The additional difficulties for the automatic synthesis of specifications posed by logic features in functional-logic languages*

Giovanni Bacci[1], Marco Comini[1], Marco A. Feliú[2], and Alicia Villanueva[2]

1 **Dipartimento di Matematica e Informatica**
University of Udine (Italy)
{giovanni.bacci,marco.comini}@uniud.it
2 **DSIC, Universitat Politècnica de València (Spain)**
{mfeliu,villanue}@dsic.upv.es

Abstract

This paper discusses on the additional issues for the automatic inference of algebraic property-oriented specifications which arises because of interaction between laziness and logical variables in lazy functional logic languages.

We present an inference technique that overcomes these issues for the first-order fragment of the lazy functional logic language Curry. 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. Our proposal is a (glass-box) semantics-based inference method which can guarantee, to some extent, the correctness of the inferred specification, differently from other (black-box) approaches based on testing techniques.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Curry, property-oriented specifications, semantics-based inference methods

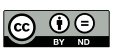
Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.144

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, 16, 6, 12, 8, 19, 14, 9]. We can find several proposals of (automatic) inference of high-level specifications from an executable or the source code of a system, like [2, 6, 12, 9], which have proven to be very helpful.

There are many classifications in the literature depending on the characteristics of specifications [13]. It is common to distinguish between *property-oriented* specifications and *model-oriented* or *functional* specifications. Property-oriented specifications are of higher description level than other kinds of specifications: they consist in an indirect definition of the system's behavior by means of stating a set of properties, usually in the form of axioms,

* 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.



that the system must satisfy [18, 17]. 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.

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.

We can identify two mainstream approaches to perform the inference of specifications: glass-box and black-box. The glass-box approach [2, 6] 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 [6]. 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 [6] or to verify that a given property holds [2]. The black-box approach [12, 9] 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) [9]. 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).

QuickSpec [6] is an (almost) black-box approach for Haskell programs [15] based on testing. The tool 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:

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.

We aim to develop a method with similar outcomes for the lazy functional logic language Curry [10, 11]. 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).

However, due to its very high-level features (in particular lazy evaluation and logical variables), the problem of inferring specifications for this kind of languages immediately poses several additional problems w.r.t. the functional paradigm (as well as other paradigms).

In this paper we discuss these issues in detail and we motivate why any proposal that aims to infer property-oriented specifications, like those of the QuickSpec approach, for lazy functional logic languages need to be radically different from the method used by QuickSpec.

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 program rules are evaluated non-deterministically.[1] Differently from the functional case

[1] Variables and function names start by a character in lower case, whereas data constructors and type names start by a letter in upper case. For a complete description of the Curry language, the interested

(of QuickSpec), 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.

▶ **Example 1** (Boolean logic example)**.** Consider the definition for the boolean data type with values `True` and `False` and boolean 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 the program of Example 1, one could expect to have in its property-oriented specification equations like

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

which are well-known laws among the (theoretical) boolean operators. 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. (2.5)

In the following, we call this equivalence *computed result equivalence* and we denote it by $=_{CR}$. Actually, Equations (2.1), (2.2) and (2.3) are *literally* valid in this sense since, in Curry, free variables are admitted in expressions, and the mentioned equations are valid *as they are*. This is quite different from the 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 since the goal `not (not x)` is evaluated to $\{\texttt{x/True}\} \cdot \texttt{True}$[2] and $\{\texttt{x/False}\} \cdot \texttt{False}$, whereas `x` is evaluated 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)` are evaluated to $\{\} \cdot \texttt{True}$, and both `False` and `not (not False)` are evaluated to $\{\} \cdot \texttt{False}$.

Decidedly, also this notion of *ground equivalence* is interesting for the user, and we denote it by $=_G$. This notion coincides with the (only possible) one 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

reader can consult [11].

[2] The expression $\{\texttt{x/True}\} \cdot \texttt{True}$ denotes that the normal form `True` has been reached with computed answer substitution $\{\texttt{x/True}\}$.

$e_1 = e_2$, 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 when it is embedded in a context that binds its free variables (as shown by the following artificial example), which makes the answer to the question posed above not straightforward.

▶ **Example 2.** Given a program with the following rules

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

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

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

Thus, in the Curry case, it becomes mandatory to *additionally* ask in the equivalence notion of (2.5) 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) and (2.2) 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.

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

The inference of $=_C$ equalities poses serious issues to testing-based methods like QuickSpec. First, expressions 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 level of depth can 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 will degrade sensibly. Second, since the typical real life case is that the program in consideration is just a module of a complex software system, it may happen that no function in the considered module can discriminate contexts; but in other modules there could be one. Clearly, we could imagine to run the tool on the entire system but, besides the obvious increment of cost, the "caller" module could have not been implemented yet. Thus, we would need to reconsider the outputs of synthesis whence some new module is added.

This kind of issues do not arise with languages, like Haskell, which are referentially transparent: essentially, languages where the semantics of all nested expressions can be obtained by instantiating the semantics of the outer expression with those of the arguments.

Contrary to testing-based approaches, a semantics-based method that computes the (compositional) semantics of a part of code does not suffer of these issues[4]. Obviously, in

[3] The concept of referential transparency of a language can be stated in terms of a formal semantics as: the semantics equivalence of two expressions e, e' implies the semantics equivalence of e and e' when used within any context $C[\cdot]$. Namely, $\forall e, e', C.\ [\![e]\!] = [\![e']\!] \implies [\![C[e]]\!] = [\![C[e']]\!]$.

[4] Evidently, the semantics to be employed needs to be fully abstract w.r.t. contextual embedding in order to compute correct $=_C$ equations.

this case the problem is the undecidability of the semantics' computation, thus suitable approximations must be used. This would lead to possibly erroneous equations, but this also happens with testing-based approaches.

Since Curry is *not* referentially transparent, we do not consider the semantics-based approach an option, but a must. In the following we present a first proposal of a semantics-based method that tackles the presented issues and discuss about its limitations.

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 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.

In order to state formally the equivalences described before we need two semantic evaluation functions $\mathcal{E}^{C}[\![]\!]$ and $\mathcal{E}^{CR}[\![]\!]$ which enjoy some properties.

$\boldsymbol{\mathcal{E}^{CR}[\![t;\ P]\!]}$ gives the *computed results* (CR) semantics of the term t with (definitions from) the program P. This semantics has to be fully abstract w.r.t. the behavior of computed results. Namely, the semantics of two terms t_1, t_2 are identical if and only if the evaluations of t_1 and t_2 compute the same results. It is theoretically possible to use just a correct semantics, but clearly in such case we will not have all equivalences which are valid w.r.t. a fully abstract semantics.

$\boldsymbol{\mathcal{E}^{C}[\![t;\ P]\!]}$ gives the *contextual* (C) semantics of the term t with the program P. This semantics has to be fully abstract w.r.t. the behavior of computed results *under any context*. Namely, the semantics of two terms t_1, t_2 are identical if and only if, for any context C, the evaluations of $C[t_1]$ and $C[t_2]$ compute the same results. We say that such a semantics fulfills referential transparency.

The semantics which can be obtained by collecting all results of the official small-step operational semantics of Curry [11, App. D.5], as well as the I/O semantics of [1], can be used for $\mathcal{E}^{CR}[\![t;\ P]\!]$ but they are not referentially transparent. On the contrary, the full small-step operational semantics of Curry is referentially transparent but is far from being fully-abstract.

The $\mathbb{WERS}$-semantics of [3, 4] is an appropriate choice for $\mathcal{E}^{C}[\![t;\ P]\!]$ and moreover the set of its leaves is an appropriate choice for $\mathcal{E}^{CR}[\![t;\ P]\!]$. However, our proposal does not rely on these particular semantics and any semantics which fulfills the aforementioned requirements can be used.

Now we are ready to formally introduce our notion of specification. 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 $=_C$. 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 $C[\cdot]$. Namely,

$$t_1 =_C t_2 \iff \mathcal{E}^{C}[\![t_1;\ P]\!] = \mathcal{E}^{C}[\![t_2;\ P]\!]$$

Computed-result equivalence $=_{CR}$. This equivalence states that two terms are equivalent when the outcomes of their evaluation are the same. Namely

$$t_1 =_{CR} t_2 \iff \mathcal{E}^{CR}[\![t_1;\, P]\!] = \mathcal{E}^{CR}[\![t_2;\, P]\!]$$

The $=_{CR}$ equivalence is coarser than $=_C$ ($=_C \subseteq =_{CR}$) as shown by Example 2.

Ground Equivalence $=_G$. This equivalence states that two terms are equivalent if all possible ground instances have the same outcomes. Namely

$$t_1 =_G t_2 \iff \forall \theta \,\text{grounding}.\ \mathcal{E}^{CR}[\![t_1\theta;\, P]\!] = \mathcal{E}^{CR}[\![t_2\theta;\, P]\!]$$

By definition, the $=_G$ equivalence is coarser than $=_{CR}$ ($=_{CR} \subseteq =_G$).

User Defined Equality Equations. The symbol $=_{Ueq}$ is used for *user-defined equality equations*. Equality equations depend upon a user-defined notion of equality. When dealing with a user-defined data type, the user may have defined a specific notion of equivalence by means of an "equality" function. Let us call $equal(t_1, t_2)$ such user-defined function. Then, we state that

$$t_1 =_{Ueq} t_2 \iff \mathcal{E}^{CR}[\![equal(t_1, t_2);\, P]\!] = \mathcal{E}^{CR}[\![\texttt{True}]\!] \iff equal(t_1, t_2) =_{CR} \texttt{True}$$

Clearly, we do not have necessarily any relation between $=_{Ueq}$ and the others, as the user function *equal* may have nothing to do with equality. However, in typical situations such a function is defined to preserve at least $=_G$, meaning that $t_1 =_G t_2$ implies $t_1 =_{Ueq} t_2$.

In any case, as clear from the definition, this is technically just a particular instance of $=_{CR}$, so it does not need to be considered by itself and in the following we will not consider it explicitly.

Nevertheless, these equations can provide the user significant information about the structure and behavior of the program and a pragmatical tool should reasonably present a sequence $\texttt{True} =_{CR} equal(t_1, t_2) =_{CR} \ldots =_{CR} equal(t_{n-1}, t_n)$ as $t_1 =_{Ueq} \ldots =_{Ueq} t_n$ for readability purposes.

Note that $=_G$ 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, we have $=_C \subseteq =_{CR} \subseteq =_G$ and only $=_C$ is referentially transparent (i.e., a congruence w.r.t. contextual embedding).

4 Deriving Specifications from Programs

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

There is also another source of redundancy due to the inclusion of relations $=_K$. For example, since $=_C$ is the finer relation, if $t =_C s$ then $t =_{CR} s$ and $t =_G s$. To avoid

the generation of coarser redundant equations, a simple solution is that of starting with $=_C$ equivalences and, once these are all settled, to proceed with the evaluation of the CR semantics *only* of non $=_C$ equivalent terms. Thereafter, we can evaluate the ground semantics of non $=_{CR}$ equivalent terms.

Let us describe in more detail the specification inference process. The input of the process consists of the Curry program to be analyzed and two additional parameters: a *relevant* API, Σ^r, and a maximum term size, *max_size*. The *relevant* API allows the user to choose the operations in the program that will be present in the inferred specification, whereas the maximum term size limits the size of the terms in the specification. The inference process consists of three phases, one for each kind of equality: first $=_C$ and then $=_{CR}$ and $=_G$. Terms are classified by their semantics into a data structure, which we call *classification*, consisting of a set of *equivalence classes* (*ec*) formed by

- *sem*(*ec*): the semantics of (all) the terms in that class;
- *terms*(*ec*): the set of terms belonging to that equivalence class;
- *rep*(*ec*): the *representative term* of the class (*rep*(*ec*) ∈ *terms*(*ec*)).

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

For the sake of comprehension, we present an untyped version of the method.

Computation of the initial classification

We initially create a classification which contains:

- one class for a free (logical) variable $\langle \mathcal{E}[\![x]\!], x, \{x\}\rangle$ [5];
- the classes for any built-in or user-defined constructor.

Then, for all symbols f/n of the relevant API, Σ^r, and distinct variables $x_1, \ldots, x_n$, we *add to classification* the term $t = f(x_1, \ldots, x_n)$ with semantics $s = \mathcal{E}^C[\![t; P]\!]$. This operation looks for an equivalence class *ec* in the current classification 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.

Generation of $=_C$ classification

We iteratively select all symbols f/n of the relevant API Σ^r and n equivalence classes $ec_1, \ldots, ec_n$ from the current classification. We build the term $t = f(t_1, \ldots, t_n)$, where each t_i is the representative term of ec_i, $t_i = rep(ec_i)$; then, we compute the semantics $s = \mathcal{E}^C[\![t; P]\!]$ and update the current classification by *adding to classification* t and s as described before.

If the classification changes, then we iterate by considering again all the symbols in the relevant API to build and evaluate new (maybe greater) terms. This phase is doomed to terminate because at each iteration we consider, by construction, terms which are different from those already existing in the classification and whose size is strictly greater than the size of its subterms (but the size is bounded by *max_size*).

Let us show an example:

[5] The typed version uses one variable for each type

▶ **Example 3.** Let us recall the program of Example 1 and choose as relevant API the functions `and`, `or` and `not`. In the first iteration, the terms $t_{1.1}$ = `not x`, $t_{1.2}$ = `and x y`, and $t_{1.3}$ = `or x y` are built. After computing the semantics, and since the semantics of none of them coincides with the semantics of an existing equivalence class, three new equivalence classes appear, one for each term.

During the second iteration, the following two terms (among others) are built: the term $t_{2.1}$ = `and (not x) (not x')` is built as the instantiation of $t_{1.2}$ with $t_{1.1}$, and the term $t_{2.2}$ = `not (or x y)` as 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, thus during this iteration (at least) this new class is computed. From this point on, only the representative of the class will be used for constructing new terms. We choose the smaller term as the representative, which in the example is $t_{2.2}$ ($rep(ec_1) = t_{2.2}$), thus terms like `not (and (not x) (not x'))` will never be built.

Generation of the $=_C$ specification

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

Generation of $=_{CR}$ equations

The second phase works on the former classification by first transforming each equivalence class ec by replacing the C-semantics $sem(ec)$ with $\mathcal{E}^{CR}[\![rep(ec); P]\!]$ and $terms(ec)$ with the (singleton) set $\{rep(ec)\}$.

After the transformation, some of the equivalence classes which had different semantic values may now have the same CR-semantics and then we merge them, making the union of the term sets $terms(ec)$.

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

Let us clarify this phase by an example.

▶ **Example 4.** Assume we have a classification consisting of three equivalence classes with C-semantics s_1, s_2 and s_3 and representative terms t_{11}, t_{22} and t_{31}:

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

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

Now, assume that $\mathcal{E}^{CR}[\![t_{11}]\!] = x_0$ and $\mathcal{E}^{CR}[\![t_{22}]\!] = \mathcal{E}^{CR}[\![t_{31}]\!] = x_1$. Then (since t_{12}, t_{13} and t_{21} are removed) we obtain the new classification

$$ec_4 = \langle x_0,\ t_{11},\ \{t_{11}\}\rangle \qquad ec_5 = \langle x_1,\ t_{22},\ \{t_{22}, t_{31}\}\rangle$$

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

Successive (sub-)phases

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

transform again the classification by replacing the *CR*-semantics with the ground semantics (and term sets with singleton term sets). Then we merge eventual equivalence classes with the same semantics and, finally, we generate $=_G$ equations for non singleton sets.

4.1 Feasibility considerations

In a semantics-based approach, one of the main problems to be tackled is effectiveness. The semantics of a program is in general infinite and thus some approximation has to be used in order to have a terminating method.

Several solutions can be adopted. To experiment on the validity of our proposal we have implemented the basic functionality of this methodology in a prototype, written in Haskell, available at `http://safe-tools.dsic.upv.es/absspec`. The computation of $\mathcal{E}^C[\![]\!]$ is based on an implementation of the immediate consequences operator $\mathcal{P}^\nu[\![P]\!]$ of the (fixpoint) $\mathbb{WERS}$-semantics of [3, 4]. To achieve termination, the prototype computes a fixed number of steps of $\mathcal{P}^\nu[\![P]\!]$. Then, it proceeds with the classification as described with a further enhancement which is possible due to properties of the $\mathbb{WERS}$-semantics. Namely, the semantics $\mathcal{E}^{CR}[\![]\!]$ can be obtained directly by transforming the $\mathcal{E}^C[\![]\!]$ semantics, concretely just by loosing internal structure. Therefore, no (costly) computation of $\mathcal{E}^{CR}[\![]\!]$ is needed, but just a quick filtering. The implementation of $=_G$ equalities is still ongoing work.

We are aware that many other attempts to guarantee termination could be used. Certainly, given our know-how, in the future we will experiment with abstractions obtained by abstract interpretation [7] (the $\mathbb{WERS}$-semantics itself has been obtained as an abstract interpretation).

5 Conclusions and Future Work

This paper discusses about the issues that arise for the automatic inference of high-level, property-oriented (algebraic) specifications because of the presence of logical features in functional-logic languages. Then, a first preliminary proposal which overcomes these issues is presented. 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 [5], 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.

Our 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 program's correctness.

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

It would be interesting in the future, once our proposal is mature, to investigate on the appropriateness also for referentially transparent languages like Haskell.

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.

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.

5 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.

6 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.

7 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.

8 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.

9 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.

10 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.

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

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

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

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

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

16 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.

17 H. van Vliet. *Software Engineering–Principles and Practice*. John Wiley, 1993.

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

19 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.