or True _ = True
or False x = x
not True = False
not False = True
imp False x = True
imp True x = x
```

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 this example, one could expect a (property-oriented) specification with equations like[2]

```
imp x y = or (not x) y                    (2.1)
not (or x y) = and (not x) (not y)        (2.2)
not (and x y) = or (not x) (not y)        (2.3)
not (not x) = x                           (2.4)
and x (and y z) = and (and x y) z         (2.5)
and x y = and y x                         (2.6)
```

which are well-known laws among the (theoretical) boolean operators. This comprehensible 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. (2.7)

In the following, we call this notion of equivalence *computed result equivalence* and we denote it by $=_{CR}$.

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 $\{\texttt{x}/\texttt{True}\} \cdot \texttt{True}$ denotes that the normal form `True` (at the right of the $\cdot$ symbol) has been reached with computed answer substitution $\{\texttt{x}/\texttt{True}\}$ (at the left of the $\cdot$ 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: $\{\texttt{x}/\texttt{True}\} \cdot \texttt{True}$ and $\{\texttt{x}/\texttt{False}\} \cdot \texttt{False}$, whereas the right hand side of the equation `x` evaluates just to $\{\} \cdot \texttt{x}$. Note however that any ground instance of the two goals evaluates to the same results, namely both `True` and `not (not True)` evaluate to $\{\} \cdot \texttt{True}$, and both `False` and `not (not False)` evaluate to $\{\} \cdot \texttt{False}$.

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 $=_G$. 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[3] w.r.t. its operational behavior, i.e., an expression can produce different computed values

[1] Actually, different from a language without logical variables that may be instantiated during execution.

[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-

when it is embedded in a context that binds its free variables (as shown by the following example), which makes the answer to the question posed above not straightforward.

EXAMPLE 2.2
Given a program with the following rules

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

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

Thus, in the Curry case, 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 $=_C$. Actually, Equations (2.1), (2.2), (2.3) and (2.5) are valid w.r.t. this equivalence notion.

We can see that $=_C$ is (obviously) stronger than $=_{CR}$, which is in turn stronger than $=_G$. As a conclusion, for our example we would get the following (partial) specification.[4]

$$\texttt{imp x y} =_C \texttt{or (not x) y}$$
$$\texttt{not (or x y)} =_C \texttt{and (not x) (not y)}$$
$$\texttt{not (and x y)} =_C \texttt{or (not x) (not y)}$$
$$\texttt{and x (and y z)} =_C \texttt{and (and x y) z}$$
$$\texttt{not (not x)} =_G \texttt{x}$$
$$\texttt{and x y} =_G \texttt{and y x}$$

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 equalities 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 expression are also equivalent within any context.

The need of determining $=_C$ 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 nested within some outer context in order to establish their $=_C$ 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/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 $=_C$ 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 specification. 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 Σ which is partitioned in $\mathcal{C}$, the *constructor* symbols, and $\mathcal{D}$, the *defined* symbols. $\mathcal{V}$ denotes a (fixed) countably infinite set of variables and $\mathcal{T}(\Sigma, \mathcal{V})$ denotes the terms built over signature Σ and variables $\mathcal{V}$. 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, 14] 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. $=_C$. 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 $\mathcal{E}[\![t]\!]$ which, for any term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, gives the semantics of t as $\mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]}$. Intuitively, the evaluation $\mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]}$ computes a 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 σ 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 $\mathcal{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho)$ that may contain special variables $\varrho_0, \varrho_1, \ldots \in \mathcal{V}_\varrho$ (a set disjoint from $\mathcal{V}$) indicating an unevaluated subterm. Leaves with no occurrences of special variables are computed results. We denote by $cr(T)$ 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 $cr(\mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]})$ corresponds to the set of computed outcomes of t using P. Moreover, given two terms e and 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 $\mathcal{T}(\Sigma, \mathcal{V})$. Then $cr(\mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]})$ corresponds to the set of computed outcomes of t using P.*

Moreover, $\mathcal{E}[\![\cdot]\!]_{\mathcal{F}[\![P]\!]}$ fulfills referential transparency:

THEOREM 3.2 *Let P be a first-order Curry program, e, e' terms in $\mathcal{T}(\Sigma, \mathcal{V})$, 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:

$$\mathcal{F}[\![P]\!] = \begin{cases} \texttt{g x} \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} \varepsilon \cdot \texttt{C } \varrho_1 \xrightarrow{\varrho_1} \{\texttt{x/A}\} \cdot \texttt{C A} \\ \texttt{g' x} \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} \{\texttt{x/A}\} \cdot \texttt{C A} \\ \texttt{h x} \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} \{\texttt{x/A}\} \cdot \texttt{A} \\ \texttt{f x y} \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} \{\texttt{x/C x'}, \texttt{y/B}\} \cdot \texttt{B} \end{cases}$$

sions e, e' implies the semantics equivalence of e and e' when used within any context $C[\cdot]$. Namely, $\forall e, e', C.\ [\![e]\!] = [\![e']\!] \implies [\![C[e]]\!] = [\![C[e']]\!]$.

[4] As we will show later, our technique computes a complete specification for a specific *size* of terms in equations.

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:

$$\mathcal{E}[\![\texttt{f (g x) x}]\!]_{\mathcal{F}[\![P]\!]} = \varepsilon \cdot \varrho \xrightarrow{\varrho} \{\texttt{x/B}\} \cdot \texttt{B}$$
$$\mathcal{E}[\![\texttt{f (g' x) x}]\!]_{\mathcal{F}[\![P]\!]} = \varepsilon \cdot \varrho$$

Note that $\varepsilon \cdot \varrho$ is not a computed result due to the occurrence of the ϱ variable.

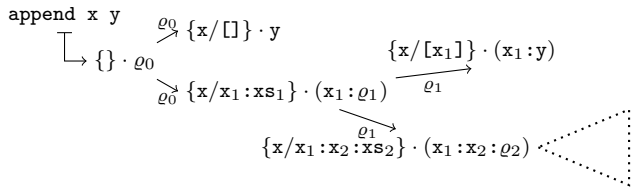
Even if this semantics is condensed, the trees in denotations can be infinite both in depth and in width, as we show with the following example.

EXAMPLE 3.4

Consider the classical append function:

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

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



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 $\mathcal{S}$ 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 $t_1, t_2, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$. 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 $=_{CR}$. 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 $=_{CR}$ just by collecting the leaves of $\mathcal{E}[\![e]\!]_{\mathcal{F}[\![P]\!]}$. This means that if we define a function cr that, given a semantic tree computed by the evaluation function, collects the leaves of the tree, then it holds that

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

The $=_{CR}$ equivalence is coarser than $=_C$ ($=_C \subseteq =_{CR}$) 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 $=_C \subseteq =_{CR} \subseteq =_G$ 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 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. 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 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.

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.

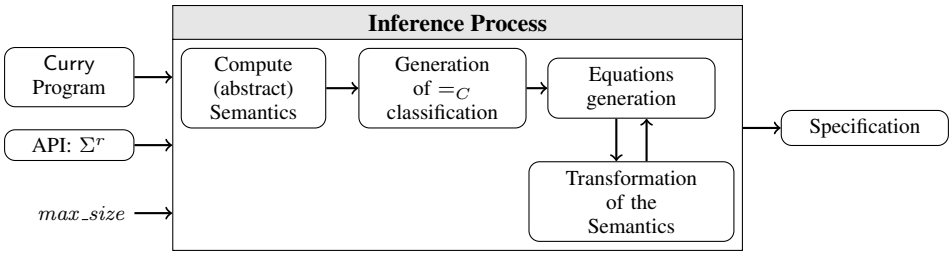


Figure 1. A general view of the inference process.

Algorithm 1 Inference of an algebraic specification

Require: Program P;
Program's *relevant* API Σ^r;
Maximum term size max_size

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

- $sem(ec)$: the semantics of (all) the terms in that class
- $rep(ec)$: the *representative term* of the class, which is defined as the smallest term in the class (w.r.t. the function $size$), and
- $terms(ec)$: the set of terms belonging to that equivalence class.

The *representative term* is used in order to avoid much redundancy in the generation of equations. 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 $\langle \mathcal{E}[\![x]\!]_{\mathcal{F}[\![P]\!]}, x, \{x\}\rangle$;[5]
- 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 $\langle \mathcal{E}[\![t_1]\!]_{\mathcal{F}[\![P]\!]}, t_1, \{t_1, t_2\}\rangle$.

Generation of $=_C$ classification. The second phase of the algorithm, Lines 3 to 14, is the (iterative) computation of the classification of terms w.r.t. $=_C$. 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 Σ^r (Line 5) [6] 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., $t_i = rep(ec_i)$. In this way, by construction, the term t has surely not been considered yet; Then, we compute the semantics $s = \mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]}$ and update the current partition $part'$ by using the auxiliary method $add\_to\_partition(t, s, part')$ (Lines 7 to 11). Here, the compositionality of the semantics makes possible to compute the semantics of terms (Line 9) efficaciously: since the semantics $s_i = sem(ec_i)$ 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.[7]

The $add\_to\_partition(t, s, part)$ function looks for an equivalence class ec in the current classification $part$ 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 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:

$$t_{1.1} := \texttt{not x}$$
$$t_{1.2} := \texttt{and x y}$$
$$t_{1.3} := \texttt{or x y}$$

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

$$t_{2.1} := \texttt{and (not x) (not x')}$$
$$t_{2.2} := \texttt{not (or x y)}$$

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 is 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., $rep(ec_1) = t_{2.2}$).

From this point on, only the representative of the class will be used for constructing new terms. This means that terms like `not (and (not x) (not x'))`, which is the instantiation of $t_{1.1}$ with $t_{2.1}$, 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, $t_{2.1}, t_{2.2} \in terms(ec_1)$, which means that, as we describe below, the equation `and (not x) (not y)` $=_C$ `not (or x y)` can be generated. Now, assume that there exists a second equivalence class, ec_2, that includes the terms `not (not x)` and `x`, thus allowing the generation of the equation `not (not 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 `not (and (not x) (not y))`, which is the instantiation of `not x` with $t_{2.1}$. Let us suppose that the semantics of this term is the same as that of the term `or x y`, thus it is added to its equivalence class. This implies that the corresponding equation `not (and (not x) (not y))` $=_C$ `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 (not x) (not y)` $=_{CR}$ `not (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 $t_{11} =_{CR} t_{12}$ is uninteresting, since we already know $t_{11} =_C t_{12}$ and equation $t_{21} =_{CR} t_{31}$ is redundant (because $t_{21} =_C t_{22}$ and $t_{22} =_{CR} t_{31}$).

In summary, if $t_1 =_C t_2$ holds, then $t_1 =_{\{CR,G\}} t_2$ 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 instancies. The transformation which corresponds to this representation

- retains only the most general instancies of the original semantics (removing computed results which are instancies 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 instancies of r is the same as that of the union of all ground instancies 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. $=_G$) 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 $depth(k)$ abstract domain. 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.

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, $depth(k)$ 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 $depth(k)$ 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 *have not been falsified up to that point*, analogously to what happens in the testing-based approach. We call these equations *unfalsified* equations. When showing the specification, we mark the latter with a special equivalence symbol $=^{\alpha}_{C}$. Unfalsified equations are the only kind of equations that testing-based approaches can compute in general.

The main advantage of our proposal w.r.t. the testing-based approaches is the fact that we are able to distinguish when an equation *certainly* holds, and when it just *can* hold. Moreover, we can deal with non terminating programs.

Since the overall construction is (almost) independent of the actual structure of the abstract semantics, it would be possible in the future to use other abstract domains to reach a better trade-off between efficiency of the computation and accuracy of the specifications.

5. Case Studies

Let us start by discussing the results for a more elaborated example. The following Curry program implements a two-sided queue where it is possible to insert or delete elements on both left and right sides:

```
data Queue a = Q [a] [a]

new = Q [] []

inl x (Q xs ys) = Q (x:xs) ys
inr x (Q xs ys) = Q xs (x:ys)

outl (Q [] ys) = Q (tail (reverse ys)) []
outl (Q (_:xs) ys) = Q xs ys

outr (Q xs []) = Q [] (tail (reverse xs))
outr (Q xs (_:ys)) = Q xs ys

null (Q [] []) = True
null (Q (_:_) _) = False
null (Q [] (_:_)) = False

eqQ (Q xs ys) (Q xs' ys') =
  (xs++reverse ys) =:= (xs'++reverse 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 `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. 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:

$$\texttt{null new} =_C \texttt{True} \tag{5.1}$$

$$\texttt{new} =_C \texttt{outl (inl x new)} =_C \\ =_C \texttt{outr (inr x new)} \tag{5.2}$$

$$\texttt{outl (inl x q)} =_C \texttt{outr (inr x q)} \tag{5.3}$$

$$\texttt{outr (inl x new)} =_C \texttt{outl (inr x new)} \tag{5.4}$$

$$\texttt{inr x (inl y q)} =_C \texttt{inl y (inr x q)} \tag{5.5}$$

$$\texttt{inl x (outl (inl y q))} =^{\alpha}_{G} \\ =^{\alpha}_{G} \texttt{outr (inl x (inr y q))} \tag{5.6}$$

$$\texttt{outl (inl x (outl q))} =^{\alpha}_{G} \\ =^{\alpha}_{G} \texttt{outl (outl (inl x q))} \tag{5.7}$$

$$\texttt{outr (outl (inl x q))} =_C \\ =_C \texttt{outl (inl x (outr q))} \tag{5.8}$$

$$\texttt{null (inl x new)} =_C \texttt{null (inr x new)} =_C \\ =_C \texttt{False} \tag{5.9}$$

$$\texttt{eqQ (inr x new) y} =_C \texttt{eqQ (inl x new) y} \tag{5.10}$$

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 *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, `inr x new` is equivalent to `inl x 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 $\Sigma^r = \{$`inl, 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 `Queue a` and `a`, respectively.

$$ec_{1.1} = \langle s_{1.1}, \texttt{inl x q}, \{\texttt{inl x q}\}\rangle$$
$$ec_{1.2} = \langle s_{1.2}, \texttt{outl q}, \{\texttt{outl q}\}\rangle$$
$$ec_{1.3} = \langle s_{1.3}, \texttt{q}, \{\texttt{q}\}\rangle$$
$$ec_{1.4} = \langle s_{1.4}, \texttt{x}, \{\texttt{x}\}\rangle$$

During the second iteration, the term `inl x (outl q)` is built as the instantiation of the representative of $ec_{1.1}$ with the representative of $ec_{1.2}$. This term adds a new equivalence class to the partition:

$$ec_{2.1} = \langle s_{2.1}, \texttt{inl x (outl q)}, \{\texttt{inl x (outl q)}\}\rangle$$

During the same iteration, also the term `outl (inl x q)` is built. For the sake of this discussion, we assume that it has the same semantics of `q`:

$$ec_{1.3} = \langle s_{1.3}, \texttt{q}, \{\texttt{q,outl (inl x q)}\}\rangle$$

This means that the equation `outl (inl x q)` $=_C$ `q` will be generated from $ec_{1.3}$. Then, in the third iteration, the term `outl (inl x (outl q))` is built as the instance of the representative of $ec_{1.2}$ with the representative of $ec_{2.1}$. The semantics of this new term coincides with that of `outl q`, thus $ec_{1.2}$ is updated:

$$ec_{1.2} = \langle s_{1.2}, \texttt{outl q}, \{\texttt{outl q,outl (inl x (outl q))}\}\rangle$$

As a consequence, the equation `outl (inl x (outl q))` $=_C$ `outl q` is generated. However, this equation is redundant because it is an instance of the equation already generated from $ec_{1.3}$, `outl (inl x q)` $=_C$ `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` $=_G$ `outl (inl x 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:

```
data Nat = Z | S Nat

double, double' :: Nat -> Nat
double Z = Z
double (S x) = S (S (double x))

double' x = plus x x

plus :: Nat -> Nat -> Nat
plus Z y = y
plus (S x) y = S (plus x y)
```

Some of the inferred equations for the program are:

$$\texttt{double' (double' (double x))} =^{\alpha}_{CR} =^{\alpha}_{CR} \texttt{double' (double (double' x))} \quad (5.11)$$

$$\texttt{double x} =^{\alpha}_{CR} \texttt{double' x} \quad (5.12)$$

$$\texttt{double' (double' x)} =^{\alpha}_{CR} \texttt{double' (double x)} =^{\alpha}_{CR} =^{\alpha}_{CR} \texttt{double (double' x)} =^{\alpha}_{CR} =^{\alpha}_{CR} \texttt{double (double x)} \quad (5.13)$$

$$\texttt{(S (double x))} =^{\alpha}_{CR} \texttt{double (S x)} \quad (5.14)$$

$$\texttt{plus (double x) y} =^{\alpha}_{CR} \texttt{plus (double' x) y} \quad (5.15)$$

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 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 to 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.

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 $=_{\mathbf{C}}$, and $=_{CR}$ show, respectively, the number of $=_C$, and $=_{CR}$ 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 $max\_size = 3$. For example, we can see that for the Queue example, with $max\_size = 2$, only one equivalence class is defined whereas for $max\_size = 3, 13$ (sequences of) equations belong to the specification. We have used also $max\_size = 4$, 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 $=_{CR}$ equations: with $max\_size = 2$ 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 $=_G$ equations can be actually falsified. However EasyCheck is not capable of checking the $=_C$ and $=_{CR}$ 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 equations 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 $=_{Ueq}$. 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 `eqQ` identifies queues which contain the same elements and Equation (5.10) should be presented as `inl x new` $=_{Ueq}$ `inr x new`. 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. Feliú 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.

[9] The prototype does not compute the $=_G$ equations yet.

Program	Rules	API size	Term size	Terms	$=_C$ (# classes)	$=_{CR}$ (# classes)	Time
Booleans	9	5	2	121	65	1	0m0.130s
			3	2549	410	1	0m6.550s
			4	6399	1378	1	0m42.320s
Queue	11	6	2	34	1	0	0m0.080s
			3	132	12	1	0m0.180s
			4	473	56	8	0m0.580s
Double	5	3	2	25	0	5	0m0.110s
			3	676	55	157	0m19.860s
			4	2638	410	344	0m43.710s
BooleanDataStruct	8	4	2	42	0	0	3m53.980s
			3	1648	2	0	9m0.140s
			4	-	-	-	-

Table 1. Inference process of example programs

References

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.

[2] 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. ISBN 1-58113-450-9. doi: http://doi.acm.org/10.1145/503272.503275. URL http://doi.acm.org/10.1145/503272.503275.

[3] G. Bacci. *An Abstract Interpretation Framework for Semantics and Diagnosis of Lazy Functional-Logic Languages*. PhD thesis, Dipartimento di matematica e Informatica, Università di Udine, 2011.

[4] G. Bacci and M. Comini. A Compact Goal-Independent Bottom-Up Fixpoint Modeling of the Behaviour of First Order Curry. Technical Report DIMI-UD/06/2010/RR, Dipartimento di Matematica e Informatica, Università di Udine, 2010. URL http://www.dimi.uniud.it/comini/Papers/.

[5] G. Bacci and M. Comini. Abstract Diagnosis of First Order Functional Logic Programs. In M. Alpuente, editor, *Logic-based Program Synthesis and Transformation, 20th International Symposium*, volume 6564 of *Lecture Notes in Computer Science*, pages 215–233, Berlin, 2011. Springer-Verlag. ISBN 9783642205507. doi: 10.1007/978-3-642-20551-4_14.

[6] G. Bacci, M. Comini, M. A. Feliú, and A. Villanueva. The additional difficulties for the automatic synthesis of specifications posed by logic features in functional-logic languages. In *ICLP (Technical Communications)*, volume To appear of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum Fuer Informatik, 2012.

[7] J. Christiansen and S. Fischer. Easycheck – test data for free. In *Proceedings of the 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2008.

[8] K. Claessen, N. Smallbone, and J. Hughes. QuickSpec: Guessing Formal Specifications using Testing. In *4th International Conference on Tests and Proofs (TAP 2010)*, volume 6143, pages 6–21, 2010. ISBN 978-3-642-13976-5.

[9] 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.

[10] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, San Antonio, Texas, January 29–31*, pages 269–282, New York, NY, USA, 1979. ACM Press.

[11] C. Ghezzi and A. Mocci. Behavior model based component search: an initial assessment. In *Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation (SUITE'10)*, pages 9–12, New York, NY, USA, 2010. Acm. ISBN 978-1-60558-962-6.

[12] 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.

[13] M. Hanus. A unified computation model for functional and logic programming. In *24th ACM Symposium on Principles of Programming Languages (POPL 97)*, pages 80–93, 1997.

[14] M. Hanus. Curry: An integrated functional logic language (vers. 0.8.2), 2006. Available at URL: http://www.informatik.uni-kiel.de/~curry.

[15] J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for java container classes. *IEEE Transactions on Software Engineering*, 33(8):526–542, 2007.

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

[17] I. Nunes, A. Lopes, and V. Vasconcelos. Bridging the Gap between Algebraic Specification and Object-Oriented Generic Programming. In S. Bensalem and D. Peled, editors, *9th International Workshop on Runtime Verification (RV 2009)*, volume 5779 of *Lecture Notes in Computer Science*, pages 115–131. Springer, 2009.

[18] S. Peyton Jones. *Haskell 98 Language and Libraries - The Revised Report*. Cambridge University Press, Cambridge, UK, 2003. ISBN 0521826144. Available at http://www.haskell.org/definition/.

[19] D. Rayside, A. Milicevic, K. Yessenov, G. Dennis, and D. Jackson. Agile specifications. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, pages 999–1006. Acm, 2009.

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

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

[22] B. Yu, L. Kong, Y. Zhang, and H. Zhu. Testing Java Components based on Algebraic Specifications. In *First International Conference on Software Testing, Verification, and Validation (ICST 2008)*, pages 190–199. IEEE Computer Society, 2008.