1 Conservative Analyses

A conservative analysis is an analysis that finds a superset of all possible errors. Thus, if the program contains an error of type α, a conservative analysis will find all errors of type α.

Let's consider a specific example. Suppose we want to write an analysis that will find dereferences of pointers that have been deallocated with the function `free`. A non-conservative analysis may find some such dereferences, but it may ignore others. If a program uses aliasing, for example, a non-conservative analysis may not include an alias analysis. Thus, if a deallocated pointer has an alias that is dereferenced, the non-conservative analysis will miss an error.

A conservative analysis will not miss this error. In the context of freed pointers, a conservative analysis to find dereferences of such pointers must include an alias analysis that is also conservative.

At a more concrete level, if our analysis thinks it is possible that a pointer is freed, it must assume the worst. Thus, suppose we have a pointer and we cannot tell if it is freed or not, or maybe it was freed along one path to the current program point and not along an another. The safe assumption is that this pointer is not dereferenceable because there may be a scenario where the pointer was freed. Conservative analyses will over-report errors. They may report errors that cannot be triggered during an execution of the program, but they will not miss an error that could be triggered during an execution of the progrm.

2 Lambda Reduction

In lambda calculus, function application is achieved by renaming and substitution. Rename bound variables so there are no name conflicts, then substitute. Some specific hints are:

1. Parenthesize arguments before the application.
2. Rename during the intermediate steps. It is easiest if you always make sure that your bound variables have different names.
3. Apply arguments one at a time. Each application removes one $\lambda\ arg.exp$ and causes a substition in exp.

3 Denotational Semantics

We went over the semantics for `x := x + 1; x := x + 2;` and argued that it is equivalent to the semantics for `x := x + 3;`. I will reproduce the semantics for `x := x + 1; x := x + 2` here.

$$\begin{aligned}&\mathbf{C}[\![\mathtt{x := x + 1; x := x + 2;}]\!]\sigma = \mathbf{C}[\![x := x + 2]\!](\mathbf{C}[\![x := x + 1]\!]\sigma)\\ &= \mathbf{C}[\![x := x + 2]\!](modify(\sigma, x, \alpha_x + 1))\end{aligned}$$

We are assuming that our initial state, σ, is a function that maps each variable v to some arbitrary constant, α_v. If that is confusing, just substitute 0 for α_x wherever it appears. Continuing:

$$
\begin{aligned}
&Let \sigma_1 = modify(\sigma, x, \alpha_x + 1)\\
&= \mathbf{C}[\![\texttt{x} := \texttt{x} + 2;]\!]\sigma_1\\
&= modify(\sigma_1, x, \mathbf{E}[\![\texttt{x}]\!]\sigma_1 + 2)\\
&= modify(\sigma_1, x, \alpha_x + 1 + 2)\\
&= modify(\sigma_1, x, \alpha_x + 3)
\end{aligned}
$$

Thus, the program fragment we have analyzed adds 3 to the initial value of `x`. Analyzing `x := x + 3` reveals that this expression has exactly the same semantics.