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 \( \geq 1.2.5 \land < 2.0.0 \)
- package C version \( \geq 2.3.7 \land < 4.0.0 \)

Semantic versioning: \(<\text{major}>, <\text{minor}>, <\text{patch}>\)
Version numbers describe semantic properties:
- alternative implementation \( \rightsquigarrow \) increase \(<\text{patch}>\)
- extend API \( \rightsquigarrow \) increase \(<\text{minor}>\)
- change API \( \rightsquigarrow \) increase \(<\text{major}>\)
Semantic Versioning (www.semver.org)

Advantages

- describe necessary dependencies
- choose newest packages (∼ 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
**Properties**: tests parameterized over some arguments

**Property**: List concatenation is associative

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

\[
\text{concIsAssociative } xs \ ys \ zs = (xs ++ ys) ++ zs \ <\sim> \ xs ++ (ys ++ zs)
\]

**Checking declarative properties**

- no side effects $\Rightarrow$ 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]
Non-deterministic list insertion

\[
\text{ins} :: \text{a} \rightarrow \text{[a]} \rightarrow \text{[a]}
\]

\[
\text{ins } x \text{ ys} = x : \text{ys}
\]

\[
\text{ins } x \text{ (y:ys)} = y : \text{ins } x \text{ ys}
\]

\[> \text{ins0 [1,2]} \rightsquigarrow [0,1,2] ? [1,0,2] ? [1,2,0]\]

Property: insertion increments list length

\[
\text{insLength } x \text{ xs} = \text{length} \ (\text{ins } x \text{ xs}) <\sim> \text{length} \ \text{xs} + 1
\]

Set-based interpretation relevant:

\[
e_1 <\sim> e_2 :\Leftrightarrow e_1 \text{ and } e_2 \text{ have identical sets of results}
\]
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_{v_1}
   import qualified M_{v_2}
   
   check_M_f x = M_{v_1}.f x <~> M_{v_2}.f x
   ```
3. run CurryCheck on this module

Main problem: $f$ might not terminate

use termination analysis

$\sim$ no check or specific check for productive operations
Lazy languages supports infinite data structures:

\[
\text{ints} :: \text{Int} \rightarrow [\text{Int}] \\
\text{ints} \ n = n : \text{ints} \ (n+1) \\
\text{ints} \ 0 \leadsto 0 : 1 : 2 : \ldots
\]

\[
\text{ints}2 :: \text{Int} \rightarrow [\text{Int}] \\
\text{ints}2 \ n = n : \text{ints}2 \ (n+2) \\
\text{ints}2 \ 0 \leadsto 0 : 2 : 4 : \ldots
\]

\text{-- Equivalence testing:}
\text{checkInts} \ x = \text{ints} \ x <\sim> \text{ints}2 \ x \leadsto \text{no counter example}...

Non-terminating but productive operations

\( f \ \text{productive} \iff \text{no infinite reduction without producing root-constructors} \)

\text{ints, ints2: productive}

\text{loop n = loop (n+1) -- not productive}
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
**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
  - otherwise: no check, warning
- program analysis implemented with CASS [PEPM’14]

**Curry prelude: 126 operations**

Analysis result: 112 terminating, 11 productive, 3 non-terminating
User annotations to override analysis results:

### Annotate terminating operations

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

### Annotate productive operations

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

### Annotate unchecked operations

```haskell
{-# 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 \( \rightsquigarrow \) 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