CS560 2011-Jan-27 make-up lecture chat log

Sanjay Rajopadhye& Matthew Wiegel

On Thursday Jan 27, there was a problem with the recording and the lecture was not recorded. Sanjay announced and conducted a chat session with online students. This is the transcript of that session, edited and cleaned up with a few figures.

Sanjay: For the lecture, can you try to pull up the scribes' notes and the slides (both are PDF files)?

Sanjay: So the lecture was about how AlphaZ generates the code produced by WriteC.

Sanjay: The slides describe the operational semantics of the Alphabets language.

Sanjay: The scribes' notes are lecture notes taken by a couple of students when I talked about this a couple of years ago.

Sanjay: These notes describe what is called a MEMOIZED implementation, but in some sense that is an optimization (extremely important but not immediately relevant) to what the slides describe. Memoization provides a huge improvement -- so it is essential, but the first part of the slides does not worry about it, but simply gives the "operational semantics" of Alpha.

Sanjay: What a program means and how to compute it. Are you with me so far?

Sanjay: OK, When we write an Alpha program we are specifying a function from the inputs to the outputs, and using the locals as local data (all Alpha variables can be thought of as multidimensional array variables).

Sanjay: The caller (Wrapper) is responsible for allocating the memory for the input and output variables and for passing references to these to the C function generated by WriteC for the Alphabets system.

Sanjay: Can you see that if we have a tool that generates the recursive fib function from the Alphabets fib system, then modifying it so that it produces a memoized C function would be "easy"?

Sanjay: OK.

Sanjay: Let's proceed. What if the Alphabets program is not the simple Fibonacci? What if it is an arbitrary Alphabets program, with complicated, mutually recursive equations, and where the expressions can be arbitrarily complex?

Matthew Weigel: (asked whether we would now see how to modify the Alphabets program to do memoization).

Sanjay: Alphabets is a functional language so you cannot write the memoized function in Alphabets -- you can simply specify that `Fib[i]` is the sum of `Fib[i-1]` and `Fib[i-2]`. It is the job of the code generator -- WriteC -- to produce a memoized function.

Sanjay: What would be the structure of the loop?

Sanjay: Why should it start from 0 and go to n-1?

Sanjay: What if I wrote foo(i) = foo(i+1) + foo (i+2) (with appropriate boundary conditions)?

Sanjay: This program would compute values similar to the fib function (it would compute fib (n-i).

Matthew Weigel: (said something about needing to do dependency analysis)

Sanjay: You are absolutely right.

Sanjay: But what if we cannot do any dependency analysis?

Matthew Weigel: (said something about making the programmer do it)

Sanjay: Actually dependency analysis (scheduling) is known to be a difficult problem (in the general case it's undecidable) so I cannot hope to write a compiler that could take any arbitrary Alphabets program and generate the correct order to evaluate different points. So you are right it has to come from the programmer. But what if the programmer does not know and AlphaZ cannot do a dependence analysis or that the result of the analysis was inconclusive?

Sanjay: So the slides describe the mechanism for constructing the recursive function for each Alphabets program. Remember this scheme has to work for ANY program, in the absence of ANY dependence analysis. It is a fallback strategy for the compiler.

Sanjay: Because our "code-generation" scheme has to work for any Alphabets program, not just the Fibonacci, it has to be a set of rules based on the grammar of the language. In Alphabets, the two main things are equations and expressions.

Sanjay: A program is a set of equations, and each equation is of the form Var = Expression.

Sanjay: So first I have to tell you what the recursive function to evaluate each expression should be, and how to construct this function. Actually I need to explain what the compiler did for (i) equations and (ii) expressions.

Sanjay: Let's first look at expressions.

Sanjay: In Alphabets we only have seven kinds of expressions:

1. atomic expressions -- constant or variable
2. point-wise operator expressions
3. case expressions
4. restrict expressions
5. dependence expressions
6. reduce expressions
7. index expressions

Matthew Weigel: So there are relatively few interactions that the generator has to handle?

Sanjay: Right on! Regardless of the syntax, every expression has a domain (a set of indices where the expression is defined) and we need to define function eval that evaluates an expression at some point *z* in its domain -- this is what slide 7 is saying.

Sanjay: So what does it mean to evaluate a constant at any point in its domain? A function that just returns that constant.

Matthew Weigel: Visiting all the integer points in the polyhedron?

Sanjay: No, visiting integer points in the polyhedron comes later. What does it mean to evaluate an Alphabets expression, *given* a point *z* in its domain? That is what the next few slides are doing.

Sanjay: Now that we know what happens with `constants`, what does it mean to evaluate a `variable` expression, say, X, at some point *z* in its domain?

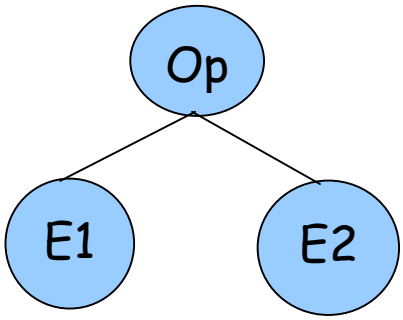
Sanjay: Answer: evaluating the function `evalVar(X)` at *z*.

Matthew Weigel: It means to calculate the function with the indices of *z*? I'm not sure I understand the question.

Sanjay: This may seem a bit confusing, [but look ahead] we will come back to it later when we talk about equations.

Matthew Weigel: OK, sounds good.

Sanjay: Next, what does it mean to evaluate `e1 op e2` at some point *z* in **its** domain? I should use the whiteboard.



Sanjay: That's what the expression tree looks like: `e` is the expression with two children and the root node is `op`.

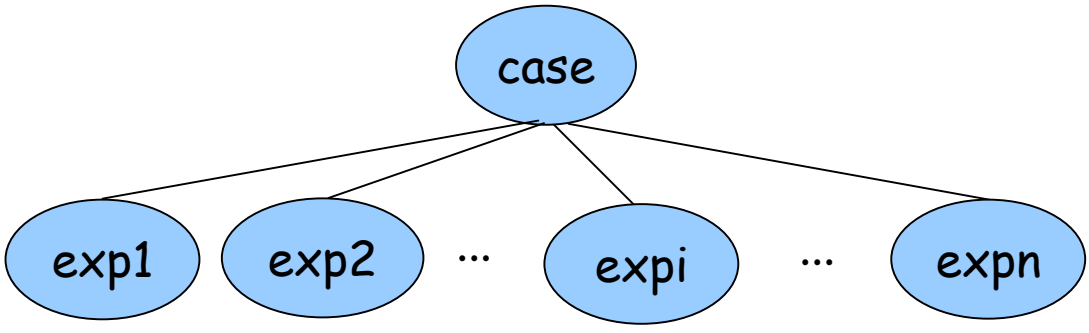
Sanjay: What does it mean to evaluate such an expression at some point *z*?

Matthew Weigel: `e1 op e2` evaluates `e1` at *z*, and `e2` at *z*, and then performs the operation, `op`? It might be something like (in C) `operator(eval(e1, z) eval(e2, z))` is that what you mean?

Sanjay: Absolutely right. So what should be the code that the compiler should generate? The right hand side of the rule in the slides.

Sanjay: The next rule is for `case`.

Sanjay: Again, drawing the syntax tree may be useful. For a `case` the syntax tree is shown above.



Sanjay: In order to evaluate the case expression at *z*, it should first find out which branch is applicable and based on this, call the `eval` function of that particular child with argument *z*. So `Dom(E_i)` is the test to see membership of *z* in `D_i`.

Matthew Weigel: Is it possible for multiple case statements to be evaluated for a single point *z*? For example, for the discussion assignment I had case `{|i == 0}`, `{|j >= i}`, and `{|j < i}`. Two of those are clearly distinct, but `i == 0` can be true when another is.

Sanjay: Great question, but the answer to your question is no. This is because in Alphabets, it is illegal to provide multiple definitions of a variable at any point in its domain.

Matthew Weigel: OK.

Sanjay: In other words, the domains of the different branches of a `case` MUST be disjoint.

Matthew Weigel: Does the first case domain that matches win?

Sanjay: Alphabets is unlike C or other imperative languages, the order for the sub-expressions of the case do not matter. For this, it is essential that their domains be disjoint.

Matthew Weigel: OK, so our program was wrong but it just happened to work. More correct would have been to have the second and third domains specify i > 0?

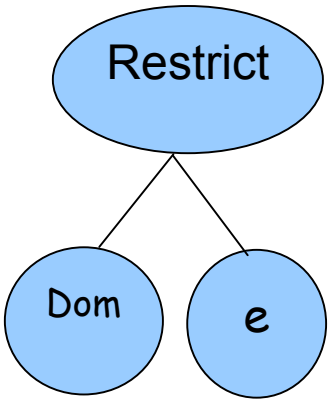
Sanjay: Right. So if domains of different branches are not disjoint it should be flagged as an error. This is being implemented as we chat.

Matthew Weigel: Interesting.

Sanjay: With me so far? Should we move to the next construct?

Matthew Weigel: Yes.

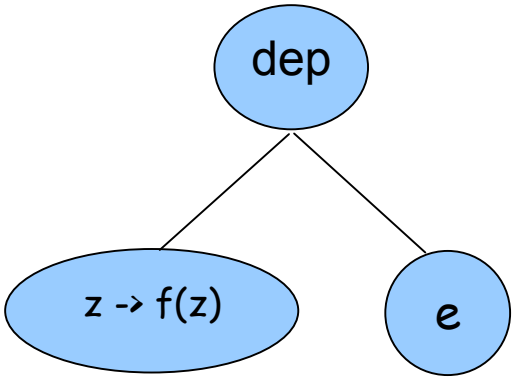
Sanjay: Next, we have a restricted expression, whose syntax tree has just one child sub-expression and a domain as drawn below



Sanjay: Evaluating the restrict expression, `Dom:e` at *z* is the same as evaluating `e` at *z*.

Sanjay: Now you might ask (as did Michelle in class), "Don't we have to test that *z* belongs to Dom?" The answer is no, and it will be clearer in a week or so.

Sanjay: So the four rules on slide 11 should be clear. Moving on the dependence expressions. Now, the syntax tree of a dependence expression is drawn below.



Sanjay: Evaluating this expression at *z* is simply evaluating e at f(*z*). It might seem obvious but remember that f is any dependence function the number of dimensions on the left of the arrow may not be the same as the number of dimensions on the right.

Sanjay: So we are almost done with the Eval' function for expressions. The only kinds of expressions that we have not considered are reductions and index expressions. Index expressions are very simple. They have the syntax just like a dependence expression, but are wrapped in square brackets.

Matthew Weigel: Right.

Sanjay: So evaluating [(z->f(z)] at some point z is simply evaluating the function f at *z* and returning that value.

Sanjay: So all that's left are reductions. For reductions, we need a bit of machinery (actually it's machinery that we deferred earlier). We have a tool that "scans polyhedra," i.e., a tool that takes as input a polyhedron and produces as output a loop that visits all the integer points in the polyhedron. Note that this part of a compiler/code-generator, so the points are not visited when the scanner is executes, but when the generated code is compiled and run.

Matthew Weigel: This is to generate an ordered list of points ... and the generated C has this ordering "baked in"?

Sanjay: Excellent question about the order of the visiting. The order that the scanner uses is lexicographic order. If lexicographic order seems a bit confusing right now, let it go -- for our code generator, the order is NOT relevant. Any order is OK, as long as all the points in the domain are visited.

Sanjay: But you are absolutely right about the "baked in." You will see that this baked in ordering will generate code that produces correct results EVEN if the order of visit is against the order of the dependencies. For a reduce expression too, a separate eval function is generated. To evaluate a reduce expression at any point, *z* in its domain, we call this function with *z* as a parameter. This function has a loop that visits all points in the domain of e that are mapped to *z* by f(*z*) and it accumulates the vale of e (by recursively calling the Eval' function) at those points, and returns the

result.

Sanjay: So we have the mechanism so that the code generator can produce an eval function for any Alphabets expression, regardless of how complicated it is, and how it is nested.

Sanjay: So (slide 17) what should we generate for an equation? For each equation we define a function that is the EvalVar function for the LHS variable. Remember the eval function for a Var expression -- an atomic, non-constant expression? It calls this EvalVar function. OK, so we are (almost) done with what it means to compile an Alphabets program. For each local and output variable (these are the variables for which he system MUST have one and only one equation), we generate these recursive functions, then we again call the polyhedron scanner to generate loops that visit all the integer points in all the output variables (in any order).

Matthew Weigel: And this is where results are memoized, right?

Sanjay: You're right on the memoization, but the slides don't talk about it (yet). In the generated loops for each output variable, we simply call the EvalVar function at that point.

Matthew Weigel: That makes sense.

Sanjay: So slide "Scanning Polyhedra" mentions this scanning problem and basically says that it is a solved problem -- we have the engine to do this. Now go to Slide 21-22**.**

Sanjay: As I described it the generated code above, it will be inefficient (not yet memoized). But this is actually very simple (even for arbitrary Alphabets programs) -- thanks to the fact that domains are polyhedra). We only need to tweak the EvalVar function. At every call to EvalVar(X,z) to evaluate some variable X at some point *z* in its domain, (i) we first read the memo table at address *z*, (ii) if the value has already been computed we return immediately with the value (iii) only if it is not already evaluated, we actually call the actual recursive evaluator, and after that returns, we set the flag to be true before returning. OK?

Matthew Weigel: Right, this is the approach that's described in the scribe's notes.

Sanjay: Yes, but the scribes notes were specific for Fib. We need something that works for any Alphabets program. For this we have a data structure where we store at each index point, a pair <value, flag>. So in the beginning we need to allocate memory for all the local variables in the program.

Matthew Weigel: Oh right, because you can't rely on any particular numeric value to mean "not yet computed."

Sanjay: Right. We also need to allocate flag variables for all the output variables (the storage for the value part of these variables is allocated in the wrapper, not in the function).

Sanjay: Now, with a simple tweak to the flag we can also do one step better. Can you guess?

Matthew Weigel: "not yet evaluated", "evaluated", and "invalidated"?

Sanjay: Right idea: "not yet evaluated", "evaluated" and "*being* evaluated." That third flag is very useful.

Matthew Weigel: Oh right, because it can run in parallel.

Sanjay: No it has nothing to do with parallelism. Provided that when we return from any evaluation we reset that flag to evaluated.

Matthew Weigel: in case there's an infinite recursive loop?

Sanjay: Exactly!! If at any evaluation we encounter that the flag is "being evaluated" we have just detected a cyclic dependence at runtime. So we simply report it to the user. Again in general, detecting cyclic dependences is undecidable -- we cannot expect a compiler to do that automatically. So the generated code has a run-time test.

Matthew Weigel: Understood.

Sanjay: This code is still inefficient on two counts - speed and memory. The memory problem can be very serious (in the original language when we did not have reductions) this would mean that matrix multiplication would require N^3 storage. Currently, we generate code from programs with reductions without forcing the user to serialize, and this is not such an issue. The other drawback is speed.

Matthew Weigel: This is where the scheduling you mention in "Drawbacks and Improvements" comes in?

Sanjay: Right. Since the order of evaluation can be arbitrary and each evaluation requires a call to the eval function, there are too many context switches -- our generated code would run about 8 TIMES slower. So we have the rest of the semester to look at these issues.

Sanjay: How to make the code more efficient through scheduling (and the schedule could easily be a parallel schedule)**.** And it could do different memory allocations.

Matthew Weigel: Right.

Sanjay: Questions?

Matthew Weigel: I don't think I have any right now.

Sanjay: OK. I have to leave in a few minutes.

Sanjay: I think that if you consolidate these notes and send them to me, I'll take a final edit and post.

Matthew Weigel: And Tomofumi, you're already familiar with / actively working on Alphabets and AlphaZ, right?

Sanjay: Yes, Tomofumi is really the AlphaZ guru.

Sanjay: The goal is that by the end of the semester students in the class could/should be contributing to development/extension of AlphaZ.

Sanjay: OK thanks a lot.

Sanjay: I have to leave.

Sanjay: Bye

Matthew Weigel: I will try to add in the diagrams too, but you will definitely need to check those.

Matthew Weigel: OK, bye!