Union and Intersect Contracts

Tim Disney and Cormac Flanagan

1 Introduction

The following is a sketch of union and intersection contracts. The key idea is to hold onto a reference to the original union or intersection contract and potentially defer blaming until one or both sides have failed their contract.

2 Language

We extend the standard lambda calculus with contract checking (see Figure 1). We write $e \ @^b \ C$ to denote applying a contract C to a term e, where b is the blame label.

Contracts C include the standard predicate and function contracts along with union $C \cup C$ and intersection $C \cap C$.

We use b to range over blame labels, which are either positive blame labels k, or negative blame labels $!k$. The operator $!$ is an involution, so $!!k = k$.

3 Semantics

Dealing with union and intersection contracts requires a bit of extra machinery in the semantics. In particular, to check a union contract

$$v \ @^b \ (C \cup D)$$

we reduce it to a term that checks *both* contracts:

$$(v \ @^k \ C) \ @^h \ D$$

where k and h are fresh blame labels.

If only one of these two contracts, say $@^k \ C$ fails, that failure is not reported and instead the program keeps executing, as v may still satisfy D,

Figure 1: Language

e	::=	$x \mid c \mid \lambda x.\, e \mid e\ e \mid e \oplus e \mid e\ @^b\ C$	**Expressions**
$\oplus$	::=	$+ \mid - \mid < \mid > \mid \ldots$	**Operators**
E	::=	$[\,] \mid E\ e \mid v\ E \mid E \oplus e \mid c \oplus E \mid$ $E\ @^b\ C \mid v\ @^b\ E$	**Evalulation Contexts**
v	::=	$c \mid \lambda x.\, e$	**Values**
c	::=	$\mathit{boolean} \mid \mathit{number} \mid \ldots$	**Constants**
C, D	::=	$\texttt{flat}\ e \mid C \to C \mid C \cup C \mid C \cap C$	**Contracts**
$k, h \in$		$\mathit{PosBlameLabel}$	**Blame Labels**
b	::=	$k \mid !k$	

and hence also satisfy $C \cup D$. It is only when the $@^k\ C$ and $@^h\ D$ contracts *both* fail that the original $@^b\ (C \cup D)$ contracts fails.

To implement this semantics, we need an additional state component F, where $F(k)$ records what *failure action* to take when a contract labeled with k or $!k$ fails. In the example above, we set

$$\begin{array}{rcl} F(k) & = & \texttt{union}(h, b) \\ F(h) & = & \texttt{union}(k, b) \end{array}$$

to record that each of the contracts labeled h and k is unioned with the other; and that if both fail then blame should propagate to b.

If the k contract is later violated by its subject, we record that info by updating $F(k)$ to `failed-subj`, but no further action is taken as h has not failed. However, if the h contract is later violated by its subject, since $F(h) = \texttt{union}(k, b)$ and $F(k)$ is already `failed-subj`, we propagate blame to b.

For any blame label k in a source program, we keep $F(k)$ undefined; failures of the corresponding contract are immediately reported to the user.

The operational semantics in Figure 2 is based on these ideas. Each evaluation state includes an expression and a failure map F (initially empty).

The rules for flat and function contracts are standard, except that contract failures invoke the auxiliary function `fail` (Figure 3), which updates the failure map and may report contract violations.

$$\texttt{fail} : \texttt{FailureMap} \times \texttt{BlameLabel} \to \texttt{FailureMap}$$

We first describe the behavior of $\texttt{fail}(F, k)$, when called with a positive blame label reflecting a subject violation of k.

- For $\texttt{fail}(F, k)$, if $F(k)$ has already failed, then it is left unchanged. In particular note that if k was first violated by its context and later violated by its subject, the first contract violation takes precedence (in part, because it may have been the cause of the later subject violation).
- If $F(k) = \texttt{intersect}(h, b)$ arising from a reduction

  $$F_1, E[v \ @^b \ (C \cap D)] \to F_2, E[(v \ @^k \ C) \ @^h \ D]$$

  then we record that k has failed, and proceed to blame b, since the intersection has now also failed.
- Next supposed $F(k) = \texttt{union}(h, b)$ arising from a reduction

  $$F_1, E[v \ @^b \ (C \cup D)] \to F_2, E[(v \ @^k \ C) \ @^h \ D]$$

  We first record the failure on k by extending the failure map with $F[k := \texttt{failed-subj}]$. Next, if h has already been subject-violated, then we proceed to blame b.
- If $F(k)$ is undefined, then it originated from a contract in the source program, and that error is reported.

For negative blame labels reflecting a context violation of k, $\texttt{fail}(F, !k)$ behaves as the dual of the above, swapping subject and context violations, and union and intersection.

Figure 2: Semantics

$$
\begin{array}{rcll}
T & ::= & \texttt{failed-subj} \mid \texttt{failed-ctxt} \mid & \textbf{Failure Action} \\
 & & \texttt{union}(k, b) \mid \texttt{intersect}(k, b) & \\
F & ::= & \mathit{PosBlameLabel} \to T & \textbf{Failure Map}
\end{array}
$$

$$
\begin{array}{lcl}
F, E[(\lambda x.\, e)\ v] & \to & F, E[e[x := v]] \\
F, E[c_1 \oplus c_2] & \to & F, E[\delta(\oplus, c_1, c_2)] \\
F, E[v\ @^b\ \texttt{flat}\ e] & \to & F, E[v\ @^b\ e\ v] \\
F, E[v\ @^b\ \mathit{true}] & \to & F, E[v] \\
F, E[v\ @^b\ \mathit{false}] & \to & \texttt{fail}(F, b), E[v] \\
F, E[(\lambda x.\, e)\ @^b\ (C \to D)] & \to & F, E[\lambda y.\, ((\lambda x.\, e)\ (y\ @^{!b}\ C))\ @^b\ D] \\
F, E[v\ @^b\ (C \cup D)] & \to & F', E[(v\ @^k\ C)\ @^h\ D] \\
 & & \text{where } F' = F[k := \texttt{union}(h, b), \\
 & & \qquad\qquad h := \texttt{union}(k, b)], \\
 & & \quad h, k \text{ are fresh} \\
F, E[v\ @^b\ (C \cap D)] & \to & F', E[(v\ @^k\ C)\ @^h\ D] \\
 & & \text{where } F' = F[k := \texttt{intersect}(h, b), \\
 & & \qquad\qquad h := \texttt{intersect}(k, b)], \\
 & & \quad h, k \text{ are fresh}
\end{array}
$$

Figure 3: Fail

$$\texttt{fail} : \texttt{FailureMap} \times \texttt{BlameLabel} \to \texttt{FailureMap}$$

$$\begin{array}{ll}
\texttt{fail}(F,k) = F & \texttt{fail}(F,!k) = F \\
\quad \text{where } F(k) = \texttt{failed-subj} & \quad \text{where } F(k) = \texttt{failed-subj} \\
\\
\texttt{fail}(F,k) = F & \texttt{fail}(F,!k) = F \\
\quad \text{where } F(k) = \texttt{failed-ctxt} & \quad \text{where } F(k) = \texttt{failed-ctxt} \\
\\
\texttt{fail}(F,k) = \texttt{fail}(F',b) & \texttt{fail}(F,!k) = \texttt{fail}(F',!b) \\
\quad \text{where } F(k) = \texttt{intersect(h, b)}, & \quad \text{where } F(k) = \texttt{union(h, b)}, \\
\qquad F' = F[k := \texttt{failed-subj}] & \qquad F' = F[k := \texttt{failed-ctxt}] \\
\\
\texttt{fail}(F,k) = \texttt{fail}(F',b) & \texttt{fail}(F,!k) = \texttt{fail}(F',!b) \\
\quad \text{where } F(k) = \texttt{union(h, b)}, & \quad \text{where } F(k) = \texttt{intersect(h, b)}, \\
\qquad F(h) = \texttt{failed-subj} & \qquad F(h) = \texttt{failed-ctxt} \\
\qquad F' = F[k := \texttt{failed-subj}] & \qquad F' = F[k := \texttt{failed-ctxt}] \\
\\
\texttt{fail}(F,k) = F' & \texttt{fail}(F,!k) = F' \\
\quad \text{where } F(k) = \texttt{union(h, b)}, & \quad \text{where } F(k) = \texttt{intersect(h, b)}, \\
\qquad F(h) \neq \texttt{failed-subj} & \qquad F(h) \neq \texttt{failed-ctxt} \\
\qquad F' = F[k := \texttt{failed-subj}] & \qquad F' = F[k := \texttt{failed-ctxt}] \\
\\
\texttt{fail}(F,k) = \texttt{blame}(k, \texttt{subject}) & \texttt{fail}(F,!k) = \texttt{blame}(k, \texttt{context}) \\
\quad \text{where } F(k) \text{ is undefined} & \quad \text{where } F(k) \text{ is undefined}
\end{array}$$