Compositional and Lightweight Dependent Type Inference for ML

Center for Science of Information
NSF Science and Technology Center

He Zhu and Suresh Jagannathan
Computer Science Department of Purdue University

Motivation

- Consider an open-source bit-vector library
- Procedure has complex invariants
  - ★ `((ofs2+n)-1)/b) < v2.length` discovered to be a pre-condition of procedure unsafe_blit.
  - ★ Popeye, our tool, discovers this invariant only using counterexample guided refinement
- Bug detection

```
fun blit {bits=b1, length=l1} {bits=b2, length=l2}
        ofs1 ofs2 n =
  if n < 0 || ofs1 < 0 || ofs1 + n > l1
          || ofs2 < 0 || ofs2 + n > l2
  then assert false
  else unsafe_blit b1 ofs1 b2 ofs2 n
```

guard condition requires offset value and number of bits to be copied be positive, and range of the copy to fit within source and target vectors

access element out of bound

provide an explicit counterexample witness to the bug

{length (b1)=2, length (b2)=0, l1=60, ofs1=32, l2=0, ofs2=0, n=0}.

but unsafe_blit attempts to access the offset (say 0) in the target array before initiating the copy loop leading to an array out-of-bounds exception

Refinement and Dependent Types

```
fun f g x =
  if x>=0 then
    let r = g x in r end
  else
    let p = f g
        q = compute x
        s = f p q
    in s end
```

```
fun main h n =
  let r = f h n
  in assert (r >= 0) end
```

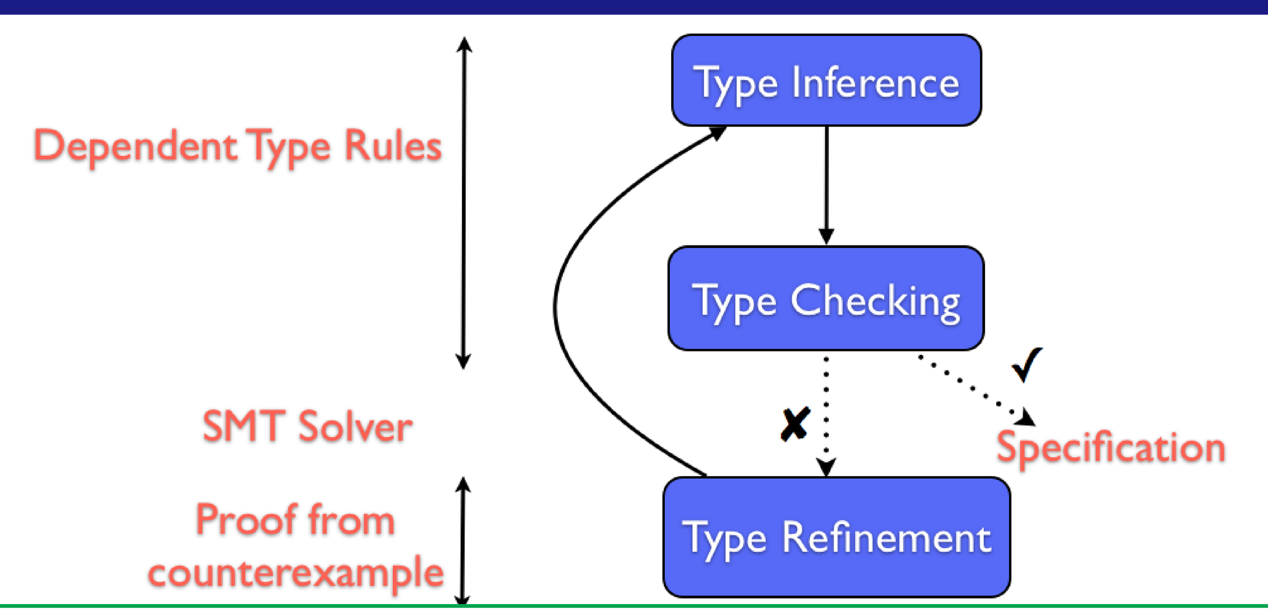
Associate the type:

$$f : \{g : \{g_{arg} : \{true\}\} \to \{\nu \geq 0\}\} \to x : \{true\} \to \{\nu \geq 0\}\}$$

to f

Extend standard types with predicates (sometimes called logical qualifiers) that refer to program variables, primitive functions and the special variable (ν)
Well-typed program implies correctness

Framework



Inference and Checking

```
fun f g x =
  if x>=0 then
    let r = g x in r end
  else
    let p = f g
        q = compute x
        s = f p q
    in s end
```

```
fun main h n =
  let r = f h n
  in assert (r >= 0) end
```

Inference follows typing rules which are quite like tranditional typing rules

Subtype constraint built for this application leads to the verification condition:

$$((\mathrm{x} \geq 0 \wedge r = R_g(x)) \Rightarrow \nu = r) \wedge ((\neg(x \geq 0) \wedge s \geq 0) \Rightarrow \nu = s) \Rightarrow (\nu \geq 0)$$

path predicate found in Γ

refinement associated with returns

established by verification of `main`

Counterexample path

- Negation of the verification condition is supplied to SMT which may produces counterexample as satisfiable assignment
- Build counterexample paths from counterexamples
  - ★ "`if` p `then` e_t `else` e_f" translates to "`assume` p`;` e_t" if VC assignment evaluates p to true

Verification condition:

$$((x \geq 0 \wedge r = R_g(x)) \Rightarrow \nu = r) \wedge ((\neg(x \geq 0) \wedge s \geq 0) \Rightarrow \nu = s) \Rightarrow (\nu \geq 0)$$

Counterexample path:

```
fun f g x = assume (x >= 0); let r = g x in r
```

assuming counterexample as r = -1 and x = 1

Refinement: WP Generation

```
fun f g x =
  if x>=0 then
    let r = g x in r end
  else
    let p = f g
        q = compute x
        s = f p q
    in s end
```

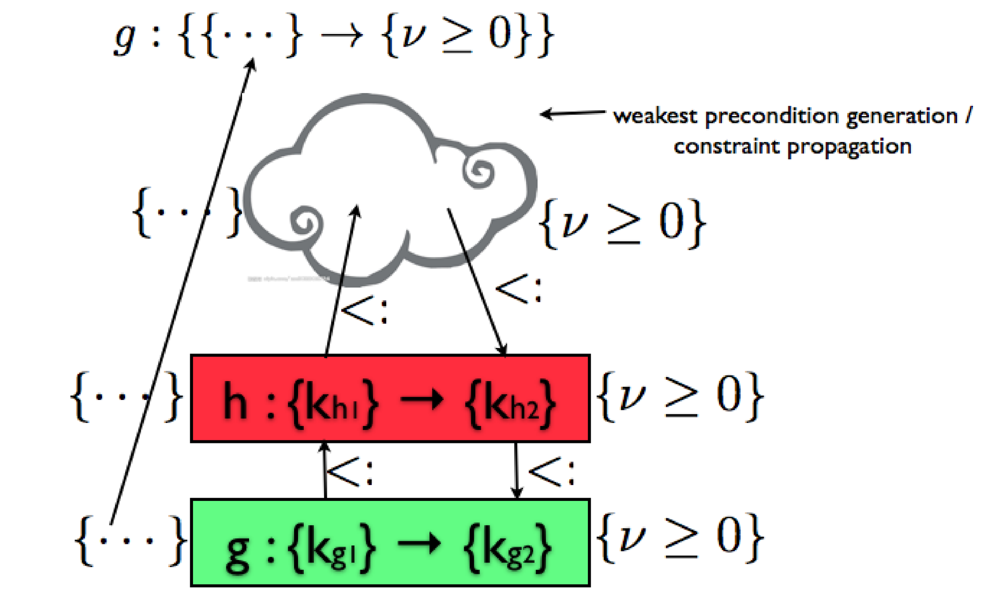
```
fun main h n =
  let r = f h n
  in assert (r >= 0) end
```

$$\mathrm{wp}(assume(x \geq 0); \text{let } r = g\ x \text{ in } r), \nu \geq 0) =$$
$$\mathrm{wp}(assume(x \geq 0), \mathrm{wp}(\text{let } r = g\ x \text{ in } r, \nu \geq 0)) =$$
$$\mathrm{wp}(assume(x \geq 0), \mathrm{wp}(r = g\ x, (\mathrm{wp}(\nu = r, \nu \geq 0))))$$
$$\mathrm{wp}(assume(x \geq 0), \mathrm{wp}(r = g\ x, r \geq 0))$$
$$\mathrm{wp}(assume(x \geq 0), R_g(x) \geq 0) =$$
$$x \geq 0 \Rightarrow R_g(x) \geq 0$$
$$g : \{\{true\} \to \{\nu \geq 0\}\}$$

Refinement: Constraint Propagation

```
fun f g x =
  if x>=0 then
    let r = g x in r end
  else
    let p = f g
        q = compute x
        s = f p q
    in s end
```

```
fun main h n =
  let r = f h n
  in assert (r >= 0) end
```



Benchmarks

Program	num_ref	num_cegar	prover_call	cegar_time	run_time
fhnhn	3	4	35	0s	0.014s
neg	15	20	230	0.004s	0.18s
max	10	11	175	0.005s	0.95s
r-file	11	21	205	0.012s	1.56s
r-lock	10	18	108	0.006s	0.60s
r-lock-e	13	18	113	0.01s	0.68s
repeat-e	39	18	237	0.11s	4.87s
list-zip	2	4	149	0.01s	1.55s
array-init	35	106	3617	0.03	102.3s

Small benchmarks (< 100 LOC) but with complex control- and dataflow
Lots of HO procedures

Non-trivial qualifiers that are not included in DSolve's basic qualifier set

Two buggy programs for which we can provide explicit witnesses

Mochi (HOMC) could not synthesize invariants for array-init

Related Work

- Liquid Types (Rondon et. al [PLDI'08], Kawaguchi et. al [PLDI'09])
  - ★ Qualifier discovery vs. selection
- Higher-Order Program Model Checking (Kobayashi et. al [PLDI'11]),
  - ★ First-order vs. higher-order verification engine
- Dependent Types from Counterexamples (Terauchi [POPL'10])
  - ★ Concrete counterexample paths vs. abstract program slices
- Verifying Functional Programs using Abstract Interpreters (Jhala et. al, [CAV'11])
  - ★ Program analysis vs. program transformation