Lecture Notes on Purity Checking

15-411: Compiler Design
Frank Pfenning

Lecture 19
October 30, 2014

1 Introduction

In this lecture we discuss purity checking, which analyzes functions to determine if they are "pure", where purity can take slightly different meanings in different contexts. It is a good example for a program analysis which can be relevant in the front end to provide errors or warnings, and also in the back end for optimizations.

C0 uses purity checking in the front end to rule out contracts that change the behavior of programs. Such programs could behave differently with and without contracts even if there are no contract violations, which goes against their intent.

In the back end purity checking can be used to determine if function calls may represent dead code (something we did not account for in Lecture 5), or if a function call may affect heap content (see Lecture 18, page L18.4).

Intraprocedural optimizations are those that confine themselves to analyzing and optimizing one function at a time. Because of the locality of the analysis and transformation, these are designed to improve loops, basic operations, register allocations, register moves, etc., and work well with local variables. However, functions may share the heap in which larger data structures are allocated. Optimizations that try to optimize memory consumption or accesses are therefore less effective if they consider code only one function at a time. Thus the need for so-called *interprocedural analysis and optimizations*. We have already seen inlining and tail-call optimization as particular case that involve two functions and can avoid or drastically reduce the overhead of function calls. Perhaps even more importantly, they enable further optimizations. Knowing when cells in the heap may or may not be modified is similarly important.

2 Purity Checking Contracts

The C0 language has a built-in function `\length(e)` for an expression e of type $\tau[]$. However, this function can only be called from inside contracts in order to be consistent with C where the length of arrays cannot be calculated at runtime. Clever students in the course on Principles of Imperative Computation have devised the following hack to get around this restriction:

```
bool store_int(int* p, int n) {
  *p = n;
  return true;
}

int get_length(int[] A) {
  int* p = alloc(int);
  //@assert store_int(p, \length(A));
  return *p;
}
```

However, this is deeply flawed: while `get_length(A)` will indeed return the length of A if the code is compiled with contract-checking enabled (`cc0 -d`), it will always return 0 when contract-checking is disabled. In the latter case the `//@assert` is ignored and the default value of type `int` is returned.

In order to flag such code as incorrect, the cc0 compiler checks that functions that are called in contracts are *pure*. Such functions (and contracts in general) are still allowed to raise exceptions or even have side effects such a printing, but they may not mutate memory that has been allocated before they are called. It is permitted to mutate memory allocated inside pure functions, because this will not affect the outcome of computation if contracts are erased.

Contracts may raise an exception (which we allow—it is one purpose of contracts to raise exceptions if they are not satisfied) and they could fail to terminate (which we also allow). Otherwise, they can only affect memory through function calls, since assignments in C0 are statements rather than expressions. So consider a function call $f(e_1, \ldots, e_n)$ inside a contract. It is permitted if all heap locations that are reachable from the argument e_i are not modified by f or any other functions it might call. In other words, all memory reachable from the arguments must be *read-only*. "Reachability" here refers to being able to access array elements, struct fields, and dereferencing pointers.

3 Purity Checking for Memory Optimizations

Consider the following code segment in three-address form.

$$\begin{array}{lcll} t & \leftarrow & M[a] & \\ r & \leftarrow & f(s_1, \ldots, s_n) & \\ t' & \leftarrow & M[a] & \text{(replace by } t' \leftarrow t\text{?)} \end{array}$$

We can optimize away the memory access if we can be sure that the call to f will not modify $M[a]$, the memory at address a. As we discussed in lecture, if all of the s_i are booleans or integers, this is guaranteed in C0 since we have no global variables. A refinement is to allow addresses to be passed but check f to make sure that neither those addresses nor any other heap addresses reachable from them are modified. A further improvement on that would be to allow memory modifications, but only if they can be shown not to alias with $M[a]$.

We take here the middle ground, because this coincides with the information we obtain from purity checking as motivated in the previous section. The last refinement would be aided both by type-based and by allocation-based global alias analysis which we do not detail here.

4 Purity Analysis as a Type System

When designing program analyses we first need to consider which representation to define it on. Generally speaking, a program analysis that is designed for programmer feedback should be done earlier in the compiler, while an analysis done to enable optimization should be done later. Because purity analysis can be used to report errors for impure contracts, we provide it here on the abstract syntax after elaboration. If we also want to use it for optimization we can preserve that information and then reference it while optimizing in the middle end of the compiler.

The second question is how to define the analysis. Generally, this can be broken down into (a) a specification, and (b) an implementation. For analyses on abstract assembly, the specification so far has been mostly via inference rules, while the implementation simply runs them to saturation. We illustrate here a different technique, whereby the specification of the property in question is as a type system, and the implementation is a form of type inference. This is particularly common for analyses applied near the front end of a compiler, usually on the abstract syntax.

For simplicity, we omit structs from the formal development and focus on pointers and arrays. We will make some remarks on extensions with structs in Section 6. So we have:

$$\text{Types} \quad \tau \quad ::= \quad \mathsf{int} \mid \mathsf{bool} \mid \tau* \mid \tau[\,]$$

We now want to generalize this type language to track whether memory is read-only or may be mutated. Since $\tau*$ and $\tau[\,]$ reference memory, only these types are

annotated as such. We write τ^r* for the type of a cell that is read-only, and similarly $\tau^r[\,]$ for an array whose elements are read-only. We write m indicate mutable memory.

$$\begin{array}{lrcl} \text{Permissions} & p & ::= & m \mid r \\ \text{Permission Types} & \tau & ::= & \mathsf{int} \mid \mathsf{bool} \mid \tau^p* \mid \tau^p[\,] \end{array}$$

Our next task will be to rewrite the typing rules to take permissions into account. One important property will be that if all types are mutable, then all programs that check with regular types should also check with purity types. In other words, read-only permissions impose a restriction on programs, in accordance with our goal to restrict functions called from contracts to be pure.

We begin with the typing of expressions. Recall that the typing judgment has the form

$$\Gamma \vdash e : \tau$$

where Γ is a context assigning types to variables, written as $x_i{:}\tau_i$. Expressions only read the content of memory, except possibly through a function call, so the rules are for the most part straightforward.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \mathsf{int}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \texttt{ == } e_2 : \mathsf{bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (e_1 \texttt{ ? } e_2 : e_3) : \tau}$$

The last two rules, for equalities and conditionals, raise a question: what if the types of two expressions have the same regular type, but their permissions vary? Let's consider an example:

```
bool cond = ...;
int[] A = ...;
int[] B = ...;
int[] C = cond ? A : B;
```

Which combinations of permissions for A, B, and C make sense? If A and B have the same permission p, then C should clearly have the same permission. If one of them (say, A) is read-only (r) and the other mutable (m), should the expression type-check, and would should be the permission of C? Since we do not statically predict the condition `cond`, it is possible for C to alias A. This means if we were permitted to write to elements of C, then we might actually write to elements of A. But A is read-only, so we must prohibit this. In other words, if A is read-only and B is mutable, then C must be read-only.

We can leave the rules for conditionals and equality tests as they are, if we add general rules that allow us to treat mutable memory as if it were read-only.

$$\frac{\Gamma \vdash e : \tau^m *}{\Gamma \vdash e : \tau^r *} \qquad \frac{\Gamma \vdash e : \tau^m[\,]}{\Gamma \vdash e : \tau^r[\,]}$$

We refer to these rules as *coercions*. Of course, we do not want to use them randomly, but only in certain places where we need the flexibility like conditionals or equality tests. In our example above, we would apply the rule to $B : \tau^m[\,]$ so both branches of the conditional have the same type, and C becomes read-only as expected. We abbreviate

$$\Gamma_0 = \mathsf{cond{:}bool}, A : \mathsf{int}^r[\,], B : \mathsf{int}^m[\,]$$

and deduce

$$\frac{\dfrac{}{\Gamma_0 \vdash \mathsf{cond} : \mathsf{bool}} \quad \dfrac{}{\Gamma_0 \vdash A : \mathsf{int}^r[\,]} \quad \dfrac{\dfrac{}{\Gamma_0 \vdash B : \mathsf{int}^m[\,]}}{\Gamma_0 \vdash B : \mathsf{int}^r[\,]}}{\Gamma_0 \vdash (\mathsf{cond}\ ?\ A : B) : \mathsf{int}^r[\,]}$$

One might think function calls is an interesting case, but using the coercions it turns out to be entirely straightforward: we can require the argument types to match the parameter types exactly. If an argument e_i is the address of a mutable location, but the parameter requires it to be read-only, we can apply a coercion. Of course, the other way around is unsound because read-only memory might then be modified by the function f.

$$\frac{\tau f(\tau_1, \ldots, \tau_n) \quad \Gamma \vdash e_i : \tau_i \quad (1 \le i \le n)}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau}$$

Also, allocation returns a location that is mutable. In fact, since we likely need to initialize that location, it better be mutable.

$$\frac{}{\Gamma \vdash \mathsf{alloc}(\tau) : \tau^m *} \qquad \frac{\Gamma \vdash e : \mathsf{int}}{\Gamma \vdash \mathsf{alloc\_array}(\tau, e) : \tau^m[\,]}$$

The key, then, is the treatment of assignment, where memory is actually mutated. An assignment has the form $\mathsf{assign}(d, e)$ (written $d = e$; in concrete syntax), where d is an l-value (we call it destination here).

$$\mathsf{Destination} \quad d \quad ::= \quad x \mid *d \mid d[e]$$

As before, we are ignoring structs. We have a new judgment

$$\Gamma \vdash d :^p \tau$$

which we read as "*d denotes a memory location of type t with permission p*". We will use this as follows:

$$\frac{\Gamma \vdash d :^m \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{assign}(d, e) : [\sigma]}$$

Recall that statements s are typed as $[\sigma]$ which means that executing s might return a value of type σ. If we ignore the permission, this is exactly the ordinary rule for assignment. The permission m indicates that this location must be mutable since we write to it.

The first rule is simple: variables are allocated on the stack or held in registers, so they are always mutable.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x :^m \tau}$$

Otherwise, we use exactly the permission that the type of destination implies.

$$\frac{\Gamma \vdash d :^q \tau^p *}{\Gamma \vdash *d :^p \tau} \qquad \frac{\Gamma \vdash d :^q \tau^p [\,] \quad \Gamma \vdash e : \mathsf{int}}{\Gamma \vdash d[e] :^p \tau}$$

Notice that the permission q in the premise is ignored since we do not write to the location denoted by d in the premise. For example,

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p{:}(\mathsf{int}^m*)^r* \vdash p :^m (\mathsf{int}^m*)^r*}}{p{:}(\mathsf{int}^m*)^r* \vdash *p :^r \mathsf{int}^m*}}{p{:}(\mathsf{int}^m*)^r* \vdash **p :^m \mathsf{int}} \qquad \dfrac{}{p{:}(\mathsf{int}^m*)^r* \vdash 5 : \mathsf{int}}}{p{:}(\mathsf{int}^m*)^r* \vdash \mathsf{assign}(**p, 5) : [\_]}$$

should hold because we are allowed to mutate the location denoted by $*p$.

5 Inferring Permissions

In the source language, we still use ordinary types, so we need to infer the permissions. When we see a function call $f(e_1, \ldots, e_n)$ for a function $\tau\, f(\tau_1, \ldots, \tau_n)$ we apply the function read_only to all the parameter types:

$$\begin{array}{lcl} \mathsf{read\_only}(\mathsf{bool}) & = & \mathsf{bool} \\ \mathsf{read\_only}(\mathsf{int}) & = & \mathsf{int} \\ \mathsf{read\_only}(\tau*) & = & (\mathsf{read\_only}(\tau))^r* \\ \mathsf{read\_only}(\tau[\,]) & = & (\mathsf{read\_only}(\tau))^r[\,] \end{array}$$

The we try to infer permissions in the definition of f so that these parameter types are respected. This is a form of *type inference* and there are a number of techniques

at our disposal. Perhaps the simplest is to introduce *permission variables* for every pointer and array and collect constraints on what they might be. If the constraints are consistent, the program is typable; if not an error is signaled. We might describe this formally in a later lecture; for now we apply it informally.

Let's return to our motivating example.

```
bool store_int(int* p, int n) {
  *p = n;
  return true;
}

int get_length(int[] A) {
  int* p = alloc(int);
  //@assert store_int(p, \length(A));
  return *p;
}
```

We see that `store_int` is called inside an `@assert` contract. We therefore try to check:

```
bool store_int(int@r* p, int n) {
  *p = n;
  return true;
}
```

where we have written `int@r*` for int^r*. This fails, because `*p :@r int` so it is not mutable.

However, in code such as

```
bool is_sorted(int@r[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
{
  for (int i = lower; i < upper-1; i++)
    //@loop_invariant lower <= i;
    if (A[i] > A[i+1]) return false;
  return true;
}
```

we do not mutate any element of A at all, so the permission types are satisfied and `is_sorted` can be called from in contracts.

6 Structs

Structs are a large type, so $\mathsf{struct}\, s$ can occur only in three places: directly as a reference $\mathsf{struct}\, s*$, as an array element $\mathsf{struct}\, s[\,]$, or as the type of a field of a struct.

It would be consistent with the philosophy behind C0, if a large type did not directly have a permission. Instead, we give separate permissions to those fields of a struct that are small, so that they actually appear on the left-hand side of an assignment.

Another option would be to give all fields of a struct the same permission, which is inherited from the pointer or array that references the struct. This is much less differentiated, but it may be sufficient for purity analysis.

As an example, we consider linked lists and functions to return an empty linked list that is read-only, as well as function that creates a new read-only list by prepending a new element x. We view here linked lists as *read-only* to illustrate that we construct them safely in contracts.

```
struct list_node {
  int elem;
  struct list_node@r* next;
};
typedef struct list_node list;

list@r* nil() {
  return NULL;
}

list@r* cons(int x, list@r* l) {
  list@m* head = alloc(list);
  head->elem = x;
  head->next = l;
  return head;
}
```

In the `nil()` function, the type of `NULL` comes up. Even though trying to write to (or read from!) the location 0 will lead to an exception, we treat it here as it if were mutable because we would like to allow `nil()` to be called in contracts (where exceptions are allowed!).

For `cons`, we return a read-only list if the argument is read-only (again, for use in a contract), but we could just as well make it mutable. It would not affect its use in a contract, unless it were passed into a function that required treating it in a read-only fashion. Lists in a functional language are typically immutable in this fashion. Note that for `return head;` to type-check, we have to coerce `list@m* head` to be of type `list@r*`, which is the permitted direction. We just cannot modify the list any more from that point on in the execution of the program.

The recursive nature of structs demands some consistency of permissions, which is enfored by the fact that in the struct definition itself a permission should be assigned to the pointer to the recursive instance.