Safe Functional Mobile Code – CPPCC: Certified Proved-Property-Carrying Code

Zoltán Horváth

University Eötvös Loránd, Budapest, Hungary
Department of General Computer Science
E-mail: hz@inf.elte.hu
Overview

- Mobile expressions (functional code) in Clean - dynamics
- Proving properties of functional programs - dedicated theorem prover: Sparkle
- Property carrying code architecture, type and semantical checks
Clean dynamics

- static type system extended with type Dynamic

- conversion to type Dynamic by function dynamic, example:
  
  ```
  dynamic 3::Int
  ```

- conversion from Dynamic to static type using pattern match,
  example: `unpackInt :: Dynamic -> Int;`

  ```
  unpackInt (n::Int) = n; unpackInt _ = abort "not an integer"
  ```

- mobile code, persistent data: SendDynamic, ReceiveDynamic, ReadDynamic, WriteDynamic
module f
import StdDynamic, StdEnv
:: Tree b = Node b (Tree b) (Tree b) | Leaf

Start world
  #! (ok,world) = writeDynamic (p +++ "function")
  DynamicDefaultOptions dt world
  | not ok = abort "could not write dynamic"
  = (dt,world)

where
  dt = dynamic count_leafs
  p   = "C:\hz\Clean 2.0\Examples\Dynamic"

count_leafs :: (Tree Int) -> Real

count_leafs tree = toReal (count tree 0)

where
  count :: (Tree Int) Int -> Int
  count Leaf n_leafs = inc n_leafs
  count (Node _ left right) n_leafs =
    count left (count right n_leafs)
module v
import StdDynamic, StdEnv
:: Tree a = Node a (Tree a) (Tree a) | Leaf
Start world
  #! (ok,world)= writeDynamic (p +++ "value")
  DynamicDefaultOptions dt world
  | not ok = abort "could not write dynamic"
  = (dt,world)
  where dt = dynamic (Node 99 tree2 tree2)
  tree2 = (Node 2 (Node 1 Leaf Leaf) Leaf)
  p = "C:\hz\Clean 2.0\Examples\Dynamic\"
module apply
import  StdDynamic, StdEnv
:: Tree a = Node a (Tree a) (Tree a)  |  Leaf
Start world
  # (ok,f,world)  = readDynamic (p +++ "function")  world
  | not ok           = abort " could not read"
  # (ok,v,world) = readDynamic (p +++ "value")  world
  | not ok           = abort " could not read"
  # result = apply f v;
  = (result, world);
where
  apply (f :: (Tree Int) -> Real) (v :: (Tree Int)) =  f  v
  apply _ _                                             = abort "unmatched"
p       = "C:\hz\Clean 2.0\Examples\Dynamic\"

Verification of functional programs

- dedicated theorem prover of Clean : Sparkle - part of IDE
- referential transparencies, equational reasoning possible
- example: \( \forall n \in \text{Int}, \forall \alpha, \forall xs \in [\alpha] : \)
  \[ n \neq \text{undef} \rightarrow \text{take} \; n \; xs + + \text{drop} \; n \; xs = xs \]

  take :: Int ![a] -> [a]
  \[
  \text{take} \; n \; [x:xs] = \begin{cases} 
    [x: \text{take} \; (n-1) \; xs] & \text{if } n > 0 \\
    [] & \text{otherwise}
  \end{cases}
  \]
  drop n [x:xs] = \begin{cases} 
    \text{drop} \; (n-1) \; xs & \text{if } n > 0 \\
    [x:xs] & \text{otherwise}
  \end{cases}

- Proof: using 8 tactics of 42: structural induction on xs,
  introduce (deduction rule for implication), reduce (substitution of definition),
  reflexivity, split cases, definedness (contradiction with definedness), rewrite goal (using hypothesis), contradiction
Property carrying code architecture

Requirements: mobile code does not use to much resources, read or modify data unauthorised, etc. Does do its task.

- Full dynamic-time code verification just before the application of the code (static, structural and type correctness verification: well-formedness, data-flow analysys for illegal memory access, type of instruction arguments, etc.)
- trusting the code producer unconditionally (with using a certificate mechanism, to check identity),
- trusting code integrity and performing run time pattern match for types.
- A hybrid model of safe mobile code exchange: Certified Proven Property Carrying Code. Minimal run-time overhead but proved correct code.
Further semantic requirements, examples

Insertion into a sorted list (safety-critical, time-critical):

- the result of the operation is such that every element of the list is either the first element or an element that is greater than or equal to the previous element,
- termination
- upper bound for memory usage
- upper bound for execution time
The Certified Proved-Property-Carrying Code architecture (CPPCC) Three main parties involved in the scenario:

1. producer of the mobile code, adds proofs of properties

2. receiver, executes code only after safety checks which ensure that the code satisfies the requirements specified in the receiver’s code,

3. certifying authority, reduce the work-load of the receiver, does static-time.