Hybrid contract checking via symbolic simplification

Dana N. Xu

INRIA Paris-Rocquencourt

From Types to Contracts

```
(* val inc : int -> int *)
contract inc = {x | x > 0} -> {y | y > x}
let inc x = x + 1
```

From Types to Contracts

```
(* val inc : int -> int *)
contract inc = {x | x > 0} -> {y | y > x}
let inc x = x + 1

let h1 = inc true       (* type error *)
```

From Types to Contracts

```
(* val inc : int -> int *)
contract inc = {x | x > 0} -> {y | y > x}
let inc x = x + 1

let h1 = inc true        (* type error *)

let h2 = inc 0           (* contract error *)
```

Example - append

```
contract len = {xs | true} -> {n | n >= 0}
let rec len (xs: int list) = match xs with
  | [] -> 0
  | (h::t) -> 1 + len t

contract append = {xs | true} -> {ys | true}
               -> {rs | len xs + len ys = len rs}
let rec append (xs: int list) (ys: int list) =
 match xs with
 | [] -> ys
 | x::l -> x :: append l ys
```

Example - filter

```
let rec for_all (p : int -> bool) (xs : int list) =
 match xs with
  | [] -> true
  | a::l -> p a && for_all p l

contract filter = {p | true} -> {xs | true}
                -> {zs | for_all p xs}
let rec filter (p : int -> bool)  (xs : int list) =
 match xs with
 | [] -> []
 | (a::l) -> let res = filter p l in
             if p a then a::res
                    else res
```

Example - map, rev_map

```
contract map = {f | true} -> {xs | true}
            -> {ys | len xs = len ys}
let rec map f (xs: int list) = match xs with
              | [] -> []
              | (h::t) -> f h :: map f t

contract rmap_f = {f | true} -> {xs | true}
 -> {ys | true} -> {zs | len zs = len xs + len ys}
let rec rmap_f f (accu : int list) (ys: int list) =
  match ys with
  | [] -> accu
  | a::l -> rmap_f f (f a :: accu) l

contract rev_map = {f | true} -> {xs | true}
                -> {ys | len xs = len ys}
let rev_map f l = rmap_f f [] l
```

Example - flatten

```
let rec sum_len (xs: (int list) list) =
 match xs with
 | [] -> 0
 | (a::l) -> len a + sum_len l

contract flatten = {xs | true}
                -> {ys | len ys = sum_len xs}
let rec flatten (xs : (int list) list) =
  match xs with
  | [] -> []
  | l::r -> append l (flatten r)
```

Example - rev

```
contract rev_append = {xs | true} -> {ys | true}
                  -> {zs | len xs + len ys = len zs}

let rec rev_append (l1 : int list) (l2 : int list) =
  match l1 with
  | [] -> l2
  | a :: l -> rev_append l (a :: l2)

contract rev = {xs | true} -> {ys | len xs = len ys}
let rev l = rev_append l []
```

Example - McCarthy's 91 function

```
contract mc91 = {n | true}
            -> {z | if n <=101 then z = 91
                                else z = n- 10}
let rec mc91 x =
  if x > 100 then x - 10
  else mc91 (mc91 (x + 11))
```

Error message reporting

```
(* val f1 : (int -> int) -> int *)
contract f1 = ({x | x >= 0} -> {y | y >= 0})
            -> {z | z >= 0}
let f1 g = (g 1) - 1
let f2 = f1 (fun x -> x - 1)
```

f1 does not satisfy its postcondition

The Language

$$
\begin{array}{lcll}
a, e, p & \in & \textbf{Exp} & \textbf{Expressions} \\
a, e, p & ::= & n & \text{integers} \\
 & \mid & r & \text{blame} \\
 & \mid & x \mid \lambda(x^{\tau}).e \mid e_1\ e_2 & \\
 & \mid & \text{match } e_0 \text{ with } \overrightarrow{alt} & \text{pattern-matching} \\
 & \mid & K\ \overrightarrow{e} & \text{constructor} \\
 & & & \\
alt & ::= & K\ (x_1^{\tau_1}, \ldots, x_n^{\tau_n})\ \rightarrow\ e & \textbf{Alternatives} \\
r & ::= & \text{BAD}^{l} \mid \text{UNR}^{l} & \textbf{Blames} \\
l & ::= & (n_1, n_2, \text{String}) & \textbf{Label} \\
 & & & \\
val & ::= & n \mid x \mid r \mid K\ \overrightarrow{val} \mid \lambda(x^{\tau}).e & \textbf{Values} \\
tv & ::= & n \mid x \mid K\ \overrightarrow{tv} & \\
tval & ::= & tv \mid \lambda(x^{\tau}).e & \textbf{Trivial values}
\end{array}
$$

Contracts

$$
\begin{array}{rcll}
t & \in & \textbf{Contracts} & \\
t & ::= & \{x \mid p\} & \text{predicate contract} \\
 & \mid & x : t_1 \rightarrow t_2 & \text{dependent function contract} \\
 & \mid & (x : t_1,\ t_2) & \text{dependent tuple contract} \\
 & \mid & \text{Any} & \text{polymorphic Anycontract}
\end{array}
$$

E.g.,

$$k : (\{x \mid x > 0\} \rightarrow \{y \mid y > x\}) \rightarrow \{z \mid k\ 5 > z\}$$

E.g.,

$$(\{x \mid x > 0\}, \{y \mid y > x\})$$

Contract Satisfaction

For a well-typed expression e, define $e \in t$ thus:

$$e \in \{x \mid p\} \iff e\uparrow \text{ or } (e \text{ is crash-free and } p[e/x] \to^* \text{true}) \qquad [A1]$$

$$e \in x{:}t_1 \to t_2 \iff e\uparrow \text{ or } (e \to^* \lambda x.e_2 \text{ and } \forall val \in t_1.\ (e\ val) \in t_2[val/x]) \qquad [A2]$$

$$e \in (x{:}t_1, t_2) \iff e\uparrow \text{ or } (e \to^* (val_1, val_2) \text{ and } val_1 \in t_1 \text{ and } val_2 \in t_2[val_1/x]) \qquad [A3]$$

$$e \in \text{Any} \iff \text{true} \qquad [A4]$$

Contract Wrappers

$$e \triangleright t = e \overset{\text{BAD}^{l_1}}{\underset{\text{UNR}^{l_2}}{\bowtie}} t \qquad e \triangleleft t = e \overset{\text{UNR}^{l_2}}{\underset{\text{BAD}^{l_1}}{\bowtie}} t$$

$$e \overset{r_1}{\underset{r_2}{\bowtie}} \{x \mid p\} \;=\; \text{let } x = e \text{ in if } p \text{ then } x \text{ else } r_1 \qquad \text{[P1]}$$

$$e \overset{r_1}{\underset{r_2}{\bowtie}} x : t_1 \to t_2 = \text{let } y = e \text{ in} \qquad \text{[P2]}$$

$$\lambda x_1.((y\ (x_1 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(x_1 \overset{\underline{r_2}}{\underset{\underline{r_1}}{\bowtie}} t_1)/x])$$

$$e \overset{r_1}{\underset{r_2}{\bowtie}} (x : t_1, t_2) = \text{match } e \text{ with} \qquad \text{[P3]}$$

$$(x_1, x_2) \to (x_1 \overset{r_1}{\underset{r_2}{\bowtie}} t_1, x_2 \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(x_1 \overset{\underline{r_2}}{\underset{\underline{r_1}}{\bowtie}} t_1)/x])$$

$$e \overset{r_1}{\underset{r_2}{\bowtie}} \text{Any} = r_2 \qquad \text{[P4]}$$

Main Theorem in Theory

Definition (Total contract)

A contract t is total iff

$$\begin{array}{ll} & t \textit{ is } \{x \mid p\} \textit{ and } \lambda x.p \textit{ is total (i.e. crash-free, terminating)} \\ \textit{or} & t \textit{ is } x : t_1 \to t_2 \textit{ and } t_1 \textit{ is total and} \\ & \qquad\qquad \textit{for all } val_1 \in t_1, t_2[val_1/x] \textit{ is total} \\ \textit{or} & t \textit{ is } (x : t_1, t_2) \textit{ and } t_1 \textit{ is total and} \\ & \qquad\qquad \textit{for all } val_1 \in t_1, t_2[val_1/x] \textit{ is total} \\ \textit{or} & t \textit{ is } Any \end{array}$$

Theorem (Soundness and completeness of contract checking)

For all closed expression e^τ, closed and total contract t^τ,

$$(e \triangleright t) \textit{ is crash-free} \quad \Longleftrightarrow \quad e \in t$$

Crash-free Expressions

Definition (Crash-free Expression)

A (possibly-open) expression e is crash-free iff :

$$\forall \mathcal{C}.\ BAD \notin_s \mathcal{C} \ and \ (\mathcal{C}[\![e]\!])^{bool} \ \Rightarrow \ \mathcal{C}[\![e]\!] \not\to^* BAD$$

$(2, \mathrm{BAD})$	No
$(2, 3)$	Yes
$\lambda x.\mathrm{if}\ x * x \geq 0\ \mathrm{then}\ x\ \mathrm{else\ BAD}$	Yes

Main Theorem in Practice

Theorem (Soundness of contract checking)

For all closed expression e^{τ}, *closed and terminating contract* t^{τ},

$$(e \triangleright t) \text{ is crash-free} \quad \Rightarrow \quad e \in t$$

SL machine: symbolic *simplification* with a *logical* store

$\langle \mathcal{H} \parallel e \parallel \mathcal{S} \parallel \mathcal{L} \rangle$ means "simplify e"
$\langle\!\langle \mathcal{H} \parallel e \parallel \mathcal{S} \parallel \mathcal{L} \rangle\!\rangle$ means "rebuild e"

- $\mathcal{H}$ is an environment mapping variables to trivial values.
- e is the expression under simplification (or being rebuilt).
- $\mathcal{S}$ is a stack.

$$\begin{aligned} \mathcal{S} ::= & \; [\,] \mid (\bullet\ e) :: \mathcal{S} \mid (e\ \bullet) :: \mathcal{S} \mid (\lambda x.\bullet) :: \mathcal{S} \mid (\text{let } x = \bullet \text{ in } e) :: \mathcal{S} \\ & \mid (\text{match } \bullet \text{ with } alt) :: \mathcal{S} \mid (\text{let } x = e \text{ in } \bullet) :: \mathcal{S} \\ & \mid (\text{match } e_0 \text{ with } \overrightarrow{K\ \overrightarrow{x} \to (\bullet, \mathcal{S}, \mathcal{L})}) :: \mathcal{S} \end{aligned}$$

- $\mathcal{L}$ is a logical store which contains the ctx-info.

$$\mathcal{L} ::= \emptyset \mid \forall x : \tau, \mathcal{L} \mid \phi, \mathcal{L}$$

Example

$$\langle \emptyset \parallel \begin{array}{ll}\lambda x. & \text{if } x > 0 \text{ then (if } x+1 > 0 \\ & \text{then 5 else BAD) else UNR}\end{array} \parallel [\,] \parallel \emptyset \rangle$$

$$\rightsquigarrow \langle \emptyset \parallel \begin{array}{l}\text{if } x > 0 \text{ then (if } x+1 > 0 \\ \text{then 5 else BAD) else UNR}\end{array} \parallel (\lambda x.\bullet) :: [\,] \parallel \forall x : \text{int} \rangle$$

$$\rightsquigarrow \langle\langle \emptyset \parallel x > 0 \parallel \begin{array}{ll}\text{(if } \bullet \text{ then} & \text{(if } x+1 > 0 \\ & \text{then 5 else BAD)} \\ \multicolumn{2}{l}{\text{else UNR)} :: (\lambda x.\bullet) :: [\,]}\end{array} \parallel \forall x : \text{int} \rangle\rangle$$

$$\rightsquigarrow [\langle \emptyset \parallel \begin{array}{l}\text{if } x+1 > 0 \\ \text{then 5 else BAD}\end{array} \parallel \begin{array}{l}\text{(if } x > 0 \text{ then } \bullet) \\ :: (\lambda x.\bullet) :: [\,]\end{array} \parallel \begin{array}{l}\forall x : \text{int}, \\ x > 0\end{array} \rangle;$$

$$\langle \emptyset \parallel \text{UNR} \parallel (\text{if } x > 0 \text{ else } \bullet) :: \mathcal{S} \parallel \forall x : \text{int}, \mathit{not}(x > 0) \rangle]$$

Example (cont.)

$$\rightsquigarrow \quad [\langle\!\langle \emptyset \parallel 5 \parallel \begin{array}{l}(\text{if } x > 0 \text{ then } \bullet)\\ :: (\lambda x.\bullet) :: [\,]\end{array} \parallel \begin{array}{l}\forall x : \text{int}, x > 0,\\ (x+1>0)\end{array} \rangle\!\rangle;$$

$$\langle\!\langle \emptyset \parallel \text{UNR} \parallel \begin{array}{l}(\text{if } x > 0 \text{ else } \bullet)\\ :: (\lambda x.\bullet) :: [\,]\end{array} \parallel \begin{array}{l}\forall x : \text{int},\\ \textit{not}(x > 0)\end{array} \rangle\!\rangle]$$

$$\rightsquigarrow \quad \langle\!\langle \emptyset \parallel \text{if } x > 0 \text{ then } 5 \text{ else UNR} \parallel (\lambda x.\bullet) :: [\,] \parallel \forall x : \text{int} \rangle\!\rangle$$

$$\rightsquigarrow \quad \langle\!\langle \emptyset \parallel \lambda x.\ \text{if } x > 0 \text{ then } 5 \text{ else UNR} \parallel [\,] \parallel \forall x : \text{int} \rangle\!\rangle$$

$$\rightsquigarrow \quad \lambda x.\ \text{if } x > 0 \text{ then } 5 \text{ else UNR}$$

$$\langle \mathcal{H} \parallel \lambda v.\ \text{let } y = v + 1 \text{ in if } y > v \text{ then } y \text{ else BAD} \parallel [\,] \parallel \emptyset \rangle$$

$$\rightsquigarrow \quad \langle \mathcal{H} \parallel v + 1 \parallel \begin{array}{l} (\text{let } y = \bullet \text{ in if } y > v \\ \text{then } y \text{ else BAD}) :: (\lambda v.\bullet) :: [\,] \end{array} \parallel \forall v : \text{int} \rangle$$

$$\rightsquigarrow^* \quad \langle\!\langle \mathcal{H} \parallel v + 1 \parallel \begin{array}{l} (\text{let } y = \bullet \text{ in if } y > v \\ \text{then } y \text{ else BAD}) :: (\lambda v.\bullet) :: [\,] \end{array} \parallel \forall v : \text{int} \rangle\!\rangle$$

$$\rightsquigarrow \quad \langle \mathcal{H} \parallel \begin{array}{l} \text{if } y > v \\ \text{then } y \\ \text{else BAD} \end{array} \parallel \begin{array}{l} (\text{let } y = v + 1 \text{ in } \bullet) \\ :: (\lambda x.\bullet) :: [\,] \end{array} \parallel \begin{array}{l} \forall v : \text{int}, \\ \forall y : \text{int}, \\ y = v + 1 \end{array} \rangle$$

$$\rightsquigarrow \quad \forall v : \text{int}, \forall y : \text{int}, y = v + 1 \Rightarrow y > v$$

$$\rightsquigarrow \quad \langle \mathcal{H} \parallel y \parallel \begin{array}{l} (\text{let } y = v + 1 \text{ in } \bullet) \\ :: (\lambda v.\bullet) :: [\,] \end{array} \parallel \begin{array}{l} \forall v : \text{int}, \forall y : \text{int}, \\ y = v + 1, y > v \end{array} \rangle$$

$$\rightsquigarrow \quad \lambda v.\ \text{let } y = v + 1 \text{ in } y$$

Correctness

Definition (Semantically Equivalent)

Two expressions e_1 and e_2 are semantically equivalent, namely $e_1 \equiv_s e_2$, iff $\forall \mathcal{C}$, $(\mathcal{C}[\![e_i]\!])^{bool}$ for $i = 1, 2$, $r \in \{BAD, UNR\}$,

$$\mathcal{C}[\![e_1]\!] \to^* r \iff \mathcal{C}[\![e_2]\!] \to^* r$$

Theorem (Correctness of SL machine)

For all expression e, if $\langle \emptyset \mid e \parallel [\,] \parallel \emptyset \rangle \rightsquigarrow^* a$, then $e \equiv_s a$.

Theorem (Soundness of static contract checking)

For all closed expression e, and closed and terminating contract t,

$$\langle \emptyset \mid e \triangleright t \parallel [\,] \parallel \emptyset \rangle \rightsquigarrow^* e' \text{ and } BAD \notin_s e' \quad \Rightarrow \quad e \in t$$

SMT solver Alt-ergo

Data type in OCaml

```
type 'a list = Nil | Cons of 'a * ('a list)
```

Data type in Alt-ergo

```
type 'a list
logic nil  : 'a list
logic cons : 'a , 'a list -> 'a list
```

Converting function types:

```
type ('a, 'b) arrow
logic apply : ('a, 'b) arrow , 'a -> 'b
```

$$\begin{aligned} [\![\tau_1 \ldots \tau_n\ T]\!] &= [\![\tau_1]\!] \ldots [\![\tau_n]\!]\ T \\ [\![\tau_1 \to \tau_2]\!] &= ([\![\tau_1]\!], [\![\tau_2]\!])\ \text{arrow} \end{aligned}$$

Example - append

$$
\begin{array}{l}
\lambda v_1.\lambda v_2.\text{match } v_1 \text{ with} \\
\mid [\,] \rightarrow \text{if len } v_2 = \text{len } v_1 + \text{len } v_2 \text{ then } v_2 \text{ else BAD}^{l_1} \\
\mid x :: u \rightarrow \quad \text{if (len } (x :: \\
\qquad\qquad\qquad\qquad (\text{if len (append } u\ v_2) = \text{len } u + \text{len } v_2 \\
\qquad\qquad\qquad\qquad\ \text{then append } u\ v_2 \text{ else UNR})) \\
\qquad\qquad\qquad\quad = \text{len } v_1 + \text{len } v_2) \\
\qquad\qquad\quad \text{then } x :: \text{append } u\ v_2 \text{ else BAD}^{l_2}
\end{array}
$$

Example - append (cont.)

```
logic len: ('a list, int) arrow
logic append: ('a list,
               ('a list,'a list) arrow) arrow

axiom len_def_1 : forall s:'a list. s = nil ->
 apply(len,s) = 0
axiom len_def_2 : forall s:'a list. forall x:'a.
 forall l:'a list. s = cons(x,l) ->
 apply(len,s) = 1 + apply(len,l)
```

```
goal app_1 : forall v1,v2:'a list. v1 = nil ->
 apply(len,v2) = apply(len,v1) + apply(len,v2)

goal app_2 : forall v1,v2,l:'a list.forall x:'a.
 v1 = cons(x,l) ->
 apply(len,apply(apply(append,l),v2))
     = apply(len,l) + apply(len,v2) ->
 (exists y:'a list. y = apply(apply(append,l),v2)
   and apply(len,cons(x, y))
       = apply(len,v1) + apply(len,v2))
```

Logicization

$$\oplus \in [+, -, *, /] \qquad \odot \in [>, <, =]$$

$$
\begin{array}{rcl}
[\![.]\!]_f & : & \textbf{Expression} \rightarrow \textbf{Formula} \\
[\![\text{let (rec) } f = e]\!]_f & = & [\![e]\!]_f \quad \text{top-level defn} \\
[\![\text{BAD}^l]\!]_f & = & \begin{cases} \mathit{true} & \text{for axioms} \\ \mathit{false} & \text{for goals} \end{cases} \\
[\![\text{UNR}^l]\!]_f & = & \mathit{false} \\
[\![x]\!]_f & = & f = x \\
[\![n]\!]_f & = & f = n \\
[\![e_1^{\tau_1} \oplus e_2^{\tau_2}]\!]_f & = & \exists x_1 : [\![\tau_1]\!],\ \exists x_2 : [\![\tau_2]\!], \\
& & [\![e_1]\!]_{x_1} \wedge [\![e_2]\!]_{x_2} \wedge f = x_1 \oplus x_2 \\
[\![e_1^{\tau_1} \odot e_2^{\tau_2}]\!]_f & = & \exists x_1 : [\![\tau_1]\!],\ [\![e_1]\!]_{x_1} \wedge \exists x_2 : [\![\tau_2]\!],\ [\![e_2]\!]_{x_2} \wedge \\
& & ((x_1 \odot_t x_2 \wedge f = \text{true}) \vee \\
& & (\mathit{not}(x_1 \odot_t x_2) \wedge f = \text{false}))
\end{array}
$$

Logicization (cont.)

$$
\begin{array}{rcl}
[\![\lambda x^{\tau}.e]\!]_f & = & \forall x : [\![\tau]\!],\ [\![e]\!]_{(\text{apply}(f,x))} \\
[\![\text{let } x^{\tau} = e_1 \text{ in } e_2]\!]_f & = & \exists x : [\![\tau]\!],\ [\![e_1]\!]_x \ \wedge [\![e_2]\!]_f \\
[\![e_1^{\tau_1}\ e_2^{\tau_2}]\!]_f & = & \exists x_1 : [\![\tau_1]\!],\ [\![e_1]\!]_{x_1} \wedge \\
 & & \exists x_2 : [\![\tau_2]\!],\ [\![e_2]\!]_{x_2} \wedge \\
 & & f = \text{apply}(x_1, x_2) \\
[\![K\ e_1^{\tau_1} \ldots e_n^{\tau_n}]\!]_f & = & \exists x_1 : [\![\tau_1]\!],\ [\![e_1]\!]_{x_1} \wedge \cdots \wedge \\
 & & \exists x_n : [\![\tau_n]\!],\ [\![e_n]\!]_{x_n} \wedge \\
 & & f = K\ (y_1, \ldots, y_n) \\
[\![\begin{array}{l} \text{match } e_0^{\tau_0} \text{ with} \\ \overrightarrow{K\ \overrightarrow{x^{\tau}} \to e} \end{array}]\!]_f & = & \begin{array}{l} \exists x_0 : [\![\tau_0]\!],\ [\![e_0]\!]_{x_0} \ \wedge \\ \overrightarrow{(\bigwedge \overrightarrow{\forall x : [\![\tau]\!]}, (x_0 = K\ \overrightarrow{x}) \Rightarrow [\![e]\!]_f)} \end{array}
\end{array}
$$

Correctness

Theorem (Logicization for axioms)

Given closed definition $f = e^{\tau}$*, the logical fomula* $\exists f : \tau, [\![e]\!]_f$ *is valid.*

Theorem (Logicization for goals: validity preservation)

For all (possibly open) expression e^{τ}*, for all* $fv(e)$*,* $\exists f : \tau$*, if* $[\![e]\!]_f$ *is valid and* $e \rightarrow e'$ *for some* e'*, then* $[\![e']\!]_f$ *is valid.*

Preliminary Experiments

Table: Results of preliminary experiments

program	total LOC	Ann LOC	Time (sec)
intro123, neg, mc91	28	5	0.10
ack, fhnhn, zipunzip	25	4	0.16
arith, sum, max	26	4	0.20
OCaml stdlib/list.ml	81	16	0.72

Conclusion

- program verification
- debugging
- precise blaming
- fast