Combining Static and Dynamic Contract Checking for Curry

Michael Hanus

University of Kiel
Programming Languages and Compiler Construction

LOPSTR 2017

Developing Reliable Software Systems

Program verification . . . perfect but not practical:

- difficult proofs, no fully automatic tools
- exact proof obligations / specifications

Declarative programming . . . good but not perfect:

- run-time errors exist
- objective: avoid possible run-time errors

Pragmatic approach: combine static and dynamic checks

- strong typing $\leadsto$ static detection of some run-time errors
- more complex conditions $\leadsto$ dynamic assertions

Our approach:

Move boundaries by static verification of dynamic assertions

Combining Static and Dynamic Checking

Advantages

- fully automatic approach
- more efficient reliable software

```
fac :: Int  → Int
fac n = if n==0 then 1
                else n * fac (n-1)
```

- `fac "Hello"`: statically rejected
- `fac (2-5)`: still possible ⇝ infinite loop

Add contracts (pre/postconditions) [PADL'12]:

```
fac'pre  n   = n >= 0
fac'post n f = f > 0
```

- `fac (x-2-x)`: rejected at run time

Contract Verification

Dynamic contract checking

- requires additional execution time
- often turned off in production systems

Our approach

- try to verify contracts at compile time
- if successful: remove dynamic contract checks
- otherwise: leave dynamic checks

Advantages

- reliable program execution
- more efficient (if successful)
- practical (if fully automatic)

Proof of Concept

Programming language: Curry

- declarative (functional logic) language
- results applicable to functional as well as logic languages

Verification: SMT solver

- fully automatic prover
- quite powerful for integers and algebraic types

Functional Logic Programming with Curry

Curry: Haskell syntax, logic features (non-determinism)

Functions: concatenating lists

```
[]     ++ ys  =  ys
(x:xs) ++ ys  =  x : (xs ++ ys)
```

Non-determinism: list insertion + permutation

```
ins x ys     = x : ys
ins x (y:ys) = y : ins x ys
```

```
> ins 0 [1,2]  ⤳  [0,1,2] ? [1,0,2] ? [1,2,0]
```

```
perm []     = []
perm (x:xs) = ins x (perm xs)
```

Contracts for Curry [PADL'12]

Given: f `::` $\tau \to \tau'$

Contract for f: pre- and postcondition

Precondition:

f'`pre ::` τ $\to$ `Bool`

Postcondition:

f'`post ::` τ $\to$ τ' $\to$ `Bool`

Dynamic contract checking

Curry preprocessor transforms contracts into dynamic checks:

- precondition $\leadsto$ check arguments before each call
- postcondition $\leadsto$ check arguments/result after evaluation

Contract Verification: Motivation

Factorial operation with contract:

```
fac :: Int  → Int
fac n = if n==0 then 1
                else n * fac (n-1)

fac'pre  n   = n >= 0
fac'post n f = f > 0
```

Consider evaluation of *f n*:

- without contract checking: *n* calls
- with contract checking: *n* calls $+ 2 * n$ contract calls

Contract Verification

Verifying precondition

```
fac n = if n==0 then 1
                else n * fac (n-1)

fac'pre n = n >= 0
```

Consider recursive `fac` call:

`n>=0` (by precondition)

`¬(n==0)` (since `else` branch is chosen)

$n>=0 \land \neg(n==0) \implies (n-1)>=0$ (by SMT solver)

$\leadsto$ precondition of recursive call always satisfied, omit at run-time

Contract Verification

Verifying postcondition

```
fac n = if n==0 then 1
                else n * fac (n-1)

fac'post n f = f > 0
```

Consider value of right-hand side:

1. `then` branch: $1 > 0 \rightsquigarrow$ postcondition satisfied
2. `else` branch:
   `n>=0` (by precondition)
   `¬(n==0)` (since `else` branch is chosen)
   `fac(n-1)>0` (by postcondition)
   `n>=0 ∧ ¬(n==0) ∧ fac(n-1)>0 ⟹ n*fac(n-1)>0` (by SMT)
   $\rightsquigarrow$ postcondition satisfied

Altogether: postcondition always satisfied, omit at run-time

Combining pre- and postcondition verification

```
fac :: Int  → Int
fac n = if n==0 then 1
                else n * fac (n-1)

fac'pre  n   = n >= 0
fac'post n f = f > 0

g n = fac (fac n)
```

Consider outermost call to `fac` in `g`:

`(fac n)>0` (by postcondition)

`(fac n)>0` $\Longrightarrow$ `(fac n)>=0` (by SMT)

$\leadsto$ omit precondition check for this call

Formalization

For simplicity: use normalized FlatCurry representation

```
fac(n) = let x = 0
             y = n==x
         in case y of True   → 1
                      False  → let n1 = n - 1
                                   f  = fac n1
                               in n * f
```

⇝ natural semantics for Curry [Albert et al. JSC'05]

⇝ paper: include contract checking

Abstract assertion-collecting semantics

1. compute with symbolic values instead of concrete ones
2. collect properties that are known to be valid
3. do not evaluate functions (termination!)
   but collect their pre- and postconditions

Abstract Assertion-Collecting Semantics

Val $\Gamma : C \mid z \leftarrow v \Downarrow C \wedge z = v$ where v constructor-rooted or v variable not bound in Γ

VarExp $$\frac{\Gamma : C \mid z \leftarrow e \Downarrow D}{\Gamma[x \mapsto e] : C \mid z \leftarrow x \Downarrow D}$$

Fun $\Gamma : C \mid z \leftarrow f(\overline{x_n}) \Downarrow C \wedge f'\texttt{pre}(\overline{x_n}) \wedge f'\texttt{post}(\overline{x_n}, z)$

Let $$\frac{\Gamma[\overline{y_k \mapsto \rho(e_k)}] : C \mid z \leftarrow \rho(e) \Downarrow D}{\Gamma : C \mid z \leftarrow let\ \{\overline{x_k = e_k}\}\ in\ e \Downarrow D}$$ where $\rho = \{\overline{x_k \mapsto y_k}\}$ and $\overline{y_k}$ fresh

Or $$\frac{\Gamma : C \mid z \leftarrow e_1 \Downarrow D_1 \qquad \Gamma : C \mid z \leftarrow e_2 \Downarrow D_2}{\Gamma : C \mid z \leftarrow e_1\ or\ e_2 \Downarrow D_1 \vee D_2}$$

Select $$\frac{\Gamma : C \mid x \leftarrow x \Downarrow D \quad \Gamma : D_1 \mid z \leftarrow e_1 \Downarrow E_1 \ \ldots\ \Gamma : D_k \mid z \leftarrow e_k \Downarrow E_k}{\Gamma : C \mid z \leftarrow case\ x\ of\ \{\overline{p_k \to e_k}\} \Downarrow E_1 \vee \ldots \vee E_k}$$

where $D_i = D \wedge x = p_i$ $(i = 1, \ldots, k)$

Assertion-Collecting Semantics: Example

```
fac(n) = let x = 0
             y = n==x
         in case y of True   → 1
                      False  → let n1 = n - 1
                                   f  = fac n1
                               in n * f
```

Collected assertions for right-hand side:

n>=0 ∧ x=0 ∧ y=(n=x) ∧
((y=True ∧ z=1) ∨ (y=False ∧ n1=n−1 ∧ f>0 ∧ z=n∗f))

$\Longrightarrow$ z>0 (by SMT solver)

⇝ postcondition verified

Assertion-Collecting Semantics: Main Result

The correctness of our approach is justified by

Theorem (Correctness of abstract assertion collection)

Let e be an expression. If

- *e evaluates to v and*
- *the abstract semantics collects, for $z = e$, the assertion C,*

then $z = v \Rightarrow C$.

More Examples

```
last [x]        = x
last (_:x:xs) = last (x:xs)

last'pre xs = not (null xs)
```

Omit precondition of recursive call if

$$not\ (null\ xs) \wedge xs = (y{:}ys) \wedge ys = (z{:}zs) \Longrightarrow not\ (null\ (z{:}zs))$$

$\leadsto$ by evaluating right-hand side to *true*

More Examples

Select *n*th element of a list

```
nth (x:xs) n | n==0 = x
             | n>0  = nth xs (n-1)

nth'pre xs n = n>=0 && length (take (n+1) xs) == n+1
```

Omit precondition of recursive call if

$$n \geq 0 \land \mathit{length}\,(\mathit{take}\,(n+1)\ xs) = n+1 \land xs = (y{:}ys) \land n \neq 0 \land n > 0$$

implies

$$(n-1) \geq 0 \ \land \ \mathit{length}\,(\mathit{take}\,((n-1)+1)\ ys) = (n-1)+1$$

(by SMT solver with axiomatization of operations `length` and `take`)

Implementation

Contract verification tool

1. Curry preprocessor performs source-level transformation: add contracts as run-time checks
2. Preprocessed program transformed into FlatCurry program
3. For each contract: extract proof obligation
4. For each proof obligation: translate into SMT-LIB format and send to SMT solver (here: Z3)
5. If SMT solver proves validity: remove check from FlatCurry program

Benchmarks

Run time in seconds (0.00 $\approx < 10ms$)

Expression	dynamic	static+dynamic	speedup
`fac 20`	0.00	0.00	n.a.
`sum 1000000`	0.84	0.22	3.88
`fib 35`	1.95	0.60	3.23
`last [1..20000000]`	0.63	0.35	1.78
`take 200000 [1..]`	0.31	0.19	1.68
`nth [1..] 50000`	26.33	0.01	2633
`allNats 200000`	0.27	0.19	1.40
`init [1..10000]`	2.78	0.00	$>$277
`[1..20000] ++ [1..1000]`	4.21	0.00	$>$420
`nrev [1..1000]`	3.50	0.00	$>$349
`rev [1..10000]`	1.88	0.00	$>$188

`init`, `(++)`, `nrev`, `rev`: postcondition on length of lists

Conclusions

Combining static and dynamic contract checking

- support reliable software by adding contracts (more complex than standard types)
- disadvantage: run-time overhead
- avoid overhead by static verification
- if successful:
  - run time reduction ($\leadsto$ benchmarks)
  - higher confidence in overall quality of application

Future work

- more examples...
- better contract reduction: omit verified parts of contracts
- use counter-examples from verifier to check actual contract violation
- integrate abstract interpretation techniques for better precision