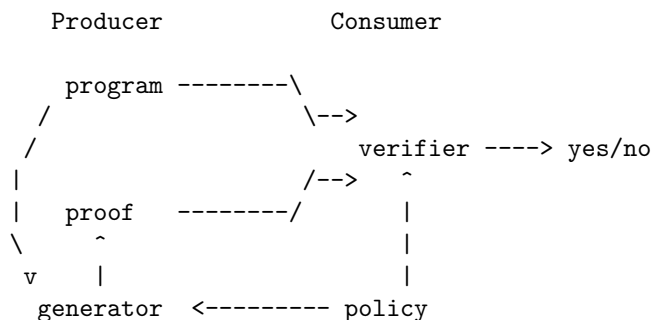


Basics:

The overall structure of PCC:



The syntax: Apart from the standard logical operators we augment the logic with the sel and upd expressions to work with memory types. sel $E_m E_a$ represents the value held at location E_a in memory E_m . upd $E_m E_a E_v$ updates location E_a in memory E_m to the value E_v . sel and upd are the ways you read and write memory declaratively (i.e. generate a new memory each time you do a memory operation). listinv is a formula that checks that memory is in a sane state.

Exercise 5.2.1: Singleton type representation in the new logic.

We had parametrized singleton types in to allow us to stick in values into types. Each parameter qualified type would represent just one value. The proof rule version of a singleton type is like $\{y \mid y = v\}$ which is equivalent in flavor to $S(v)$.

Exercise 5.2.2: An alternative representation of the maybe pair type with tags. The assertion: $\{y \mid y = \text{ptr}\{\{x \mid x = 0\}; \text{int}\} \vee y = \text{ptr}\{\{x \mid x = 1\}; \text{int}; \text{int}\}\}$

Proof rules:

Figure 5-5 in the text provides the proof rules that shall be used to reason about the safety policies which are stated to hold on blocks of code which we discussed later in class. There are the obvious AND and IMPL rules as in standard logic. We also have the rules that define the semantics of sel and upd primitives. MEM0 states that if the location being accessed has been updated then the return value of a sel is the updated value. MEM1 states that if it has not been updated then the result is just a recursion (sel) into the old memory.

In type systems we used safety policies to guarantee that a type safe program is never stuck. This time around we have an additional level of redirection by using the verifications conditions. If the VCs are provable then the program never gets stuck.

For the safety policy in question the list of proof rules is presented in Figure

5-6 in the text. The only rule requiring comments is the UPD rule:

$$\frac{\text{listinv } M \ A : \text{ptr}\{W\} \ V : W}{\text{listinv}(\text{upd } MAV)}$$

If we only have reads from memory then initial well formed memory remains well formed. But if we update then we need a rule which only allows updates which preserves the types of the pointers into memory. The rule states exactly that.

Safety Pre and Post conditions:

PCC's safety policy requires that there be pre and post conditions on the assembly language. They are expressed as logical formulae on the registers which hold the input and output values. Memory is kept in r_m . The safety precondition as mentioned for the agent in the text is $r_x : \text{mp_list} \wedge \text{listinv } r_m$. And the post condition is simply $\text{listinv } r_m$. The thing to note is that we only require that the function be given a maybe pair list and that at return the memory is sane. We actually do not care about whether the function computes what it claims to compute. We are just concerned about memory safety and that is what we guarantee.

Exercise 5.2.3: The pre and post conditions on the OCaml type: $(\text{int} * \text{int}) * \text{int list} \rightarrow \text{int list}$.

r_1 : first argument to function.
 r_2 : second argument to function.
 r_R : register to hold the return value.

Precondition: $r_1 : \text{ptr } \{\text{int}; \text{int}\} \wedge r_2 : \text{list } \{\text{int}\}$
 Postcondition: $r_R : \text{list } \{\text{int}\}$

Exercise 5.2.4: The pre and post conditions for a function that takes a ptr to a sequence of int lists and the length of the sequence.

Precondition: $\text{listinv } r_m \wedge (\forall i (i \geq 0 \wedge i < r_2) \Rightarrow r_1 + 4i : \text{ptr } \{\text{list int}\})$
 Postcondition: $\text{listinv } r_m //$ memory is sane after the function