VimNotes

Notes of the VIM meeting of March 13, 1985
Submitted by Bradley C. Kuszmaul

Present: Dennis, Nikhil, BG, SJ, Kuszmaul, Earl

We decided to let me (Brad) make outrageous statements, so I got up and started making them. Then SJ insisted that I should be the one to take the notes, so now I get to write my outrageous statements down on paper. (Hee hee)

I presented my thoughts as a decision tree:

(I) Suppose we allow generalized letrec. (We did not discuss the alternative in this meeting)

(I.A) Suppose we try to get as much bang as we can by adding only early completion structures (ECS's) and no suspensions, but without committing ourselves as to whether all structures should be ECS's.

(I.A.1) Suppose we assume that not all structures should be ECS's (possibly based on the idea that ECS's are more expensive that regular structures, or possibly based on other reasons).

We have

letrec x = **E** in ... endletrec

Claim I.A.1.a: Every structure built in the dynamic scope of **E** must be an ECS.

Proof of I.A.1.a: x's value may depend an arbitrary access of some part of itself.

For example:

letrex x = cons(1, car(x)) in x endletrec

is perfectly well defined, and is equivalent to

cons(1,1)

and to go even further:

letrec x = cons(cons(1,car(car(x)), car(x)) in x endletrec

which is equivalent to

cons(cons(1,1), cons(1,1))

Claim I.A.1.b: The decision as to whether to construct an ECS or a regular structure must be made dynamically.

Proof of I.A.1.b: Since by assumption **I.A.1** we need to avoid creating too many ECS's. Because user defined functions may create structures, which depending on the dynamic object creation context (i.e. whether the structure is being built for the **E** part of a LETREC or

not) must create ECS's or regular structures, we need a way to decide how to build objects. Since the function may have been passed as an argument, we can not make the decision statically, and thus we need an extra implicit argument telling the function how to create objects.

For example:

```
foo = function (f) returns(something) is
letrec x = f(x)
     in x endletrec
end function foo

...

foo (lambda (x). cons(1, car (x)))
```

Conclusion of I.A.1: I believe that this decision is too expensive. We have to modify all functions, pass extra arguments, and change the instruction set of the base language, just so that we can make LETREC work. That is probably too expensive a choice.

(I.A.2) All structures must be ECS's.

Claim I.A.2.a: That ain't so bad.

Observation: (Handwaving) Most early completion queues (ECQ's) (they really should be called early completion sets, since we do not care about ordering the instructions in the queue, let alone insuring FIFO behavior) never contain more than one element. (Bhaskar claims that most code already meets this criteria, and that the compiler can generate code which trys to do a single select and then distributes the value to the consumers, rather than allowing each consumer perform a select.) Admittedly, not all cases are this simple (e.g. in matrix multiplication, the ECQ's often contain four or five elements).

Implementation Hack: (Reality) However, we can observe that the space needed to keep the ECQ is already present in the instructions which are about to be enqueued. The location where the token will be placed when it is delivered is guaranteed not to be used for anything else until we are ready to deliver that token. Thus the "input box" for the instruction can serve as part of the storage for the ECQ. See the accompanying figure for a mapping from abstract ECS's and ECQ's into our hacked up version.

Note that adding an element to the ECQ is cheap (just munging two pointers, see the enclosed figure), but that we have lost potential concurency when it comes time to deliver the token to the instructions on the ECQ. (It is necessary to get the first instruction into primary memory before the fetch request for the second element can be initiated.)

Hacking up APPEND: We discussed the implementation of APPEND under this scheme, and a mechanism using forward pointers was proposed in lieu of the currently proposed mechanism which uses backward pointers. (The forward pointers live inside the ECQ's of the old array, and when an ASET is done on the old array, the value is passed on to the new array. We could construct another ASET instruction for the new array, or we can hack up ECQ's a little more to allow values to be delivered (ASET'ed) directly to another structure instead of restricting the forward pointers to point to instructions)

Stoy's argument that backward pointers are more efficient because many of the values may never actually get used becomes less strong because we have generalized LETREC. Stoy argued

that the programmer will be creating intermediate arrays in which only a few values will actually be accessed. Bhaskar and I claimed, (and people seemed to believe us) that the LETREC makes this problem go away.

Conclusion I.A.2: People seemed to think that this scheme is a good way to do ECQ's, and that since using this scheme ECS's are no more expensive than normal arrays, that there is no reason not to always use ECS's instead of regular structure objects. (In fact sometimes ECS's are much cheaper, because we can create a large structure with no copying, and without worrying about trying to keep the reference count at 1.

(I.B) We need suspensions sometimes. (Suspensions are claimed to be expensive enough to slow things down if used indiscriminatly.) We discussed conditions under which we can avoid using suspensions in a LETREC. I argued that the suspensions are only needed in code which is lexically inside the **E** part of

LETREC x = **E** in ... endletrec.

And made some claims about how it is not always needed, and that the compiler might be able to make some optimizations. (It is probably not really needed anyway since we have restricted the suspensions to a small amount of code anyway: the code in the **E**.)

Here is a formalization of the handwaving I did during the meeting:

Definition: An expression is *safe* iff
- It is not in the **E** part of a LETREC, or
- It is a compile time constant (e.g. a numeric literal)
- It is a composition of safe expressions with a built in operator or a forall statement, or
- it is a structure value (this can be determined by type checking), or
- it is suspended.

The trick is to decide on a minimal set of expressions to suspend so that every expression and subexpression in the program is safe.

Claim: A safe program has enough suspensions in it.
Proof: Inductively:
base case: We know that expressions which are not involved in letrec are do not need suspensions, and that expressions which are compile time constants do not need suspensions. Furthermore, structure values are early completion structures, so they already behave as if they are suspensions. Anything which is already suspended does not need to be further suspended.
inductive case: The composition of safe expressions with a built in operator or forall does not need to be suspended, because all the "free expressions" in are known to not need suspensions in order to compute their values.

Claim: (unproved) A safe program does not have more suspensions than it needs.

Algorithm: It is trival to see that everything comes out safe. It is also trivial to see that the algorithm is linear in the size of the program.

Algorithm: Start at the leaves of the parse tree for the program, working up, decide if the expression you are looking at is safe, if it is not, then suspend it, and then continue up the parse tree.

EndDecisionTree

Miscellaneous stuff: I pointed out that the design rationale for choosing RELEASE vs. reference count on application records (i.e. instructions) has not been documented. I can't flame about the design unless I know what the reasons for choosing it are.

Next Week: There are cycles (e.g. LETREX X = cons(1,x) in X ENDLETREC): What do we do for

- reference counts
- garbage collection
- debugging
- aborting
- compilers (optimizations?)