PL, Jr. Summary: Findler, Felleisen - Contracts for Higher-Order Functions

Stephen Chang

3/8/2010

1. OO languages recognize value of assertion-based contracts (to specify pre/post-conditions) (Eiffel - "Design by Contracts", one of most requested Java extensions) (Bigloo Scheme is only functional language with contracts)

2. functional languages use type systems to express assertions but many type systems are not expressive enough to express some assertions

3. authors present λ calculus + contracts language and prove type soundness

4. implementation: contract compiler inserts code to check conditions required by contract

5. blame/contract verification: easy with first order contracts – caller blamed for input contract violation and function blamed for output contract violation

6. not as easy to verify contracts in language with higher order functions, example:

$$\begin{array}{l} g : (\mathbf{int}[>9] \to \mathbf{int}[0,99]) \to \mathbf{int}[0,99] \\ \mathbf{val\ rec}\ g = \lambda\, proc. \cdots \end{array}$$

   It's not enough to monitor applications of *proc* in *g* because *g* may pass *proc* to another function

7. blame is also complicated with higher order functions

8. contract syntax example:

$$\begin{array}{l} ;;\ \textit{bigger-than-zero?} : \mathbf{number} \to \mathbf{boolean} \\ (\mathbf{define}\ \textit{bigger-than-zero?}(\lambda(x)(\geq x\ 0))) \\ \\ \textit{sqrt} : \mathbf{number} \to \mathbf{number} \\ (\mathbf{define/contract}\ \textit{sqrt} \\ \quad (\textit{bigger-than-zero?} \longmapsto \textit{bigger-than-zero?}) \\ \quad (\lambda(x) \cdots)) \end{array}$$

   Things to note about example:

   - contracts can be predicates or function contracts (one predicate for domain and one for range)
   - contract can be arbitrary expression that evaluates to a contract – use of *bigger-than-zero?*

9. example of dependent contract – range predicate can depend on function argument

$$
\begin{array}{l}
\mathit{sqrt} : \textbf{number} \rightarrow \textbf{number} \\
(\textbf{define/contract}\ \mathit{sqrt} \\
\quad (\mathit{bigger\text{-}than\text{-}zero?} \overset{d}{\longmapsto} \\
\qquad (\lambda(x).\lambda(res). \\
\qquad\quad (\textbf{and}\ (\mathit{bigger\text{-}than\text{-}zero?}\ res) \\
\qquad\qquad (\geq (\mathit{abs}\ (-\ x\ (*\ res\ res)))\ 0.01)))) \\
\quad (\lambda(x) \cdots))
\end{array}
$$

10. key to checking higher-order contracts is to postpone enforcement until higher order function is applied – delay/save/saved/use example

11. contracts also allow for better code modularity because checking for validity of inputs is separated

12. assigning blame when first-class functions are involved:

    - if base contract appears to the left of an even number of arrows, then function where first-class function is applied is to blame (covariant)
    - if base contract appears to the left of an odd number of arrows, then calling function is to blame (contravariant)

    example:

$$
\begin{array}{l}
;;\, g : (\textbf{int} \rightarrow \textbf{int}) \rightarrow \textbf{int} \\
(\textbf{define/contract}\ g \\
\quad ((\mathit{greater\text{-}than\text{-}nine?} \longmapsto \mathit{between\text{-}zero\text{-}and\text{-}ninety\text{-}nine?}) \\
\quad \longmapsto \\
\quad \mathit{between\text{-}zero\text{-}and\text{-}ninety\text{-}nine?}) \\
\quad (\lambda(f)(f\ 0))
\end{array}
$$

    - *greater-than-nine?* contract is violated and since it appears to the left of two arrows (even), then g is to blame and this is true
    - If instead f was applied to 10 and $(f\ 10) \Rightarrow -10$, then second *between-zero-and-ninety-nine?* contract is violated and since it appears to the left of one arrow (odd), then whoever called g is to blame and this is also true

13. in Scheme, contracts are first-class

14. contracts are useful when dealing with callbacks because the save/use model is frequently encountered (callbacks are first "registered" and then used later)