CPM: A Declarative Package Manager with Semantic Versioning

– System Description –

Michael Hanus Jonas Oberschweiber

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

Abstract. We present CPM, a package manager for the declarative multi-paradigm language Curry. Although CPM inherits many ideas from package managers for other programming languages, a distinguishing feature of CPM is its support to check the rules of semantic versioning, a convenient principle to associate meaningful version numbers to different software releases. Although the correct use of version numbers is important in software package systems where packages depend on other packages with specific releases, it is often used as an informal agreement but usually not checked by package managers. CPM is different in this aspect: it provides support for checking the semantic requirements implied by the semantic versioning scheme. Since these semantic requirements are undecidable in general, CPM uses the property-based testing tool CurryCheck to check the semantic equivalence of two different versions of a software package. Thus, CPM provides a good compromise between the use and formal verification of the semantic versioning rules.

1 Introduction

Complex software systems are usually not built from scratch but re-use various components. To structure such 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 complex package depend on other packages. Since packages change 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 an appropriate version of a package. Finding them and managing these dependencies is a non-trivial problem. As a solution to it, 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] a version number consists of major, minor, and patch number, separated by dots, and an optional pre-release specifier consisting of alphanumeric characters and hyphens

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

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, `0.1.2` < `0.3.1` < `1.1.2-beta` < `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, 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 `1.4.1` of package `B`, one can specify that `A` depends on `B` in a version greater than or equal to `1.4.1` but less than `2.0.0`. Thanks to semantics versioning, a package manager can choose newer versions of `B` (as long as they are smaller than `2.0.0`), when they become available, in order to build `A`.

However, semantic versioning requires that, if some operation f is defined in two versions of a package with identical major version numbers, these two definitions are semantically equivalent. 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 Curry package manager CPM.

In order to check the semantic equivalence of a unary operation f defined in versions v_1 and v_2 of some package, one can rename the definitions of f in these packages to f_{v_1} and f_{v_2}, respectively, and check the property $\forall x. f_{v_1}(x) = f_{v_2}(x)$.[2] Ideally, one should prove this property. Since fully automatic proof techniques are available only for limited domains, CPM uses property-based testing instead. Property-based testing automates the checking of properties by random or systematic generation of test inputs. It has been introduced with the QuickCheck tool [6] for the functional language Haskell and adapted to many other languages, like CurryCheck [8] for the functional logic language Curry. Although property-based testing provides no formal guarantees, in practice it is quite successful if the generated input data is well distributed.

In the following, we briefly survey Curry and CurryCheck before we provide an overview of CPM and its implementation of semantic versioning checking.

2 Functional Logic Programming and Curry

Functional logic languages combine the most important features of functional and logic programming in a single language (see [7] for a recent survey). In particular, the functional logic language Curry [11] conceptually extends Haskell

[2] Although this property is necessary, it is not sufficient to ensure semantic equivalence in functional logic programs [5]. Nevertheless, we use it here for the sake of simplicity.

with common features of logic programming, i.e., non-determinism, free variables, and constraint solving. The syntax of Curry is close to Haskell. In addition to Haskell, Curry applies rules with overlapping left-hand sides in a (don't know) non-deterministic manner (whereas Haskell always selects the first matching rule) and allows *free* (*logic*) *variables* in conditions and right-hand sides of rules. Function calls are evaluated lazily where free variables as demanded arguments are non-deterministically instantiated [2].

Example 1. The following simple program shows the functional and logic features of Curry. It defines an operation "++" to concatenate two lists, which is identical to the Haskell encoding. The operation ins inserts an element at some (unspecified) position in a list:

```
(++) :: [a] → [a] → [a]          ins :: a → [a] → [a]
[]     ++ ys = ys                 ins x ys     = x : ys
(x:xs) ++ ys = x : (xs ++ ys)     ins x (y:ys) = y : ins x ys
```

Note that ins is a *non-deterministic operation* since it might deliver more than one result for a given argument, e.g., the evaluation of ins 0 [1,2] yields the values [0,1,2], [1,0,2], and [1,2,0]. Curry has many other features not described here, like monadic I/O and modules as in Haskell, set functions [4] to encapsulate non-deterministic search, and functional patterns [3] to specify complex transformations in a high-level manner. For instance, we can provide an alternative and more compact definition of ins with a functional pattern:

```
ins' x (xs++ys) = xs++[x]++ys
```

3 Property-based Testing with CurryCheck

Property-based testing [6] is a useful technique to improve the reliability of software packages. Basically, properties are expressions parameterized over input data. CurryCheck [8] is a property-based test tool for Curry which automates the tests whether properties hold on various inputs. CurryCheck extracts and tests all properties, i.e., top-level entities with result type Prop, contained in a source program. 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 parameters 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, the requirement that list insertion increments the list length can be expressed by the property

```
insLength x xs = length (ins x xs) <~> length xs + 1
```

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. The semantic equivalence of `ins` and `ins'` defined above can be checked with the property

```
insSameAsIns' x xs = ins x xs <~> ins' x xs
```

4 CPM: The Curry Package Manager

The Curry Package Manager CPM[3] is a tool to distribute and install Curry software packages and manage version dependencies between them. A CPM *package* consists of at least 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) 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. Since CPM uses a central index of all known packages[4] and their versions, an important command is `cpm update` which downloads the newest version of this index. The command `cpm list` shows a table of all packages (sorted by various criteria which can be specified as command options), `cpm search` allows to search for a term within all packages, and `cpm info` shows detailed information of a package.

The command `cpm install` installs a package by resolving all dependency constraints of the current package and all dependent packages. This is a classic constraint satisfaction problem. CPM uses a lazy functional approach based on [12] to solve all dependency constraints and find appropriate package versions. If there is a solution to these constraints, CPM automatically installs local copies of all required packages (either from a cache or by downloading them from a central repository). If there are several possible versions of some package to install, CPM uses the newest one. There is also a command `cpm upgrade` 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 [13]. Since the number of packages in the current CPM index is limited, we tested our dependency resolution algorithm on a large set of packages (the central package index of *npm*, the Node package manager) and obtained acceptable run times on realistic examples (see [13] for details).

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

[4] Currently, CPM manages more than 50 packages and 400 modules.

CPM also supports package testing, documentation, and compilation. The command `cpm test` applies CurryCheck to all source modules of the package (or to some test suite specified in the package's metadata). The command `cpm doc` generates the documentation of a package, i.e., the API documentation (in HTML format) automatically extracted from source programs and, if provided, manuals in PDF format. If the package's metadata specifies a main module and the name of an executable, the package and all its dependencies are compiled by the command `cpm install`, which also installs the generated binary in the `bin` directory of CPM. Hence, complete Curry applications can be wrapped in a package so that they are easily installed by a single command.

As mentioned above, CPM adheres to the semantic versioning standard as sketched in Section 1. CPM supports the automated checking of the rules of semantic versioning by the command `cpm diff`. For instance, to compare the current package to a previous version `1.2.4` of the same package, one can invoke the command

```
> cpm diff 1.2.4
```

This starts a complex comparison process which is described in the next section.

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 the case of Curry, the API of a package is the set of all public data types and operations occurring in the exported modules[5] of this package. The semantic versioning checker of CPM performs the following steps:

1. The signatures of all API data types and operations occurring in both versions of the package 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). A violation is reported if any difference is detected.

The implementation of the first two steps can be achieved by a straightforward syntactic comparison of the packages. To implement step 3, i.e., 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

[5] The metadata of a package can also specify 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.

all packages on which these packages depend) are renamed with the version number as a prefix. For instance, a module `M` occurring in package version `1.2.3` is copied and renamed into module `V_1_2_3_M`. Thus, if there is a unary operation `f` occurring in module `M` 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_M.f` and `V_1_2_4_M.f`. Thus, CPM generates a new "comparison" module which contains the following code:

```
import qualified V_1_2_3_M
import qualified V_1_2_4_M

check_M_f x = V_1_2_3_M.f x <~> V_1_2_4_M.f x
```

Due to the use of the property "`<~>`", CPM can also compare the computed results of non-deterministic operations. If this module 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. This approach works under the following assumptions:

1. The input and result types of `V_1_2_3_M.f` and `V_1_2_4_M.f` are identical.
2. The operations to be compared are terminating on all input values.

Unfortunately, the first assumption is not satisfied if `f` works on a type `T` defined in module `M`, since the comparison module contains two copies of this type: `V_1_2_3_M.T` and `V_1_2_4_M.T`. In order to generate a single property to compare both versions of `f`, CPM generates a bijective mapping between both renamed types

```
t_T :: V_1_2_4_M.T  →  V_1_2_3_M.T
```

This operation can inductively be defined for all data constructors of type `T`, since the structure of `T` must be identical in both versions (otherwise, semantic versioning is syntactically violated). If `f` is of type `T` → `T`, then CPM generates the following property to compare both versions of `f`:

```
check_M_f x = V_1_2_3_M.f (t_T x) <~> t_T (V_1_2_4_M.f x)
```

If the second assumption (termination) is not satisfied, the property tester might not terminate. To avoid this situation, CPM analyzes the operations to be compared before the comparison properties are generated. For this purpose, CPM exploits the Curry analysis framework CASS [10], which provides a simple termination analysis, and generates the above properties only for operations which are definitely terminating. CPM also accepts specific pragmas where the programmer can annotate operations as terminating for cases where the termination checker is not powerful enough.

Unfortunately, this is not sufficient to check operations that are intentionally non-terminating since they generate infinite data structures. In order to check such operations, e.g., stream generators, CPM analyzes the "productivity" of these operations and compare finite approximations of their results. For instance, consider the following operations which generate infinite lists of ascending integers starting from the given argument:

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

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

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

since its evaluation does not terminate. However, both operations are *root-productive*: there is no infinite sequence of evaluation steps which does not produce a constructor at the root. A *productive* operation is one which is root-productive and all operations occurring in derivations of this operations are also productive. Hence, `ints` and `ints2` are productive whereas `loop` defined by

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

is not productive. The productivity property of operations can be approximated by a program analysis with a fixpoint computation on all program rules (see [9] for details).

CPM implements such a program analysis and uses its result to compare also non-terminating but productive operations. For this purpose, it limits the size of the data structures to be compared. For instance, the size of a potentially infinite list can be limited by an operation which has a first "size" argument (represented as a Peano number with the constructors `Z` and `S`):

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

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

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

For this property, CurryCheck finds a counter-example for the input arguments `n=(S (S Z))` and `x=1`. With this approach, CPM can also compare different versions of non-terminating but productive operations. Hence, the overall strategy of CPM to compare two different versions of an operation f is as follows:

1. If f is terminating, the results of both versions are directly compared (with a type mapping, if required).
2. If f is non-terminating but productive, the results of both versions are limited to a given size, where the size parameter is also part of the test inputs.
3. If f is non-terminating and not productive, both versions are not compared and a warning is issued.

If CPM cannot automatically derive the productivity of an operation, the programmer can explicitly annotate operations as productive so that they are checked with the strategy explained above. More details about this analysis and its implementation can be found in [9].

6 Concluding Remarks

We have presented a software package manager for Curry with support for semantic versioning. Although there exist many package managers which use similar versioning schemes, to the best of our knowledge, CPM is the first package manager which provides automated support for semantic versioning checking. The Elm package manager[6] also performs semantic versioning checks but this is based on simple syntactic API comparisons. Hence, it can not detect semantic differences when API types are unchanged, like replacing a decrement by an increment operation.

We have shown that declarative languages in combination with powerful property testing tools are a good basis for automated semantic versioning checking. Hence, our approach can also be transferred to Haskell with QuickCheck [6], Prolog with PrologCheck [1], or Erlang with PropEr [14]. For a fully automatic tool, it is necessary to ensure the termination of the checking process. Although this can be achieved by time limits, more powerful checks require a careful program analysis, as we have done with analyzing the productivity of possibly non-terminating operations.

CPM's semantic versioning checking is a tool that can be used by the package developer to check the changes introduced in a new version of the package. Since it is a fully automatic tool, it can also be used in the workflow to publish new package versions in the central repository of CPM, which is not yet implemented but a topic for future work.

References

1. 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.
2. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
3. 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.
4. 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 Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.
5. 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.
6. 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.
7. 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.
8. 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.
9. M. Hanus. Semantic versioning checking in a declarative package manager. In *Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017)*, OpenAccess Series in Informatics (OASIcs), 2017.

[6] `http://elm-lang.org/`

10. 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.
11. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at `http://www.curry-language.org`, 2016.
12. T. Nordin and A.P. Tolmach. Modular lazy search for constraint satisfaction problems. *Journal of Functional Programming*, 11(5):557–587, 2001.
13. J. Oberschweiber. A package manager for Curry. Master's thesis, University of Kiel, 2016.
14. 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.