Last login: IPA fall'03, November 19th, 2003.

You have new mail.

Arjen van Weelden <arjenw@cs.kun.nl>
Rinus Plasmeijer <rinus@cs.kun.nl>

Esther:> Composition of compiled code with a functional shell

Esther:>

Overview

- Motivation
  - Why an O/S and shell in a funct. lang.
- Dynamics in Clean (previous talk)
  - O/S overview
- Our functional shell Esther
  - A type checking and code combining shell
  - Features and demonstration
  - Implementation
- Conclusions

Motivation

- Modern programming languages:
  - High level, especially functional ones
  - Abstraction: functions, polymorphism, overloading, generic progamming
  - Composition: application, higher-order functions
  - Verification: strong type checking, type inference, proofs
  - Clean and Haskell: referential transparent, no (unexpected) side effects, types warn about 'dangerous' functions

Motivation

- Introduce functional concepts to O/S
  - No (accidental) side effects, stronger type checking, no (void) pointers
  - More analysis/checking, less run-time errors
- Resulted in a prototype functional micro-kernel for process management
  - Type safe communication of <u>any</u> value (*and code*) of <u>any</u> type (using Clean's dynamics)
- Functional shell (this presentation)
- And a typed file system in development
  - Files have types, executables are functions

Motivation

- Shell/command line languages
  - Scripting, interpreted execution
  - No type checking: all streams of characters
  - Limited composition: pipe-lining
  - No higher-order programs (except maybe ***Es***)
  - Only simple and specialized syntactical constructs (except ***ScSh***)
- Programs use more and more external plug-ins, types could be put to good use
- Program become more modular, we would like to use type safe run-time composition

Dynamics in Clean

- Remember previous talk:
  - **dynamic** 20.5 + 21.5 **::** Real
    dynamic map (\x -> x + 1) **::** [Int] -> [Int]
  - isZero :: Dynamic -> Bool
    isZero **(**x **::** Int**)** = x == 0
    isZero d = False
  - **readDynamic** :: String *World -> (Dynamic,*World)
  - **writeDynamic** :: String Dynamic *World -> *World
  - dynApply :: Dynamic Dynamic -> Dynamic
    dynApply **(**f **::** a -> b**)** **(**x **::** a**)** = **dynamic** f x **::** b
    dynApply df dx = **raise** “cannot apply something”

Functional operating system

- Stores files as dynamic, executables have a function type (a -> b):
  - writeDynamic “document” (dynamic doc)
    where doc :: PostScript
  - writeDynamic “format” (dynamic fun)
    where fun :: *World -> *World
- Simple program loader:
  - (d, world1) = readDynamic “format” world
    case d of
            (f :: *World -> *World) -> f world1
- Type safe communication with dynamics

Running applications

- Shell X: program input
  - Run binary executable named program, with command line argument the string “input”
- Esther: program “input”
  - Esther uses same syntax as Clean compiler
  - Searches for the dynamic named program
  - Constructs the dynamic for the argument
  - Type checks/combines code to construct a dynamic semantically equal to the expression
  - Evaluates and shows resulting dynamic

Running applications

- Type checks/combines code:
  - df = **readDynamic** "program"
    dx = **dynamic** "input" **::** String
    dy = dynApply df dx

    type of x

  - dynApply :: Dynamic Dynamic -> Dynamic
    dynApply **(**f **::** a -> b**) (**x **::** a**)** = **dynamic** f x **::** b
    dynApply df dx = **raise** "cannot apply"

    must unify with type of x

    result type of f
- Can be used for any type, examples:
  - sort [10, 5, 7, 3, 2, 8, 9, 1, 6]
  - sqrt 25.8
  - 40 + "bla" ← type error

Running applications

- Type checks/combines code:
  - df = **readDynamic** “program”
    dx = **dynamic** “input” **::** String
    dy = dynApply df dx

    code for application

  - dynApply :: Dynamic Dynamic -> Dynamic
    dynApply **(**f **::** a -> b**)** **(**x **::** a**)** = **dynamic** f x **::** b
    dynApply df dx = **raise** “cannot apply”

    code of f   code of x

- The dynamics run-time system keeps all application code in a data base
- The dynamics on disk give access to that code

Combining programs

- Shell X: pdf2ps document | print
  - Usually via pipe-lines, or temporary files
- Esther: print (pdf2ps document)
  or: (print o pdf2ps) document
  - Just function application or composition (o)
  - Suppose pdf2ps has type: PDF -> PostScript
  - Esther does a type check to make sure the document has type PDF, and that print expects values of type PostScript

Defining new programs

- Shell X: real programs? not possible!
  - Usually done by source code interpretation
- Esther: fac n = if (n <= 1) 1 (n * fac (n – 1))
  - This creates a dynamic of disk named fac
  - Esther uses compiled code of if, <=, *, and -.
  - Again, Esther will type check applications:
    fac 3 → 6
    fac [3] → cannot apply fac :: Int -> Int
    to [3] :: [Int]
- Such functions can be used by other (pre-compiled) programs: Demo

Defining new programs

- Programs can be defined as if they were functions:
  - Using lambda abstraction:
    (o) **infix** f g **=** ****x **->** f (g x)
  - Or using pattern matching:
    if b x y **=** **case** b **of** True **->** x**;** False **->** y
  - Or using higher-order functions:
    sort list **=** sortBy (<) list
    incList list **=** map ((+) 1) list
- First class programs: no distinction between programs and functions

Overloading

- Observation: functions can only be used on the correct types
- Problem: you want to use the same function (name) for different types
  - +, for example, can be used for Int and Real
- Sollution in Clean: overloading
  - ```
    class + a :: a a -> a
    instance + Int
    instance + Real
    ```
  - `1 + 1` == 2 :: Int
  - `1.0 + 1.5` == 3.5 :: Real
    ```

Overloading in Esther

- Requires additional infrastructure:
  - + is stored as a 'overloaded dynamic', with a special type: + :: a a -> a | + a
  - Esther looks for instances of +, when she knows which type is used: instance + Int
- Users can define their own instances for overloaded functions:
  - addLists x y **=** x ++ y
  - addLists **>>** instance + [a]
  - [1, 2] + [3, 4, 5] → [1, 2, 3, 4, 5]

File system

- Stores typed files: dynamics
- Stores globals: search path
- Stores fixity as file attributes
- Stores instances of overloaded functions by encoding it in the name:
  - instance + Int
  - instance + Real
  - instance one Real
- Effectively a catalog of typed code and data

Lazy evaluation

- Esther is lazy, demand driven evaluation
  - For example, an infinite list of 1's:
    let list **=** [1:list] **in** list **>>** ones
  - This saves an infinite (cyclic) list as a file
  - The list will only be evaluated as far as needed by other computations:
    take 5 ones → [1,1,1,1,1]
  - Just as powerful as the Clean compiler

Implementation

- All features can be done by dynamics
  - Application can be done by dynApply
- Some require syntax transformations:
  - Lambda ⇨ combinators (*I*, *K*, *S*, ...) & apply
  - Let ⇨ lambda & *Y* combinator
  - Case ⇨ if-then-else cascade & low level code
  - Functions ⇨ let & lambda
  - Sugar ⇨ functions
- Values come from the file system
- Denotations come from the parser

Lambda

- We don't know how to convert lambda in Esther to lambda in Clean.
- Instead we reuse dynamic application:
  - Use bracket abstraction to transform lambda abstraction into application and 3 combinators
  - \x -> e ⇨ [e]x
  - ```
    [\y -> e]x    = [ [e]y ]x
    [e1 e2]x      = S  [e1]x  [e2]x
    [x]x          = I
    [e]x          = K  e
    ```
  - I x = x, K x y = x, S f g x = f x (g x)
    ```

Let

- Let constructs are used to label subexpressions to create cycles or sharing
- We reuse our lambda conversion:
  - let x = e1 in e2 ⇨ (\x -> e2) (Y (\x -> e1))
  - Y f = f (Y f)
  - This is a standard trick to transform let expressions
  - There are also standard way to convert mutual recursive let expressions into a single let expression

Functions

- Functions are lambda abstraction with a name
- We reuse our lambda and let conversion:
  - f x1 ... xn = e ⇨ let f = \x1 ... xn -> e in f
  - Just another lambda function and a let, which we already know how to handle

Conclusions

- Basic, but complete, functional shell, that works on a typed file system
- Type checked/inferred command line
- Reuses compiled code
- Programs can be used as functions/ functions can be used as programs
- Command line expressions can be used in other pre-compiled programs
- Copy-paste compatible with the Clean compiler