

## CMSC 838Z - Language-Based Security (2/23/2005)

In TAL-1, we have shared pointers and unique pointers. Shared pointers (**ptr**) are used for supporting aliasing. The types of shared pointers remain unchanged during program execution. For unique pointers (**uptr**), the types can be updated but the pointers cannot be aliased. The stack pointer (**sp**) is an example of a unique pointer. For registers, it is not necessary to make this distinction since they are concrete locations, not memory (and thus cannot be aliased).

Here's an example of the problems of tracking aliasing:

```

x = ref 1           // ref ρ
y = x              // ref ρ
if rand()
    z = ref 6 // ref ρ'
    z = y

```

Then, the type for  $z$  could either be **ref**  $\rho$  or **ref**  $\rho'$  depending on which branch was taken. Thus we cannot know in advance what pointers are aliased (in general). We have to be conservative. I.e. we can generate a third location  $\rho''$  where  $\rho \leq \rho''$  and  $\rho' \leq \rho''$ , meaning that  $\rho''$  is an approximation for both original locations. Thus  $z$  has type **ref**  $\rho''$ , and the possible aliasing relationships are captured in the above constraints.

In page 158, fig. 4-5 shows the syntax of machine. The pointers in heap are shared pointers only. For example, let the heap  $H = \{l_1 \rightarrow \langle 1, 2 \rangle, l_2 \rightarrow \text{jump } l_2, \dots\}$ . Then, we can have

```

r1 = l1
r2 = Mem[r1 + 1] // i.e. r2 = 2
r3 = uptr(1)

```

In page 159, the rewriting rules prevent **uptr** becomes aliased. If  $r = \text{uptr}(h)$ , then  $r$  never can be copied. Alternative approach: after reading **uptr**, give it a type that makes it unreadable.

The rules only check stack overflow. We also have to check heap for overflow error. It is too difficult to rule out all possible errors in compile time. We need checking in runtime.

In the copy example in page 160, the input and output types should not be restricted to **int**. They can be in a more general form as shown below.

```

Input:    ∀α, β, ρ, δ, η . {r1 : ptr(α, β, ρ), r2 : δ, r3 : η}
Output:   ∀α, β, ρ . {r1 : ptr(α, β, ρ), r2 : ptr(α, β), r3 : β}

```

Although the routine only copies the first two values of the tuple pointed by  $r_1$ , the input tuple can have two or more elements. We use  $\rho$  to represent the type of the tail.

In TAL-1, we have new types shown in fig. 4-6 and new typing rules in fig. 4-7 and fig. 4-8. In addition to S-UPTR, we also need a rule for **ptr** type. In

S-MALLOC, why the result type is `uptr` to `int`? It is because we do not know the output type. We simplify the real life system to a tractable problem. Consider the example:

```

Let  $H$  =  $\{l_1 \rightarrow \text{jump } l_1, l_2 \rightarrow \text{code}(\dots)\}$ 
 $r_d$  = malloc 2 //  $r_d \rightarrow \text{uptr}(l_1, l_1)$ 
 $r_1$  = Mem[r_d]
 $r_1$  =  $r_1 + 5$  // we have a problem here

```

Note that `int` is safer than `ptr`. We also have to put such convention to the abstract machine. The type rules should correspond to the abstract machine. If we do not have such correspondence, we may not have problem in real system but it is a flaw to the proof. For solving this problem, we can introduce a junk type, which is not associated with any useful operation.

In the load and store rules (i.e. S-LDS, S-LDU, S-STS and S-STU), we have to make sure there are  $n$  values in the tuple. In S-STS, the tuple is pointed by a `ptr`. The type of  $r_s$  must be the same as the type of `Mem[r_d + n]`. Both of them are of the type  $\tau_n$ . In S-STU, the types of  $r_s$  is  $\tau$ , which can be different  $\tau_n$  (the type of `Mem[r_d + n]`) since  $r_d$  is a `uptr`.

In S-SALLOC, we must make sure  $n \geq 0$ . However, there is no corresponding rules in the abstract machine. This is a bug of the system (i.e., there SHOULD be a corresponding rule in the abstract machine). In S-SFREE, the tuple must have at least  $n$  elements. Consider the instruction `sfree 3`. If the type of `sp` is  $\forall \rho . \rho$ , there may be an underflow problem. We must make sure the stack has at least three elements. For example, the type of `sp` can be  $\forall \alpha, \rho . \langle \text{int}, \text{int}, \alpha, \rho \rangle$ .

For the rules defined with  $\sigma$ , the rules indeed define a whole family of rules.

We can solve such problem by sub-typing. Consider the following example:

$$\begin{aligned} & \tau_1 \times \tau_2 \\ & \tau_1 \times \tau_2 \times \tau_3, \forall \tau_3 \end{aligned}$$

Then,  $\tau_1 \times \tau_2 \times \tau_3 \leq \tau_1 \times \tau_2$ , which means the third argument is optional. Polymorphism is better than sub-typing in the sense that if we have  $\{\alpha, \beta, \rho\} \rightarrow \{\alpha, \beta, \rho\}$  in polymorphism, it says that the type  $\rho$  remains unchange in input and output. However, we cannot represent such semantics in sub-typing. The optional types may not be matched.

In the example in page 165, the calling type and return type are encoded in the type system. It is a caller-save example. How to do callee-save? We require the return type to be the same as input (i.e.  $r_2 : b, r_3 : c$ ). If  $r_2$  is overwritten by an `int`, then we cannot return with type checked. We must return with the original value of  $r_2$  since only the original value has the type  $b$ .

Type checker takes advantage in compiling from high level language. If the machine codes are modified arbitrarily, the type checker may fail it even if non-trivial proofs of type checked do exist. The type checker still has the soundness property.

At last, we point out some drawbacks of TAL-1. TAL-1 does not support objects and closures. It does not work well on dynamic contents. There is no

instruction for releasing allocated memory. It does not handle dangling pointer problem. There is no way to uncommit shared pointers. It needs a garbage collector to free up memory.