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

Stephen Chang

3/8/2010

Intro

This talk is about contracts in languages with higher order functions. I'll basically be following the paper by Findler and Felleisen - Contracts For Higher-Order Functions from ICFP 2002.

History

1. Contracts have been around long before this paper and assertion-based contracts are commonly used in OO programming languages (preconditions/postconditions)
2. Eiffel coined the phrase "Design by Contracts" and the philosophy behind it (1988-1992)
3. Contracts is one of the most requested Java extensions

1. languages with higher order functions generally dont support contracts because enforcing invariants on function arguments is undecidable
2. such languages instead have focused on type systems that can approximate some invariants

Paper gives 2 motivations for adding contracts to higher order languages:

1. contracts are more expressive than type systems so you can express more invariants
2. By showing what invariants cannot be expressed by current type systems, contracts can inspire future type system research

Contract Syntax

First Order contract example - used to illustrate syntax:

$$
\begin{array}{l}
;;\ \textit{bigger-than-zero?} : \textbf{number} \to \textbf{boolean} \\
(\textbf{define}\ \textit{bigger-than-zero?}(\lambda(x)(\geq x\ 0)))
\end{array}
$$

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

Things to note from example:

1. contracts can be predicates or function contracts (one predicate for domain and one for range)
2. contract can be arbitrary expression that evaluates to a contract – use of *bigger-than-zero?*
3. contracts for function (range) can depend on function input
4. Blame is easy, function caller blamed if input predicate violated, function itself blamed if output predicate is violated

Contracts for Higher Order Fns

Basic example:

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

Things to note:

1. contract can't be checked until f is applied
2. easier said than done because g can pass f as argument to another function

– Give delay/save/saved/use example.

However, blame for higher order functions gets complicated. How do you determine who to blame when you can't enforce contract until application and functions can get passed to other functions?

Contract compiler compiles expressions with contracts into "obligation" expression $\boxed{e^{c,x,y}}$ where e is expression with a contract, often a function, c is contract, x is party responsible for values coming out of e, and y is party responsible for values coming into e

Obligation reductions:

$$D[V_1^{\textbf{contract}(V_2),out,in}] \overset{flat}{\to} D[if\ V_2(V_1)\ then\ V_1\ else\ blame(out)]$$

$$D[(V_1^{(V_3 \longmapsto V_4),out,in}\ V_2)] \overset{hoc}{\to} D[(V_1\ V_2^{V_3,in,out})^{V_4,in,out}]$$

Assigning blame when first-class functions are involved:

1. 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)

2. if base contract appears to the left of an odd number of arrows, then calling function is to blame (contravariant)

Example:

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

$$g\ (\lambda x.25)$$

$$
\begin{array}{l}
\quad g^{((gt9 \longmapsto bet0-99) \longmapsto bet0-99),g,main}\ (\lambda x.25) \\
\to (g\ (\lambda x.25)^{(gt9 \longmapsto bet0-99),main,g})^{bet0-99,g,main} \\
\to ((\lambda x.25)^{(gt9 \longmapsto bet0-99),main,g}\ 0)^{bet0-99,g,main} \\
\to (((\lambda x.25)\ 0^{gt9,g,main})^{bet0-99,main,g})^{bet0-99,g,main} \\
\to (((\lambda x.25)\ (if\ gt9(0)\ then\ 0\ else\ blame(g))^{bet0-99,main,g})^{bet0-99,g,main} \\
\to blame(g)
\end{array}
$$

Other

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

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

5. blame is also complicated with higher order functions

6. contract syntax example:

$$
\begin{array}{l}
;; \mathit{bigger\text{-}than\text{-}zero?} : \textbf{number} \rightarrow \textbf{boolean} \\
(\textbf{define}\ \mathit{bigger\text{-}than\text{-}zero?} (\lambda(x)(\geq x\ 0)))
\end{array}
$$

$$
\begin{array}{l}
\mathit{sqrt} : \textbf{number} \rightarrow \textbf{number} \\
(\textbf{define/contract}\ \mathit{sqrt} \\
\quad (\mathit{bigger\text{-}than\text{-}zero?} \longmapsto \mathit{bigger\text{-}than\text{-}zero?}) \\
\quad (\lambda(x) \cdots))
\end{array}
$$

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

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

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

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

10. in Scheme, contracts are first-class

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