A Theory For Semantic Type Checking

Motivation: if we compile/optimize a typed program we are often unable to use the source-level type system to type check the target program.

Examples:

(1) • after ~~register~~ allocation a given variable is used with different types

```
f(x: int) {
    x ← x ==int 7;
    x ← not x
    assert (x: bool)   ← postcondition
}
```

• we want to check (just like in the source) that operations are used with ~~the~~ appropriate operands:

$==_{int}$	both operands are int and result is bool
not	operand and result is bool

• we cannot use a type system based on declarations

• we need to use a data-flow based type checker

• or VCGen!

Assume that we have predicates of the form

$e : \tau$ (for now $\tau \in \{int, bool\}$)

fun(x : int)	$VC = x : int \wedge x == 7 : bool \wedge not(x == 7) : bool$
$x \leftarrow x == int\ 7$	$VC = x : bool \wedge not\ x : bool$
$x \leftarrow not\ x$	$VC = x : bool$
assert(x : bool)	

The VC for the whole function is.

$$\forall x.\ x : int \Rightarrow x : int \wedge x == 7 : bool \wedge not\ (x == 7) : bool$$

which we can prove using some simple inference rule that model the source-level type rules

$$\frac{E_1 : int \quad E_2 : int}{E_1 ==_{int} E_2 : bool} \qquad \frac{E : bool}{not\ E : bool}$$

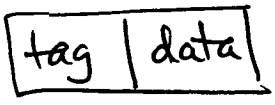
Example 2

In the process of compilation some source-level abstraction is eliminated, exposing concrete representation details

- consider union types:
- concrete representation

```
p: int + bool
case p of
   left(x)  → x ==_int 5
 | right(x) → not x
```

p is a pointer to




```
t ← *p
x ← *(p+1)
if t==0 then
    res ← x ==_int 5
else
    res ← not x
assert (res : bool)
```

VC = t==0 ⇒ x : int ∧ x==5 : bool
t≠0 ⇒ x : bool ∧ not x : bool

- we cannot assign a source-level type to x before the conditional.
- but a data-flow based approach can keep track of conditionals
- VCGen can do that too.

Example 3

- In the process of optimization the obvious mapping from concrete representation is lost
  - bounds checks removed
  - tag checking removed
  - many other optimizations
- How to prove that $3-2 : bool$?
- Cannot use syntactic rules anymore.

$$\frac{e : \tau \qquad e = e'}{e' : \tau}$$

congruence helps us do semantic type checking

Let's design a system of inference rules for formulas of the kind $e : \tau$

We will use the following language of types

$$\tau ::= int \mid bool \mid \sigma * \mid \sigma[l] \mid const(e)$$

$\sigma *$: pointers to structured type σ; $\sigma[l]$: array of length l; $const(e)$: equal to e

$$\sigma ::= \tau \mid \sigma_1 \times \sigma_2 \mid \sigma[] \mid \mu t.\, \sigma \mid t \mid \sigma_1 + \sigma_2$$

$\sigma[]$: arrays of unspecified length; t: type variable

We will want to maintain a few invariants such as

- only 0 and 1 are valid representations for booleans
- if we have a $\tau*$ then we can read and write (a τ) to that address

$$\frac{}{0 : bool} \qquad \frac{}{1 : bool} \qquad \frac{e : \tau \quad e = e'}{e' : \tau}$$

We try the rule

$$\frac{e : \tau*}{sel(m, e) : \tau}$$

but this is not sound !!
It would allow us to prove

$sel(upd(m, e, 0), e) : \tau$ from $e : \tau*$

from which we can prove $0 : \tau$!!

A big hole in the type system

The congruence rule gives us power but it destroys any abstraction !! Because of that we have to be very explicit about assumptions

We have to add an assumption that the contents of memory matches the types.

typed(m)

We need first to define the semantics of typing predicates

- we define a representation function R that associates with each base type τ a set of values in U that are valid representations of expressions of type τ

  U is the universe of all base values (machine words in assembly language)

Not all R are good. We define a set of conditions that each R must satisfy.

a) $R(int) = U$ ← all values are valid represent. of integers.

b) $R(bool) = \{0, 1\}$

c) ~~$v \in R(\sigma*) \Rightarrow$~~

$v \in R((\sigma_1 \times \sigma_2)*) \Rightarrow v \in R(\sigma_1*)$ and

~~$v+1 \in \sigma$~~

$v + |\sigma_1| \in R(\sigma_2*)$

We define the size of a structured type

$|\tau| = 1 \quad |\sigma_1 \times \sigma_2| = |\sigma_1| + |\sigma_2|$

$|\sigma[]| = 2 \quad |\mu t.\, \sigma| = |\sigma| \quad |\sigma_1 + \sigma_2| = 1 + \max(|\sigma_1|, |\sigma_2|)$

$|\cdot|$ is not defined for type variables. All t must occur inside a $\sigma*$ or $\sigma[l]$

d) $R((\sigma_1 + \sigma_2)*) = R((\text{const}(0) \times \sigma_1)*) \cup R((\text{const}(1) \times \sigma_2)*)$

e) $R(\text{const}(n)) = \{n\}$

f) $v \in R(\sigma[l])$ then $\forall i \; 0 \leq i < l$

$v + i \cdot |\sigma| \in R(\sigma*)$

g) $R((\mu t.\sigma)*) = R(([\mu t.\sigma/t]\sigma)*)$

h) $v \in R((\sigma[])*)$ then $v \notin R(\tau*)$ for any τ

i) $v \in R(\tau*)$ and $v \in R(\tau'*)$ then $\tau \equiv \tau'$

↑ this classifies all memory locations uniquely

And the meaning of typed(m) in a given representation

a) $v \in R(\tau*)$ then $v \in Dom(m)$ and $m(v) \in R(\tau)$

b) $v \in R((\sigma[])*)$ then $v, v+1 \in Dom(m)$ and

$m(v) > 0$ and

$m(v+1) \in R(\sigma[m(v)])$

Now we can give the inference rules

$$\frac{A : \sigma[L] \quad I \geq 0 \quad I < L \quad |\sigma| = S}{S * I + A : \sigma *}$$

$$\frac{A : (\sigma_1 \times \sigma_2) * \quad |\sigma_1| = S_1}{S_1 + A : \sigma_2 *}$$

$$\frac{A : (\sigma_1 \times \sigma_2) *}{A : \sigma_1 *}$$

$$\frac{A : \tau * \quad \text{typed}(M)}{\text{sel}(M, A) : \tau}$$

(works for a τ but not for a σ)

$$\frac{\text{[redacted]}}{E : \text{const}(E')}$$

$$\frac{A : (\sigma[]) * \quad \text{typed}(M)}{\text{sel}(M, A+1) : \sigma[\text{sel}(M, A)]}$$

$$\frac{}{0 : \text{bool}} \qquad \frac{}{1 : \text{bool}}$$

$$\frac{A : (\sigma_1 + \sigma_2)* \quad \text{typed}(M) \quad \text{sel}(M, A) = 0}{A+1 : \sigma_1 *}$$

$$\frac{\text{typed}(M) \quad A : \tau* \quad E : \tau}{\text{typed}(\text{upd}(M, A, E))}\ \text{upd}$$

We can even prove the soundness of these rules.
Take the upd rule

hyp.: $R \Vdash \text{typed}(M) \quad A \in R(\tau*) \quad E \in R(\tau)$

Need to prove

a) $\forall v \in R(\tau'*) \quad v \in \text{Dom}(m') \quad m'(v) \in R(\tau')$
where $m' = \text{upd}(m, A, E)$

a.1. $v = A$ then, by inv. i $\tau = \tau'$
$m'(v) = E$ ✓

a.2. $v \neq A$ then $m'(v) = m(v)$ ✓
$(\text{Dom}(m') = \text{Dom}(m))$

b) $\forall v \in R((\sigma[])*)$ then $v, v+1 \in \text{Dom}(m')$
$m'(v) > 0$
$m'(v+1) \in R((\sigma[m(v)])*)$

but $v \neq E \Rightarrow m'(v) = m(v) > 0$

Now $v+1 \in R((\sigma[m(v)])*)$ and by a) above we have q.e.d.

A proof procedure for typing predicates

- tactic based, reversing the inference rules
- cannot just do backwards chaining because there are vars in hyp. that do not appear in conclusion.
- We need a function that given A it finds out all potential τ such that $A : \tau *$

The idea is that pointers can be created in a limited nr. of ways

A is $A' + S$	. S is a constant → second element of a pair → index in an array
A is $sel(M, A')$	. try to prove that $A' : \tau *$
A is variable	. we must find the type of A within assumptions.

- This is not going to be complete (what if a pointer is computed in a convoluted way

```
f(int * x)
    * (x + alwaysZero());
}
```