Aliasing and C--

Norman Ramsey Simon Peyton Jones

July 24, 2000

C compilers must be conservative in their use of memory. For example, unless they implement sophisticated points-to analyses, C compilers must assume that a store through any pointer variable could change the value seen by a later load through another pointer variable. This conservative assumption makes it difficult to optimize code involving references to memory.

- It may be hard to move references out of loops.
- It may be difficult to eliminate common subexpressions involving references to memory.
- The compiler cannot easily change the scheduling of instructions involving loads and stores.

Often, the front end has information that would enable these optimizations.

- Some pointers may refer only to immutable data, in which case writes through other pointers cannot possibly interfere.
- In some languages, references using pointers to values of different types may never interfere.
- Stores to a newly allocated heap object cannot possibly interfere with loads from a fully initialized object, and may not interfere with each other.

The purpose of this note is to propose extensions to C-- that make it possible for front ends to communicate this sort of simple knowledge about the impossibility of aliasing.

We assume that the C-- back end does not do sophisticated alias or pointer analysis; it does simple data-dependence analysis only. Our proposal is designed to fit easily into a simple dataflow framework. It follows that it is up to the front end to perform any sophisticated analysis that might be done. We have *not* studied the literature and verified that our proposal is capable of expressing the results of standard pointer analyses. Our proposal is designed with only two aims in mind:

- The proposal should be easy to implement in a traditional optimizer based on simple use-def analysis.
- The proposed extensions should be capable of expressing the simple facts about memory that are easily deduced from the properties of safe languages and languages with immutable types.

Sets of locations

We divide the machine's locations into disjoint sets, such that a store into one set cannot possibly affect the value of a load from another set.[1] In C--, all register variables are disjoint from each other and from memory locations, so we refer only to memory from here on. C-- knows about only one set of locations: M, which is the set of all memory locations. Front ends may define subsets of M as explained below.

The kernel of our proposal is that every reference to memory carries with it an assertion about the address used in the reference. C-- already contains *alignment* assertions, which have the meaning "address a is a multiple of k," for some fixed k in the assertion. We are proposing to add *aliasing* assertions, which have the meaning "address a is a member of $S_1 \cup S_2 \cup \cdots \cup S_n$," for some fixed sets $S_1, \ldots, S_n$ specified in the assertion. If the C-- compiler sees that two addresses are asserted to belong to disjoint sets, it is free to assume that stores to one address do not interfere with stores to or loads from the other.

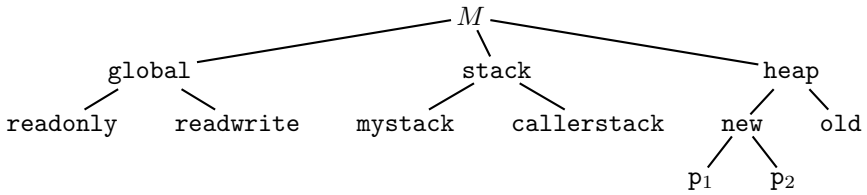
The default aliasing assertion is of course to be $a \in M$.

[1] We take this property to be the *definition* of the term "disjoint." Two unequal addresses may not be disjoint if, for example, an instruction can store a value that is larger than the difference between the addresses.

Specification of sets

How are front ends to specify the sets that addresses belong to, and how shall they specify which sets are disjoint? We propose that each set be represented by a name, and that these names be arranged in a tree with M at the root. The tree enables the compiler to infer that *each node is partitioned by its children.* Because the tree represents the front end's knowledge about aliasing, it must be specified explicitly in the C-- source program. Indeed, each C-- procedure may have its own tree. As discussion below should make clear, the tree structure is not strictly necessary for specification, but it should help front ends build simpler and more readable specifications.

An example tree might be



The names might have these meanings:

M	All of memory.
`global`	Initialized and uninitialized data.
`readonly`	Initialized data that is never written, only read.
`readwrite`	Data that is both read and written.
`stack`	Data allocated on some stack frame.
`mystack`	Data allocated in the current stack frame.
`callerstack`	Data allocated in some frame that is not the current stack frame.
`heap`	A pointer to an object allocated on the heap.
`new`	A pointer to a newly allocated heap object.
$\mathtt{p}_i$	A pointer to newly allocated object $\mathtt{p}_i$
`old`	A pointer to a heap object allocated an initialized before this procedure.

The tree structure indicates that, for example, $\mathtt{p}_1$ and $\mathtt{p}_2$ are disjoint from each other and from any old heap object, that all heap objects are different from stack data, etc.

The tree can be used to identify sets of addresses and to tell the back end which sets are disjoint.

To associate sets with references to memory, the most obvious method is an explicit assertion. For example, a front end might generate the following code to copy a cons cell, assuming the cons cell might be allocated either on the heap or in a caller's stack frame:

⟨*copy a cons cell* c⟩≡
```
copycons(bits32 c) {
  bits32 p;
  p = alloc(12);
  bits32[p   in p1] = bits32[c-4 in old, callerstack];  /* copy header */
  bits32[p+4 in p1] = bits32[c   in old, callerstack];  /* copy car */
  bits32[p+8 in p1] = bits32[c+4 in old, callerstack];  /* copy cdr */
  return p+4;
}
```

The compiler can safely emit "load; load; load; store; store; store," perhaps getting better performance from the memory unit.

It may be tedious or less than readable to write an explicit assertion after every address. We can reduce notation by adding the following rules:

- Every variable is associated with a list of sets. This association is approximately an assertion that whenever the value of the variable is used in an addressing expression, it lies in one of the sets.[2] If no association is specified, the C-- compiler associates the variable with M.
- If a reference to memory does not bear an explicit `in` assertion, its address is asserted to be in the *union* of the sets associated with the free variables of that expression.[3] We've chosen the union because it's a safe, conservative approximation, but it might make more sense to look at the intersection. Intersections would make it easier to handle offsets, array indices, and other things used in address arithmetic, as they could just be put in M.
- If a reference to memory bears an explicit `in` assertion, that assertion overrides all information attached to the free variables of its address.

Under these rules, the notation required to copy a cons cell could be reduced:

⟨*copy a cons cell* c⟩+≡
```
copycons2(bits32 c in old, callerstack) {
  bits32 p in p1;
  p = alloc(12);
  bits32[p  ] = bits32[c-4];  /* copy header */
  bits32[p+4] = bits32[c  ];  /* copy car */
  bits32[p+8] = bits32[c+4];  /* copy cdr */
  return p+4;
}
```

[2] We say "approximately" because dealing with address arithmetic is tricky.
[3] It's not clear what to do when the union is empty, i.e., for absolute addresses.

What the compiler does

Here we describe how the aliasing information can be used in an optimizer. First, the compiler computes an assertion for each reference to memory. Next, looking at the tree of sets, it replaces the names of internal nodes with the names of their children, continuing until each assertion mentions only leaves. At this point it has each assertion is in a kind of disjunctive normal form.

The leaves can now be treated almost like C-- variables, since they don't interfere with C-- variables or with each other. In particular, the compiler can add a suitable **def** and **use** for each leaf, in place of the **def** $\hat{M}$ **use** $\hat{M}$ that appear in Table 3 of our PLDI'00 paper. All the ordinary dataflow stuff just works, and we don't touch anything else in the optimizer.

A refinement

We don't need to stop with loads and stores; the PLDI paper can guide us to a useful refinement. We've replaced the $\hat{M}$s at loads and stores, but we should also do so at call sites. That is, instead of assuming that a call may use or modify any location in memory, we can annotate the call explicitly with the sets of locations it may use or modify. This is well-understood technology; see for example John Guttag's work Larch, which uses similar specifications.

As an example, suppose we have a cons cell allocated on our stack frame, and we get unlucky and must copy it onto the heap.

⟨*call site*⟩≡

```
  .
  :
stackdata {
  x: bits32[3];
}
  .
  :
q = copycons(x) uses mystack modifies new;
  .
  :
```

These kinds of annotations may be most useful for procedures in the front-end run-time system. For example, this would provide a way for someone to say that a procedure to perform bignum adds, for example, uses and modifies only the part of memory pointed to by its arguments.

Could there be any point in annotating the use of global register variables?

It remains to see what, if anything, to do about annotating exit nodes, jumps, etc.

Interpretation

How shall we interpret the names of the sets? Simply, on each execution of a load or store, we want each name to stand for a set of machine addresses. Because we optimize full procedures, we require that there exist a mapping that can be used for the entire activation. Furthermore, this mapping must be such that the (implicit) disjointness specification is sound for that mapping, and the assertions on all the memory references are satisfied.

Where does this leave us wrt interprocedural analysis? Smells like it might be a research question...

Future work

It remains to explore a number of examples and see what kind of knowledge this proposal can express.

It remains to see to what degree the proposed extensions can express the *results* of state-of-the-art pointer analysis.

If we derive address assertions from assertions on variables, we need to think more about how to deal with address arithmetic. For example, in computing the address of an element in an array, how should we treat the index?

We don't know if there are also opportunities to annotate exit nodes, jumps, etc.

It's not clear if or how this idea generalized to interprocedural analysis and optimization.

It would be a good idea to compare this proposal to what's in MLRISC and to Nicolau's proposal.