CPM: A Declarative Package Manager with Semantic Versioning

Michael Hanus

Kiel University
Programming Languages and Compiler Construction

WFLP 2017

Joint work with Jonas Oberschweiber

Contemporary software systems are complex!

Software packages as building blocks

- several modules with well-defined APIs
- evolve over time (efficiency improvements, new functionality)
  $\rightsquigarrow$ *versioning* with version numbers (`1.4.3`)

Package dependencies

Package A depends on: package B version $\geq$ 1.2.5 $\wedge <$ 2.0.0
package C version $\geq$ 2.3.7 $\wedge <$ 4.0.0

Semantic versioning: <*major*>.<*minor*>.<*patch*>

Version numbers describe semantic properties:

- alternative implementation $\rightsquigarrow$ increase <*patch*>
- extend API $\rightsquigarrow$ increase <*minor*>
- change API $\rightsquigarrow$ increase <*major*>

Semantic Versioning (`www.semver.org`)

Advantages

- describe necessary dependencies
- choose newest packages ($\rightsquigarrow$ package manager)
- upgrade to newer versions without code breaks

Requirements

Packages with identical <*major*> must be *semantically compatible*

Semantic compatibility

- important to support automatic upgrading
- not checked in contemporary package managers

Our proposal:
check it automatically with property-based test tools

Property-based Testing

Properties: tests parameterized over some arguments

Property: List concatenation is associative

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

concIsAssociative xs ys zs = (xs++ys)++zs <~> xs++(ys++zs)
```

Checking declarative properties

- no side effects $\rightsquigarrow$ repeatable tests
- generate input values:
  - random (QuickCheck, PrologCheck, PropEr)
  - systematic enumeration (SmallCheck, GAST)
  - systematic (non-deterministic) guessing (EasyCheck, CurryCheck)

Here: Curry (Haskell syntax, logic features) + CurryCheck [LOPSTR'16]

Property-based Testing with CurryCheck

Non-deterministic list insertion

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

```
> ins 0 [1,2]  ⇝  [0,1,2] ? [1,0,2] ? [1,2,0]
```

Property: insertion increments list length

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

Set-based interpretation relevant:

e_1 `<~>` e_2 $:\Leftrightarrow$ e_1 and e_2 have identical sets of results

Semantic Versioning Checking

Idea:

f defined in module M of some package in versions v_1 and v_2:

1. create renamed modules M_v_1 and M_v_2
2. create new "comparison" module:

```
import qualified M_v1
import qualified M_v2

check_M_f x = M_v1.f x <~> M_v2.f x
```

3. run CurryCheck on this module

Main problem: f might not terminate

use termination analysis
⇝ no check or specific check for productive operations

Productive Operations

Lazy languages supports infinite data structures:

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

ints 0 ⇝ 0 : 1 : 2 : ...
```

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

ints2 0 ⇝ 0 : 2 : 4 : ...
```

```
-- Equivalence testing:
checkInts x = ints x <~> ints2 x  ⇝ no counter example...
```

Non-terminating but productive operations

f productive ⇔ no infinite reduction without producing root-constructors

`ints`, `ints2`: productive

```
loop n = loop (n+1)  -- not productive
```

Checking Productive Operations

Limit the size of values

```
data Nat = Z | S Nat    -- Peano numbers

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

Checking with size limits

```
limCheckInts n x = limList n (ints x) <~> limList n (ints2 x)
```

⇝ CurryCheck finds counter-example: `n=(S (S Z))  x=1`

Proposition [ICLP'17]:

Limited equivalence checking is sound and complete (for total operations) for equivalence checking.

CPM: Curry Package Manager

- tool to distribute and install Curry software packages
- central package index (currently: $>$ 50 packages, $>$ 400 modules)
- package: Curry modules $+$ package specification:
  - metadata in JSON format
  - standard fields: version number, author, name, synopsis,...
  - *dependency constraints*:

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

$\leadsto$ depends on package `B` with major version `2` or in a version greater than `4.1.0`

Some CPM commands

cpm update	download newest version of package index
cpm search	search in package index
cpm install	installs a package (resolve all dependency constraints) with local copies of all required packages
cpm upgrade	re-install with newer package versions
cpm test	run CurryCheck on all source modules

CPM: Semantic Versioning Checking

cpm diff 1.2.4 check current package version against `1.2.4`

Implementation of semantic versioning checking

- rename modules with version numbers
- generate comparison module
- analyze each operation defined in two package versions:
  - terminating: use standard equivalence check
  - non-terminating but productive: use equivalence checks with limits
    $\leadsto$ generate limit operations for each data type
  - otherwise: no check, warning
- program analysis implemented with CASS [PEPM'14]

Curry prelude: 126 operations

Analysis result: 112 terminating, 11 productive, 3 non-terminating

CPM: Semantic Versioning Checking

User annotations to override analysis results:

Annotate terminating operations

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

Annotate productive operations

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

Annotate unchecked operations

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

Conclusions

CPM: Curry Package Manager

- first package manager with semantic versioning checker
  (Elm package manager: purely syntactic API comparison)
- termination important for automatic tool ⇝ program analysis
- productivity: check also non-terminating operations (data generators)
- supports *specification-based software development*
  - package *n*.0.0 contains specification [PADL'12]
  - newer package versions: better implementations
- approach applicable to all kinds of declarative languages
  functional (QuickCheck), logic (PrologCheck), functional-logic (CurryCheck), ...

Future work:

- better termination analysis
- avoid *testing*:
  - check structural equivalence of source code
  - use theorem provers to proof equivalence