Semantic Versioning Checking in a Declarative Package Manager

Michael Hanus

Institut für Informatik, CAU Kiel, 24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract

Semantic versioning is a principle to associate version numbers to different software releases in a meaningful manner. The correct use of version numbers is important in software package systems where packages depend on other packages with specific releases. When patch or minor version numbers are incremented, the API is unchanged or extended, respectively, but the semantics of the operations should not be affected (apart from bug fixes). Although many software package management systems assumes this principle, they do not check it or perform only simple syntactic signature checks. In this paper we show that more substantive and fully automatic checks are possible for declarative languages. We extend a package manager for the functional logic language Curry with features to check the semantic equivalence of two different versions of a software package. For this purpose, we combine CurryCheck, a tool for automated property testing, with program analysis techniques in order to ensure the termination of the checker even in case of possibly non-terminating operations defined in some package. As a result, we obtain a software package manager which checks semantic versioning and, thus, supports a reliable and also specification-based development of software packages.

1998 ACM Subject Classification D.2.5 Testing and Debugging, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases functional logic programming, semantic versioning, program testing

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.6

1 Motivation

Contemporary software systems are complex and based on many components. To structure such systems and support the re-use of components in different software systems, software packages with well-defined APIs (application programming interfaces) are used. A software package consists of one or more modules and is used as a building block of a larger system. Hence, a software system or even a complex package depends on other packages. Since packages develop over time, e.g., new functionality is added, more efficient implementations are developed, or the usage of operations (i.e., the API) is changed, it is important to use appropriate versions of packages. Finding them and manage these dependencies is often called "dependency hell." As a solution to this problem, package managers use version numbers associated to package releases and allow to express such dependencies as relations on version numbers.

Semantic versioning is a recommendation to associate meaningful version numbers to software packages. In the semantic versioning standard,[1] each version number consists of major, minor, and patch number, separated by dots, and an optional pre-release specifier

[1] http://www.semver.org

Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017).
Editors: Ricardo Rocha, Tran Cao Son, Christopher Mears, and Neda Saeedloei; Article No. 6; pp. 6:1–6:16

Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

consisting of alphanumeric characters and hyphens appended with a hyphen (and optional build metadata, which we do not consider here). For instance, `0.1.2` and `1.2.3-alpha.2` are valid version numbers. Furthermore, an ordering is defined on version numbers where major, minor, and patch numbers are compared in lexicographic order and pre-releases are considered unstable so that they are smaller than their non-pre-release versions. For instance, `0.1.1` < `0.1.2` < `0.3.1` < `1.1.2-alpha` < `1.1.2`. Furthermore, semantic versioning requires that the major version number is incremented when the API functionality of a package is changed, the minor version number is incremented when new API functionality is added and existing API operations are backward compatible, and the patch version number is incremented when the API functionality is unchanged (only bug fixes, code refactorings, code improvements, etc).

The advantage of semantic versioning is an increased flexibility to choose packages when building larger software systems. For instance, if package `A` requires some functionality which has been introduced in version `2.3.1` of package `B`, one can specify that `A` depends on `B` in a version greater than or equal to `2.3.1` but less than `3.0.0`. Thanks to semantics versioning, a package manager can choose newer versions of `B` (as long as they are smaller than `3.0.0`), when they become available, in order to build `A` without dependency problems.

However, semantic versioning requires the semantic compatibility of two packages with identical major version numbers (apart from new operations or operations with bug fixes). Since this property is obviously undecidable in general, the developer is responsible for this semantic compatibility so that this is not checked in contemporary package management systems. Improving this situation is the objective of the work described in this paper. Due to the absence of side effects in declarative (functional, logic) programming languages, one can easily write repeatable test suites. Tests parameterized over some arguments are also called *properties*. Property-based testing automates the checking of properties by random or systematic generation of test inputs. It has been introduced with the QuickCheck tool [12] for the functional language Haskell and adapted to other languages, like PrologCheck [2] for Prolog, PropEr [26] for the concurrent functional language Erlang, or EasyCheck [11] and CurryCheck [17] for the functional logic language Curry.

In order to check the semantic equivalence of a unary operation f defined in versions v_1 and v_2 of some package, a first approach is renaming the definitions of f in these packages to f_{v_1} and f_{v_2}, respectively, and checking the property $\forall x.f_{v_1}(x) = f_{v_2}(x)$, which is called *computed result equivalence* in [9].[2] Ideally, one should prove this property. Since fully automatic proof techniques are available only for limited domains, we propose to use property-based testing instead. Although this method is incomplete in general, in practice it is quite successful if the generated input data is well distributed (which is a goal of all property-based test tools). Unfortunately, the brute-force testing of the equivalence of all operations, as described above, does not yield an automatic checker for semantic versioning, since it might not terminate if some operations are non-terminating. Moreover, declarative languages like Haskell or Curry are based on lazy evaluation to enable optimal computations and modularity by stream-based programming [20]. Hence, operations might also compute infinite results that cannot be compared in a finite amount of time. Therefore, we propose to combine property-based testing with program analysis techniques in order to ensure the termination of property testing. In general, operations which might not terminate are excluded from equivalence checking. In

[2] This property is a necessary but not sufficient condition to ensure semantic equivalence in functional logic programs [7]. Since we do not intend to provide a faithful method for semantic versioning checking but use a testing-based approach to detect inconsistencies, we use this simplified property.

order to check operations which compute infinite data structures, e.g., stream generators, we analyze the "productivity" of these operations, i.e., a property which ensures that partial results are produced after a finite amount of time, and check finite approximations of their results.

In order to use these ideas in practice, we integrated this kind of semantic versioning checking into CPM [25], a new package management system for the functional logic language Curry. In this way, we obtain a software package manager which checks semantic versioning and, thus, supports a reliable and also specification-based development of software packages.

This paper is structured as follows. In the next section we briefly survey functional logic programming and features of Curry. Sections 3 and 4 discuss the main features of property-based testing and the Curry package manager CPM. The integration of semantic versioning checking into CPM is shown in Section 5. The techniques to check also possibly non-terminating operations are introduced in Section 6 and their implementation is discussed in Section 7. Before we conclude, we show in Section 8 an important application of our approach: the specification-based development of software systems.

2 Functional Logic Programming and Curry

In this section we briefly review some features of functional logic programming and Curry that are relevant for this paper. More details can be found in surveys on functional logic programming [6, 16] and in the language report [19].

Functional logic languages [6, 16] integrate the most important features of functional and logic languages in order to provide a variety of programming concepts. They support functional programming concepts like higher-order functions and lazy evaluation as well as logic programming concepts like non-deterministic search and computing with partial information. The declarative multi-paradigm language Curry [19] is a functional logic language with advanced programming concepts.

The syntax of Curry is close to Haskell [27], i.e., type variables and names of defined operations usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. $\alpha \to \beta$ denotes the type of all functions mapping elements of type α into elements of type β (where β can also be a functional type, i.e., functional types are "curried"). The application of an operation f to e is denoted by juxtaposition ("$f\ e$").

In addition to Haskell, Curry allows *free* (*logic*) *variables* in rules and initial expressions. Function calls with free variables are evaluated by a possibly non-deterministic instantiation of demanded arguments.

▶ **Example 1.** The following simple program shows the functional and logic features of Curry. It defines the well-known list concatenation and an operation that returns some element of a list having at least two occurrences:

```
(++) :: [a] → [a] → [a]        someDup :: [a] → a
[]     ++ ys = ys              someDup xs | xs == _++[x]++_++[x]++_
(x:xs) ++ ys = x : (xs ++ ys)             = x    where x free
```

Since "`++`" can be called with free variables in arguments, the condition in the rule of `someDup` is solved by instantiating `x` and the anonymous free variables "`_`" to appropriate values (i.e., expressions without defined functions) before reducing the function calls. This corresponds to narrowing [28], but Curry narrows with possibly non-most-general unifiers to ensure the optimality of computations [3]. Curry is a non-strict language, i.e., derivation steps are performed at outermost positions, which supports computations with infinite data structures [20]. We do not recapitulate the details of the operational semantics which can be

found in [1]. When we later consider evaluations of expressions, we denote by "$\to$" the one step outermost derivation relation and by "$\overset{*}{\to}$" its reflexive-transitive closure.

Note that `someDup` is a *non-deterministic operation* since it might deliver more than one result for a given argument, e.g., the evaluation of `someDup [1,2,2,1]` yields the values `1` and `2`. Non-deterministic operations, which can formally be interpreted as mappings from values into sets of values [14], are an important feature of contemporary functional logic languages. Hence, Curry has also a predefined *choice* operation:

```
x ? _  =  x
_ ? y  =  y
```

Thus, the expression "`0 ? 1`" evaluates to `0` and `1` with the value non-deterministically chosen.

A *functional pattern* [4] is a pattern in the left-hand side of a rule containing defined operations (and not only data constructors and variables). Such a pattern abbreviates the set of all standard patterns to which the functional pattern can be evaluated (by narrowing). For instance, we can rewrite the definition of `someDup` as

```
someDup (_++[x]++_++[x]++_) = x
```

Functional patterns are a powerful feature to express arbitrary selections in tree structures, e.g., as shown for processing XML documents in [15].

Curry has also features which are useful for application programming, like *set functions* [5] to encapsulate non-deterministic computations, *default rules* [8] to deal with partially specified operations and negation, and standard features from functional programming, like modules or monadic I/O. Other features are explained when they are used in the following.

3 Property-based Testing and CurryCheck

Property-based testing is a useful technique to improve the reliability of software packages. Basically, properties are Boolean expressions parameterized over input data. Concrete input data is automatically generated by property-based test tools which evaluate the properties on these inputs. For instance, QuickCheck [12], PropEr [26], or PrologCheck [2] generate test inputs in a random manner, whereas SmallCheck [29], GAST [21], or EasyCheck [11] perform a systematic enumeration of test inputs so that, for finite input domains, they can actually verify properties.

CurryCheck [17] is a property-based test tool for Curry which automates the test process. CurryCheck is based on EasyCheck and extracts and tests all properties contained in a source program. A property is a top-level entity with result type `Prop` and an arbitrary number of inputs. For instance, if we add to the program of Example 1 the property

```
concIsAssoc :: [Int]  →  [Int]  →  [Int]  →  Prop
concIsAssoc xs ys zs = (xs++ys)++zs -=- xs++(ys++zs)
```

and run CurryCheck on this program, the associativity property of list concatenation is tested by systematically enumerating lists of integers for the variables `xs`, `ys`, and `zs`. The property "`-=-`" has the type `a → a → Prop` and is satisfied if both arguments have a single identical value.

To check laws involving non-deterministic operations, one can use the property "`<~>`" which is satisfied if both arguments have identical result sets. For instance, consider the following definition of a permutation of a list (which exploits a functional pattern to select some element in the argument list):

```
perm (xs++[x]++ys) = x : perm (xs++ys)
perm []            = []
```

The requirement that permutations do not change the list length can be expressed by the property

```
permLength xs = length (perm xs) <~> length xs
```

Since the left argument of "`<~>`" evaluates to many (identical) values, the set-based interpretation of "`<~>`" is relevant here. This is reasonable since, from a declarative programming point of view, it is irrelevant how often some result is computed.

Now consider an alternative definition of permutations which non-deterministically inserts the first element into a permutation of the remaining elements:

```
permIns []     = []
permIns (x:xs) = insert x (permIns xs)

insert x (xs++ys) = xs++[x]++ys
```

In order to check whether both definitions of permutations compute identical results, we (successfully) test the following property:

```
permSameAsPermIns xs = perm xs <~> permIns xs
```

4 CPM: The Curry Package Manager

The Curry Package Manager CPM[3] [25] is a tool to distribute and install Curry software packages and manage version dependencies between them. Essentially, a *package* consists of one or more Curry modules and a package specification, a file in JSON format containing the package's metadata. Beyond some standard fields, like author, name, or synopsis, the metadata of each package contains the version number of the package (in semantic versioning format, see above) and a list of dependency constraints. A *dependency constraint* consists of the name of another package and a disjunction of conjunctions of version relations, which are comparison operators (`<`, `<=`, `>`, `>=`, `=`) together with a version number. Conjunctions are separated by commas, and disjunctions are separated by `||`. Hence, the dependency constraint

```
"B" : ">= 2.0.0, < 3.0.0 || > 4.1.0"
```

expresses the requirement that the current package depends on package `B` with major version `2` or in a version greater than `4.1.0`.

CPM has various commands to manage the set of all packages and install and upgrade individual packages. CPM uses a central index of all known packages and their versions. A user can download a local copy of this index and also add other local packages and versions to this index. To install a package, CPM tries to resolve all dependency constraints of the current package and all dependent packages. This is a classic constraint satisfaction problem and CPM uses a lazy functional approach based on [24] to solve all dependency constraints and find appropriate package versions. If there is a solution to these constraints, CPM automatically installs all required packages. If there are several possible versions of some

[3] `http://curry-language.org/tools/cpm`

package to install, CPM uses the newest one. CPM also supports upgrading packages, i.e., to replace already installed packages by newer versions, if possible. The details of these processes are outside the scope of this paper and are described in [25].

CPM adheres to the semantic versioning standard as sketched in Section 1. Thus, if there are two versions of a package with identical major version numbers, they should have compatible APIs, i.e., all public data types and operations in the exported modules[4] occurring in both package versions must have identical type signatures and behavior, and new public operations can be added only if the minor version number is increased. CPM supports the automated checking of this principle by the `diff` command. For instance, to compare the current package to a previous version `1.2.4` of the same package, a package developer can invoke the command

```
> cpm diff 1.2.4
```

This starts a complex comparison process which is described in the next section. Depending on the outcome of this API comparison, the current package can be added to the central CPM index.

5 Semantic Versioning Checking

Semantic versioning checking is the process to compare the APIs of two versions of some package and report possible violations according to the semantic versioning standard. In our context, the API of a package is the set of all public data types and operations occurring in the exported modules of this package. To accomplish this task, the semantic versioning checker integrated in CPM performs the following steps:

1. The signatures of all API data types and operations occurring in both packages are compared. If there are any syntactic differences and the major version numbers of the packages are identical, a violation is reported.
2. If there is some API entity f occurring in version $a_1.b_1.c_1$ but not in version $a_2.b_2.c_2$, then a violation is reported if a_1 and a_2 are identical but b_1 is not greater than b_2.
3. If the major version numbers of the packages are identical, then, for all API operations occurring in both package versions, the behavior of both versions of such an operation is compared (see below for more details about this comparison). If any difference is detected, a violation is reported.

To compare the behavior of some operation f defined in versions v_1 and v_2 of some package, the code of both packages is copied and all modules of these packages (and all packages on which these packages depend) are renamed with the version number as a prefix. For instance, a module `Mod` occurring in package version `1.2.3` is copied and renamed into module `V_1_2_3_Mod`. Thus, if there is a unary operation `f` occurring in module `Mod` in package versions `1.2.3` and `1.2.4` to compare, one can access both versions of this operation by the qualified name `V_1_2_3_Mod.f` and `V_1_2_4_Mod.f`. After copying all modules, CPM generates a new "comparison" module which contains the following code:

```
import qualified V_1_2_3_Mod
import qualified V_1_2_4_Mod

check_Mod_f x = V_1_2_3_Mod.f x <~> V_1_2_4_Mod.f x
```

[4] A package specification can also declare a subset of all modules as "exported" so that only operations in these modules can be used by other packages. If this is not explicitly declared, all modules of the package are considered as exported.

If this is passed to CurryCheck and the property is satisfied for all generated test inputs, we have some confidence about the semantic equivalence of `f` in both packages (although full confidence requires the proof of a more complex property [7, 9]). This approach works under the following assumptions:

1. The input and result types of `V_1_2_3_Mod.f` and `V_1_2_4_Mod.f` are identical.
2. The operations to be compared are terminating on all input values.

Since these conditions might not be satisfied in practice, we develop (partial) solutions to it. To see an example where the first condition is not satisfied, consider the following excerpt of the library `Day` dealing with weekdays:

```
data Weekday = Monday | Tuesday | ... | Sunday

nextDay :: Weekday → Weekday
...
```

Since the type `Weekday` is locally defined, copying and renaming two versions of this library for semantic versioning checking results in two different `Weekday` types so that both versions of `nextDay` have incompatbile argument and result types. Thus, to generate a property to compare both versions, CPM generates a bijective mapping between both renamed types:

```
t_Weekday :: V_1_2_4_Day.Weekday → V_1_2_3_Day.Weekday
t_Weekday V_1_2_4_Day.Monday  = V_1_2_3_Day.Monday
t_Weekday V_1_2_4_Day.Tuesday = V_1_2_3_Day.Tuesday
...
```

This mapping must exist (otherwise, semantic versioning is syntactically violated) and it allows to compare both versions of `nextDay` with the following property:

```
check_Day_nextDay :: V_1_2_4_Day.Weekday → Prop
check_Day_nextDay x = t_Weekday (V_1_2_4_Day.nextDay x)
                      <~> V_1_2_3_Day.nextDay (t_Weekday x)
```

If our second assumption (termination of the operations to be compared) is not satisfied, the behavior checker might not terminate. Obviously, this should be avoided. Therefore, we analyze the operations to be compared before the comparison properties are generated. As a simple approach, one can approximate the termination behavior of these operations, e.g., by comparing the argument sizes in recursive calls [22]. For this purpose, we used the Curry analysis framework CASS [18] to implement a simple termination analysis which checks the arguments of direct recursive calls of an operation. If all these calls contain at least one syntactically smaller argument (since we consider only algebraic data types for this purpose, there are no infinite chains of size-decreasing values) and all dependent operations are terminating, the operation is classified as terminating. We can use this analysis to check only those operations which are definitely terminating and emit warnings about the remaining unchecked operations. Although there are many opportunities to improve the termination analyzer, it can only approximate the termination property. Therefore, CPM also accepts specific pragmas where the programmer can annotate operations as terminating. For instance, CPM will consider the following operation as terminating and, thus, includes it in semantic versioning checking:

```
{-# TERMINATE -#}
mcCarthy :: Int → Int
mcCarthy n = if n<=100 then mcCarthy (mcCarthy (n+11))
                       else n-10
```

Although this is reasonable to increase the number of operations considered in semantic versioning checking, an important class of operations is still excluded: operations that are intentionally non-terminating since they generate infinite data structures. A method to check such operations will be presented in the next section.

6 Checking Non-terminating Operations

It is well-known that lazy evaluation is a useful programming feature to increase modularity by separating producers and consumers of data [20]. Typically, data producers are operations which generate infinite structures, like the following operations which generate infinite lists of ascending integers starting from the argument:

```
ints :: Int → [Int]          ints2 :: Int → [Int]
ints n = n : ints (n+1)      ints2 n = n : ints2 (n+2)
```

Although these operations compute infinite lists of a different shape, this difference cannot be detected by the property

```
checkInts x = ints x <~> ints2 x
```

due to its non-termination. Since such operations are actually used in non-strict languages, semantic versioning checking should be supported for them in some way.

How can we state that `ints` and `ints2` have a different behavior? If we consider the computed result equivalence of operations introduced in Sect. 1, there is no difference since neither `ints` nor `ints2` evaluate to some value (an expression without operation symbols). Therefore, a simple strategy like running CurryCheck with a time limit would not show any difference in the values computed by `ints` nor `ints2`. We need another way to compare the behavior of these operations. Thus, we use a more general notion of equivalence of operations in non-strict functional logic languages proposed in [7], also called "contextual equivalence" in [9]. It expresses the idea that two operations are equivalent if they can be replaced by each other in any context without changing the produced values.

▶ **Definition 2** (Equivalent operations [7]). Let f_1, f_2 be operations of the same type. f_1 is *equivalent* to f_2 iff, for any expression E_1 and value v, E_1 evaluates to v iff E_2 evaluates to v, where E_2 is obtained from E_1 by replacing any occurrence of f_1 with f_2.

Since equivalence in this sense implies computed result equivalence, counter-examples found by the method introduced in Sect. 5 are also counter-examples to the equivalence of operations. Moreover, `ints` and `ints2` are not equivalent w.r.t. Def. 2: `head (tail (ints 0))` evaluates to `1` but `head (tail (ints2 0))` evaluates to `2`. To detect such differences, we put the operations into some context where only a finite outermost part is computed. In our example, we define an operation that limits the length of a list. Since the length should be limited with non-negative numbers, we define Peano numbers with the constructors `Z`(ero) and `S`(uccessor):

```
data Nat = Z | S Nat
```

We limit potentially infinite lists to some length provided as a `Nat` argument:

```
limitList :: Nat → [Int] → [Int]
limitList Z     _      = []
limitList (S n) []     = []
limitList (S n) (x:xs) = x : limitList n xs
```

Now we can check the observable equivalence of `ints` and `ints2` by the following property:

```
limitCheckInts n x = limitList n (ints x) <~> limitList n (ints2 x)
```

CurryCheck finds a counter-example for the input arguments `n=(S (S Z))` and `x=1`.

Since the list length limit is an input parameter to the property, this property is sufficient to detect observable differences between such infinite lists. A formal result about the soundness and completeness of limited property checking will be presented below. Before we have to discuss some conditions required for this method.

In order to ensure the termination of property checking, a depth restriction is not sufficient in general. For instance, when checking the equivalence of the operations

```
loop n = loop (n+1)              loop2 n = loop2 (n+2)
```

a depth limit would not avoid the non-terminating evaluations of `loop` and `loop2`. This is due to the fact that the evaluation of these operations do not produce a constructor-rooted term after finitely many steps. To exclude this kind of operations, we define the class of productive operations (for the sake of simplicity, we consider unary operations only, but all definitions and results can be extended to operations with more than one argument):

▶ **Definition 3** (Productive operations). An operation f is called *root-productive* if, for all values t, there is no infinite derivation

$$f\ t \to e_1 \to e_2 \to \cdots$$

where each e_i is operation-rooted. An operation f is called *productive* if it is root-productive and, for all values t and derivations $f\ t \overset{*}{\to} e$, all operations in e are productive.

For instance, `loop` and `loop2` are not productive whereas `ints` and `ints2` are productive (remember that "$\to$" denotes the outermost reduction relation of Curry). Obviously, terminating operations are productive but not vice versa.

If all operations in an expression are productive and we limit the result depth of evaluating this expression, as done in the property `limitCheckInts` above, all evaluations are terminating. Thus, if we restrict semantic versioning checking to productive operations and limit the depth of the results, it is always terminating. To formalize this method, we define a limit operation for some data type τ as follows (for the sake of simplicity, we consider monomorphic data types here; the extension to polymorphic type constructors will be discussed later):

▶ **Definition 4** (Limit operation). Let τ be some type defined by

```
data τ = C1 τ11 ... τ1n1 | ... | Ck τk1 ... τknk
```

Then the *limit operation* for type τ is defined as follows:

```
limitτ :: Nat → τ → τ
limitτ Z     _              = cτ        -- cτ is some ground value of type τ
limitτ (S n) (C1 x1 ... xn1) = C1 (limitτ11 n x1) ... (limitτ1n1 n xn1)
...
limitτ (S n) (Ck x1 ... xnk) = Ck (limitτk1 n x1) ... (limitτknk n xnk)
```

The operation `limitList` defined above is an example of a limit operation for lists of integers. The definition assumes that there is always a ground value c_τ (i.e., a constructor term without variables like a constant) of type τ. This assumption might not be satisfied for data types that do not have finite values, as

```
data ByteStream = Cons Byte ByteStream
```

In this case, there is no ground value which can be used as a result of `limitByteStream Z _`. However, we could extend this data type with a new constant

```
data ByteStream = Cons Byte ByteStream | EmptyByteStream
```

and define

```
limitByteStream Z _ = EmptyByteStream
```

Note that this data type extension is similarly to the representation of failures when compiling functional logic programs into purely functional programs [10] so that it does not change the set of computed values.

The termination of semantic versioning checking with limit operations for productive operations is a consequence of the following result:

▶ **Proposition 5.** *Let* `limit`τ *be a limit operation,* n *some* `Nat` *value, and* e *an expression of type* τ *which contains only productive operations. Then all derivations of* `(limit`τ n e`)` *are finite.*

Now we can state the soundness and completeness of checking the equivalence of operations with limit operations. Soundness means that every counter-example found by limited equivalence checking shows that the considered operations are not equivalent:

▶ **Proposition 6** (Soundness of limited equivalence checking)**.** *Let* f_1 *and* f_2 *be operations of type* $\tau \to \tau'$ *and* `limit`τ' *be a limit operation for type* τ'*. If there are values* n, x, y *such that* `limit`τ' n $(f_1\ x)$ *evaluates to* y *but* `limit`τ' n $(f_2\ x)$ *does not evaluate to* y*, then* f_1 *and* f_2 *are not equivalent.*

Completeness of equivalence checking with limit operations means that one can always find a counter-example for non-equivalent operations (if we search long enough for appropriate inputs). However, this is not the case in general due to partially defined operations. For instance, consider a slightly modified variant of the `ints` operations where the generated lists also include elements `head []` whose evaluation leads to a failure:

```
fints n = head [] : n : fints (n+1)    fints2 n = head [] : n : fints2 (n+2)
```

Since the evaluation of `(limitList` n `fints)` fails and does not produce any value for non-zero n, a counter-example to the equivalence of `fints` and `fints2` is not generated. Fortunately, generators of infinite structures are in practical programs totally defined, i.e., reducible on all ground constructor terms. For such operations, completeness is ensured:

▶ **Proposition 7** (Completeness of limited equivalence checking)**.** *Let* f_1 *and* f_2 *be totally defined operations of type* $\tau \to \tau'$ *and* `limit`τ' *be a limit operation for type* τ'*. If* f_1 *and* f_2 *are not equivalent, then there are values* n, x, y *such that* `limit`τ' n $(f_1\ x)$ *evaluates to* y *but* `limit`τ' n $(f_2\ x)$ *does not evaluate to* y*.*

7 Implementation of Semantic Versioning Checking

Based on the observations discussed so far, we can construct a fully automatic tool for semantic versioning checking as follows. Instead of comparing all operations of two versions, which might not terminate, we consider the following operations:

1. Terminating operations: Since their evaluations are finite on all input values, one can check their behavior on given inputs by comparing the sets of their result values (using the property "`<~>`" of CurryCheck).
2. Productive operations: Since they might produce infinite data structures, their result values cannot be fully compared. Instead, one can check their behavior on given inputs by comparing the results obtained by applying some limit operation to them.

By Propositions 6 and 7, this is a sound and complete method for equivalence checking for totally defined operations. The method is still applicable to partial operations but then it is not ensured that counter-examples are found. On the other hand, every implementation has to limit the number of test inputs to a finite set. Therefore, the theoretical incompleteness of property testing for partially defined operation does not cause a problem in practice.

From a practical point of view, it is more relevant to ensure the termination of the checking tool. This requires to approximate the termination and productivity properties of operations and the generation of limit operations for data types. First, we consider the latter (easier) requirement.

We have already seen the definition of limit operations for monomorphic types like list of integers. This scheme can be extended to polymorphic data types: in this case, we pass limit operations for the polymorphic argument types which are applied to polymorphic arguments. For instance, a limit operation for polymorphic lists can be defined as follows:

```
limitList :: (Nat → a → a) → Nat → [a] → [a]
limitList la Z     _      = []
limitList la (S n) []     = []
limitList la (S n) (x:xs) = la n x : limitList la n xs
```

CPM generates limit operations according to this scheme for all result types of productive operations.

Since termination and productivity are undecidable properties, we approximate these properties with a program analysis and use the Curry analysis framework CASS [18] to implement this analysis. For termination, the size-change principle [22] is a reasonable framework. We implemented only a simplified version of it where all directly recursive calls must have decreasing arguments. Although this is not as powerful as the general framework, it is a good starting to implement our approach. Of course, one can make the termination analysis more precise by implementing sophisticated termination methods or use, if free variables occur in right-hand sides, specific termination methods for functional logic programming [23].

The approximation of productivity is less explored than termination. A notion of productivity has been investigated in the area of term rewriting systems (TRSs), e.g., [13, 30]. However, the focus is different there. Productivity in TRS means that there is some reduction sequence that produces an outermost constructor in finitely many steps, whereas productivity in our sense means that *all* outermost reduction sequences cannot go on forever without producing outermost constructors, which is important to ensure the termination of our checking procedure (see Prop. 5). This difference becomes relevant for non-deterministic computations where it is not sufficient for our purpose that some computation branch produces constructors. Furthermore, terminating operations are always productive in our sense.

We approximate productivity by considering the *top-level operation calls* (*tlo*) of some operation. For each operation f, the set $tlo(f)$ is defined by (we denote by $\overline{o}$ a sequence of

objects $o_1 \ldots o_n$):

$$tlo(f) = \{g \mid \exists \text{ values } \overline{t}, \overline{s} \text{ and some derivation } f\ \overline{t} \xrightarrow{*} g\ \overline{s}\}$$

Similarly, we define the set $tlc(f)$ of *top-level calls inside constructors* as all operations occurring outermost in a constructor derived from a call to f. For instance, $tlo(\texttt{ints}) = \{\}$ and $tlc(\texttt{ints}) = \{\texttt{ints}\}$. These sets can be over-approximated by a fixpoint computation on the program rules. Then we classify an operation f as productive if

1. $f \notin tlo(f)$,
2. all operations in $tlo(f)$ and $tlc(f)$ are productive, and
3. all other operations which might occur in derivations of f are terminating.

Hence, the operation `ints` is productive (note that no other operation occur in a derivation of `ints`) whereas `loop` is not productive (since it violates the first requirement). Productive operations occurring in arguments of other operations lead to non-productive operations. To understand this strong requirement, consider the following operation (the standard operation `filter` removes all elements in the second argument list which do not satisfy the predicate provided in the first argument):

```
natsWith p = filter p (ints 0)
```

Although `ints` is productive, the productivity of `natsWith` depends on the value of its argument: if the argument is the predicate `(>0)`, it always produces outermost constructors, but if the argument is the predicate `(<0)`, it loops without producing any constructor. One might improve our weak but safe approximation for particular cases, but our current approximation is still useful in practice. If this approximation does not classify an operation as productive, the package developer can add a pragma to tell CPM that an operation is productive. For instance, one can compute the list of all prime numbers by the sieve of Eratosthenes, but the productivity depends on the fact that there are infinitely many prime numbers. Hence, Euclid would add the following pragma:

```
{-# PRODUCTIVE -#}
primes = sieve (ints 2)
  where sieve (p:xs) = p : sieve (filter (\x  → mod x p > 0) xs)
```

The effectiveness of the termination and productivity analysis depends on the programs under consideration. In order to evaluate our approach, we applied our analysis to the largest library available in Curry distributions: the standard prelude which contains the definitions of operations that are available to any Curry program. The prelude defines 126 operations (plus 30 I/O actions which are excluded from automated property checking due to the problem of guessing appropriate input values like file names, see also [17]). Our analysis shows that 112 operations are terminating, 11 operations are productive, and the remaining three operations might be non-terminating so that they should not be checked. A closer look at the latter operations shows that one operation is actually non-terminating (the prelude operation `until` which implements a loop which might not terminate), whereas two other are actually terminating but use other productive operations so that their termination cannot be shown by our criteria. Nevertheless, the precision is encouraging and there are non-terminating but productive operations that can be checked thanks to our techniques.

Note that our approach to equivalence checking for non-terminating operations is also applicable to non-deterministic operations. For instance, if we define lists of ascending integers in a non-deterministic manner by

```
ndints n = n : (ndints (n+1) ? (n+1) : ndints (n+2))
```

then our check with limit operations succeeds: although `ndints` non-deterministically evaluates to several infinite lists, all of them are identical to the list computed by `ints`.

The inclusion of non-determinism is relevant for packages that use logic programming features. Apart from this, it is also useful to support specification-based software development as discussed in the following section.

If a previous version of the package contains a bug in the implementation of some operation, it is meaningless to compare the operation of the current version against the previous version. For this purpose, there is a pragma to tell CPM to drop the checking of some operation:

```
{-# NOCOMPARE -#}
f ... = ...code with bug fixes...
```

If the current version is accepted to the CPM repository, this annotation should be removed.

8 Specification-based Software Development

The advantage of using functional logic languages like Curry as a wide-spectrum language for software development is discussed in [7]. There it is shown that functional logic programming features are useful to write comprehensive, executable specifications as well as more efficient implementations. Since specifications as well as implementations are written in the same language, specifications can be used as run-time assertions for implementations or their equivalence can be statically checked by property testing [17]. For this purpose, [7] proposed to define the specification of some operation f by some operation with the name f`'spec`. CurryCheck uses this name convention for generating and testing equivalence properties.

With the use of packages, one can structure this development process even better. The idea is to write the specification of the operations to be developed in a first version of a package, i.e., the package n`.0.0` (where n is a major version number) contains the specification. For instance, if we want to develop a package `sort` with sorting operations, we could define a specification of sorting a list in version `1.0.0` by

```
sort (xs++[x,y]++ys) | x>y = sort (xs++[y,x]++ys)
sort'default xs = xs
```

In the first rule, a functional pattern is used to select some arbitrary pair of elements that are swapped to improve the ordering of the list. The second default rule [8] is applicable when the first rule cannot be applied, i.e., when all elements are in the correct order.

Note that this specification is non-deterministic, i.e., its execution might return a sorted list more than one time. However, this is not relevant if we use it for semantic versioning checking since there we compare the result *sets* of two versions of an operation. Thus, if we implement a deterministic and more efficient sorting operation in version `1.0.1` of the package `sort`, we can use the semantic versioning checker to automatically test the new implementation against its specification.

9 Conclusions and Related Work

We have presented, to the best of our knowledge, the first semantic versioning checker that is integrated in a software package manager. In order to make the checking process fully automatic, it is necessary to ensure the termination of the checker. Therefore, the checker analyzes the termination and productivity behavior of all operations and generates appropriate properties that will be tested by CurryCheck. With these methods, most operations can be automatically checked, even operations which produce infinite data structures. Since the

checker can only approximate the run-time behavior of operations, a package developer can also insert annotations to increase the number of checked operations.

We developed this framework for the functional logic programming language Curry, but most of the ideas can also be transferred to other declarative (purely functional or purely logic) languages. Nevertheless, the use of Curry also supports the specification-based development of software since specifications can often be adequately expressed in a non-deterministic way.

Although semantic versioning is recommended in many package managers and used in software projects, there are almost no tools to help the developer to check semantic properties of different package versions. An exception is the Elm package manager[5] which performs semantic versioning checks based on purely syntactic API comparisons. Thus, it can not detect semantic differences when API types are unchanged, like replacing a decrement by an increment operation.

We have demonstrated that declarative programming in combination with property testing tools is a good basis for this task. Hence, all kinds of languages with property testing tools are appropriate for this technique, e.g., Haskell with QuickCheck [12], Prolog with PrologCheck [2], or Erlang with PropEr [26]. For a fully automatic tool that can be integrated into the infrastructure of package managers, it is important to ensure the termination of the checking process. For this purpose, one needs methods to ensure the termination of the programs under consideration. For non-strict languages, one should also provide methods to compare operations which produce infinite structures. For this purpose, we defined the notion of productive operations and a method to approximate this property. One can find similar notions in term rewriting systems (e.g., [30, 13]) but with a slightly different focus.

For future work, we plan to integrate better techniques for termination checking, since this would enlarge the class of checked operations. Furthermore, it would be interesting to add methods for equivalence checking without property testing. An obvious method is to check the structural equivalence of program code. This is useful for operations which are unchanged or only reformatted in two versions of a package. One might also use an abstract semantics to infer equivalences in functional logic programs, as done in [9]. Another idea is to combine property testing with theorem proving to develop and store proofs of properties. Initial ideas are supported by CurryCheck [17] but their integration for semantic versioning checking has to be explored.

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 C. Amaral, M. Florido, and V. Santos Costa. PrologCheck - property-based testing in Prolog. In *Proc. of the 12th International Symposium on Functional and Logic Porgramming (FLOPS 2014)*, pages 1–17. Springer LNCS 8475, 2014. doi:10.1007/978-3-319-07151-0_1.

3 S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000. doi:10.1145/347476.347484.

4 S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.

5 S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declar-*

[5] http://elm-lang.org/

ative Programming (PPDP'09), pages 73–82. ACM Press, 2009. doi:10.1145/1599410.1599420.

6 S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010. doi:10.1145/1721654.1721675.

7 S. Antoy and M. Hanus. Contracts and specifications for functional logic programming. In *Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012)*, pages 33–47. Springer LNCS 7149, 2012. doi:10.1007/978-3-642-27694-1_4.

8 S. Antoy and M. Hanus. Default rules for Curry. *Theory and Practice of Logic Programming*, 17(2):121–147, 2017. doi:10.1017/S1471068416000168.

9 G. Bacci, M. Comini, M.A. Feliú, and A. Villanueva. Automatic synthesis of specifications for first order Curry. In *Principles and Practice of Declarative Programming (PPDP'12)*, pages 25–34. ACM Press, 2012. doi:10.1145/2370776.2370781.

10 B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011. doi:10.1007/978-3-642-22531-4_1.

11 J. Christiansen and S. Fischer. EasyCheck - test data for free. In *Proc. of the 9th International Symposium on Functional and Logic Programming (FLOPS 2008)*, pages 322–336. Springer LNCS 4989, 2008.

12 K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP'00)*, pages 268–279. ACM Press, 2000.

13 J. Endrullis and D. Hendriks. Lazy productivity via termination. *Theoretical Computer Science*, 412(28):3203–3225, 2011. doi:10.1016/j.tcs.2011.03.024.

14 J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.

15 M. Hanus. Declarative processing of semistructured web data. In *Technical Communications of the 27th International Conference on Logic Programming*, volume 11, pages 198–208. Leibniz International Proceedings in Informatics (LIPIcs), 2011. doi:10.4230/LIPIcs.ICLP.2011.198.

16 M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013. doi:10.1007/978-3-642-37651-1_6.

17 M. Hanus. CurryCheck: Checking properties of Curry programs. In *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*. Springer LNCS 10184, 2016.

18 M. Hanus and F. Skrlac. A modular and generic analysis server system for functional logic programs. In *Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*, pages 181–188. ACM Press, 2014. doi:10.1145/2543728.2543744.

19 M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at http://www.curry-language.org, 2016.

20 J. Hughes. Why functional programming matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 17–42. Addison Wesley, 1990.

21 P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In *Proc. of the 14th International Workshop on Implementation of Functional Languages*, pages 84–100. Springer LNCS 2670, 2003.

22 C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 81–92, 2001.

23 N. Nishida and G. Vidal. Termination of narrowing via termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 21(3):177–225, 2010. doi:10.1007/s00200-010-0122-4.

24 T. Nordin and A.P. Tolmach. Modular lazy search for constraint satisfaction problems. *Journal of Functional Programming*, 11(5):557–587, 2001. doi:10.1017/S0956796801004051.

25 J. Oberschweiber. A package manager for Curry. Master's thesis, University of Kiel, 2016.

26 M. Papadakis and K. Sagonas. A PropEr integration of types and function specifications with property-based testing. In *Proc. of the 10th ACM SIGPLAN Workshop on Erlang*, pages 39–50, 2011. doi:10.1145/2034654.2034663.

27 S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

28 U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 138–151, Boston, 1985.

29 C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *Proc. of the 1st ACM SIGPLAN Symposium on Haskell*, pages 37–48. ACM Press, 2008.

30 H. Zantema and M. Raffelsieper. Proving productivity in infinite data structures. In *Proc. 21st International Conference on Rewriting Techniques and Applications (RTA 2010)*, volume 6 of *LIPIcs*, pages 401–416. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010. doi:10.4230/LIPIcs.RTA.2010.401.