Lecture Notes on Quotation

15-814: Types and Programming Languages
Frank Pfenning

Lecture 18
November 6, 2018

1 Introduction

One of the features that appear to be more prevalent in dynamically than statically typed languages is that of quotation and evaluation. In this lecture we will make sense of quotation type-theoretically and see that as a programming language construct it is closely related to prior work in philosophy on *modal logic* aimed at capturing similar phenomena in logical reasoning.

Our concrete aim and underlying intuition is to model *runtime code generation* [LL98]. This has actually become a staple of many programming language implementations in the guise of *just-in-time* compilation (see, for example, [KWM+08]). Languages such as Java may be compiled to bytecode, which is then interpreted during execution. As long as we have interpreters for various machine architectures, this makes the code portable, but for efficiency reasons we may still want to compile the bytecode down to actual machine code "just in time" (essentially: as it runs). In our model, the programmer (rather than the environment) is in full control over whether and when code is generated, so it differs in this respect from much of the work on just-in-time compilation and is more similar to the quote and eval constructs of languages such as Lisp, Scheme, and Racket.

The approach of using a modal type system [DP01, PD01] has made its way into statically typed languages such as MetaOCaml [Kis14], although some of the technical details differ from what we present in this lecture.

2 A Type for Closed Source Expressions

Early attempts at runtime code generation for functional languages were based on the simple idea that a curried function such as $f : \tau_1 \to (\tau_2 \to \sigma)$ could take an argument of type τ_1 and then generate code for a residual function $f' : \tau_2 \to \sigma$. The problem with this approach was that, often, the program was written in such a way that the best that could be done is to generate a closure. Since generating code at runtime is expensive, in many cases programs would get slower. If we had a closed source expression for $\tau_2 \to \sigma$ we would be sure it no longer depended on the argument v_1 of type τ_1 and we could generate specialized code for this particular v_1.

As a start, let's capture closed expression of type τ in a type $\Box\tau$. The constructor is easy, using $\mathbf{box}\ e$ as the notation for a quoted expression e.

$$\frac{\cdot \vdash e : \tau}{\Gamma \vdash \mathbf{box}\ e : \Box\tau}\ (\text{I-}\Box)$$

No variables that are declared in Γ may appear in e, because we erase it in the premise.

The elimination rule is difficult. Most immediate attempts will be too weak to write interesting programs or are unsound. In the end, there seem to be essentially two approaches [DP01, PD01] that are equivalent in expressive power. We choose the simpler one, introducing a new kind of variable u that stands only for *source expressions*. We generalize our judgment to

$$\Psi\ ;\ \Gamma \vdash e : \tau$$

where Ψ consists of expression variables $u_i : \tau_i$ and Γ consists of value variables $x_j : \tau_j$. Then the rule

$$\frac{\Psi\ ;\ \Gamma \vdash e : \Box\tau \quad \Psi, u : \tau\ ;\ \Gamma \vdash e' : \tau'}{\Psi\ ;\ \Gamma \vdash \mathbf{case}\ e\ \{\mathbf{box}\ u \Rightarrow e'\} : \tau'}\ (\text{E-}\Box)$$

introduces a new expression variable $u : \tau$ with scope e'. The next key insight is that expression variables may appear under **box** constructors, because they will always be bound to source code. We revise our introduction rule:

$$\frac{\Psi\ ;\ \cdot \vdash e : \tau}{\Psi\ ;\ \Gamma \vdash \mathbf{box}\ e : \Box\tau}\ (\text{I-}\Box)$$

In the dynamics, every quoted expression is simply a value.

$$\frac{}{\mathbf{box}\ e\ \mathit{val}}\ (\text{V-}\Box)$$

We have to keep in mind, however, that it is different from lazy evaluation in that e must also be available in source form (at least conceptually, if not in an actual implementation). While an equivalent-looking lazy expression $\langle\!|e, \langle\,\rangle|\!\rangle : \tau \mathbin{\&} 1$ can only be awakened for evaluation by the left projection, a quoted expression that be unwrapped and substituted into another quoted expression.

$$\frac{}{\mathbf{case}\ (\mathbf{box}\ e)\ \{\mathbf{box}\ u \Rightarrow e'\} \mapsto [\![e/u]\!]e'}\ (\text{R-}\Box)$$

We have used a different notation for substitution here to remind ourselves that we are substituting a source expression for an expression variable, which may have a very different implementation than substituting a value for an ordinary value variable.

We also have a standard congruence rule for the elimination construct.

$$\frac{e_0 \mapsto e_0'}{\mathbf{case}\ e_0\ \{\mathbf{box}\ u \Rightarrow e'\} \mapsto \mathbf{case}\ e_0'\ \{\mathbf{box}\ u \Rightarrow e'\}}\ (\text{CE-}\Box)$$

3 An Example: Exponentiation

As an example, we consider exponentiation on natural numbers in unary representation. We allow pattern matching (knowing how it is elaborated into multiple **case** expressions) and assume multiplication can be written in infix notation $e_1 * e_2$.

We define a function *pow* $n\ b = b^n$, with the exponent as the first argument since it is defined recursively over this argument.

$$\begin{array}{lcl} \mathit{pow} & : & \mathit{nat} \to (\mathit{nat} \to \mathit{nat}) \\ \mathit{pow}\ Z\ b & = & S\ Z \\ \mathit{pow}\ (S\ n)\ b & = & b * \mathit{pow}\ n\ b \end{array}$$

We would now like to rewrite this code to another function *exp* such that *exp* n returns *code* to compute b^n. It's type should be

$$\mathit{exp} : \mathit{nat} \to \Box(\mathit{nat} \to \mathit{nat})$$

The case for $n = Z$ is easy:

$$\begin{array}{lcl} \mathit{exp}\ Z & = & \mathbf{box}\ (\lambda b.\ S\ Z) \\ \mathit{exp}\ (S\ n) & = & \ldots \end{array}$$

The case for sucessor, however is tricky. We can **not** write something of the form

$$\begin{array}{lcl} \mathit{exp}\ Z & = & \mathbf{box}\ (\lambda b.\, S\ Z) \\ \mathit{exp}\ (S\, n) & = & \mathbf{box}\ (\lambda b.\ \ldots) \end{array}$$

because the value variable n is not available in the scope of the **box**. Clearly, though, the result will need to depend on n.

Instead, we make a recursive call to obtain the code for a function that computes $\lambda b.\, b^n$.

$$\begin{array}{lcl} \mathit{exp}\ Z & = & \mathbf{box}\ (\lambda b.\, S\ Z) \\ \mathit{exp}\ (S\, n) & = & \mathbf{case}\ (\mathit{exp}\ n)\ \{\mathbf{box}\ u \Rightarrow \underbrace{\qquad}_{:\ \Box(\mathit{nat} \to \mathit{nat})}\ \} \end{array}$$

Because u is an expression variable we can now employ quotation

$$\begin{array}{lcl} \mathit{exp}\ Z & = & \mathbf{box}\ (\lambda b.\, S\ Z) \\ \mathit{exp}\ (S\, n) & = & \mathbf{case}\ (\mathit{exp}\ n)\ \{\mathbf{box}\ u \Rightarrow \mathbf{box}\ (\ \underbrace{\qquad}_{:\ \mathit{nat} \to \mathit{nat}}\ )\} \end{array}$$

Instead of the recursive call $\mathit{exp}\ n$ we use u to construct the code we'd like to return.

$$\begin{array}{lcl} \mathit{exp}\ Z & = & \mathbf{box}\ (\lambda b.\, S\ Z) \\ \mathit{exp}\ (S\, n) & = & \mathbf{case}\ (\mathit{exp}\ n)\ \{\mathbf{box}\ u \Rightarrow \mathbf{box}\ (\lambda b.\, b * (u\, b))\} \end{array}$$

Let's consider this code in action by computing $\mathit{exp}\ (S\ (S\ Z))$. Ideally, we might want something like

$$\mathit{exp}\ (S\ (S\ Z)) \mapsto^* \lambda b.\, b * b$$

but let's compute:

$$\begin{array}{l} \mathit{exp}\ (S\, (S\, Z)) \mapsto^* \mathbf{case}\ (\mathit{exp}\ (S\, Z))\{\mathbf{box}\ u \Rightarrow \mathbf{box}\ (\lambda b.\, b * (u\, b))\} \\ \mathit{exp}\ (S\, Z) \mapsto^* \mathbf{case}\ (\mathit{exp}\ Z)\{\mathbf{box}\ u \Rightarrow \mathbf{box}\ (\lambda b.\, b * (u\, b))\} \\ \mathit{exp}\ Z \mapsto^* \mathbf{box}\ (\lambda b.\, S\ Z) \end{array}$$

Substituting back (including some renaming) and continuing computation:

$$\begin{array}{l} \mathit{exp}\ (S\, Z) \mapsto^* \mathbf{case}\ \mathbf{box}\ (\lambda b_0.\, S\ Z)\{\mathbf{box}\ u \Rightarrow \mathbf{box}\ (\lambda b_1.\, b_1 * (u\, b_1))\} \\ \qquad \mapsto \mathbf{box}\ (\lambda b_1.\, b_1 * ((\lambda b_0.\, S\ Z)\, b_1)) \end{array}$$

And one more back-substitution:

$$\begin{aligned}
\mathit{exp}\ (S\,(S\,Z)) &\mapsto^* \mathbf{case}\ (\mathit{exp}\ (S\,Z))\{\mathbf{box}\ u \Rightarrow \mathbf{box}\ (\lambda b_2.\, b_2 * (u\, b_2))\} \\
&\mapsto^* \mathbf{case}\ \mathbf{box}\ (\lambda b_1.\, b_1 * ((\lambda b_0.\, S\,Z)\, b_1))\ \{\mathbf{box}\ u \Rightarrow \mathbf{box}\ (\lambda b_2.\, b_2 * (u\, b_2)) \\
&\mapsto \mathbf{box}\ (\lambda b_2.\, b_2 * ((\lambda b_1.\, b_1 * ((\lambda b_0.\, S\,Z)\, b_1))\, b_2))
\end{aligned}$$

This is not quite what we had hoped for. But we can perform a simple optimization, substituting variables for variables (noting that $(\lambda x.\, e)\, y \simeq [y/x]e$):

$$\mathit{exp}\ (S\,(S\,Z)) \mapsto^* v \quad \text{with } v \simeq \lambda b_2.\, b_2 * (b_2 * S\,Z)$$

We could eliminate the multiplication by 1 by introducing another case into the definition of the function

$$\begin{array}{lcl}
\mathit{exp}\ Z & = & \mathbf{box}\ (\lambda b.\, S\,Z) \\
\mathit{exp}\ (S\,Z) & = & \mathbf{box}\ (\lambda b.\, b) \\
\mathit{exp}\ (S\,(S\,n)) & = & \mathbf{case}\ (\mathit{exp}\ (S\,n))\ \{\mathbf{box}\ u \Rightarrow \mathbf{box}\ (\lambda b.\, b * (u\, b))\}
\end{array}$$

But the variable for variable reduction is more difficult to eliminate. If we don't want to rely on the smarts of the compiler to perform this kind of inlining, we can further generalize the type $\Box\tau$ to $\Box_\Gamma\,\tau$ by allowing the free variables in Γ to appear in $e : \Box_\Gamma\,\tau$. This is a subject of *contextual modal types* [NPP08].

4 Evaluation

We have now seen an example of how we build a complex quoted expression. But how do we actually run it? For example, how do we compute 5^2 using the staged exponential function? We can define

$$\begin{array}{lcl}
\mathit{exp}' & : & \mathit{nat} \to \mathit{nat} \to \mathit{nat} \\
\mathit{exp}' & = & \lambda n.\, \lambda b.\, \mathbf{case}\ \mathit{exp}\ n\ \{\mathbf{box}\, u \Rightarrow u\, b\}
\end{array}$$

and then $\mathit{pow}\ \ulcorner 2 \urcorner\ \ulcorner 5 \urcorner \mapsto \ulcorner 25 \urcorner$.

We see that the *pow* function computes the quoted expression of type $\Box(\mathit{nat} \to \mathit{nat})$, binds u to the quoted function, and then applies that function. The way this differs from what we wrote in the definition of *exp* is that the expression variable u appears *outside* another **box** constructor. It is such an occurrence that causes the expression to be actually evaluated. In fact, we can define a polymorphic function (with parentheses in the type for emphasis):

$$\begin{array}{lcl}
\mathit{eval} & : & (\Box\alpha) \to \alpha \\
\mathit{eval} & = & \lambda x.\, \mathbf{case}\ x\ \{\mathbf{box}\ u \Rightarrow u\}
\end{array}$$

Critically, we can **not** define a (terminating) polymorphic function

$$\mathit{quote} : \alpha \to (\Box\alpha) \quad \text{(impossible)}$$

Intuitively, that's because we cannot complete

$$\mathit{quote} = \lambda x.\, \mathbf{box}\ ???$$

because underneath the **box** operator we cannot mention the value variable x. We see it is critical that **box** is a language primitive and not a *function*. This mirrors that fact that in modal logic we have an *inference rule* of necessitation

$$\frac{\vdash A}{\vdash \Box A}\ \text{NEC}$$

for an arbitrary proposition A but we cannot prove $\vdash A \supset \Box A$.

What else can we write? We can certainly quote a quoted expression. And we can take a quoted function and a quoted argument and synthesize a quoted application. We express these functions via pattern matching, but it should be clear how to decompose this into individual case expressions.

$$\begin{array}{l}
\mathit{eval} : \Box\alpha \to \alpha \\
\mathit{eval}\ (\mathbf{box}\ u) = u \\
\\
\mathit{quote} : \Box\alpha \to \Box\Box\alpha \\
\mathit{quote}\ (\mathbf{box}\ u) = \mathbf{box}\ (\mathbf{box}\ u) \\
\\
\mathit{apply} : \Box(\alpha \to \beta) \to \Box\alpha \to \Box\beta \\
\mathit{apply}\ (\mathbf{box}\ u)\ (\mathbf{box}\ w) = \mathbf{box}\ (u\, w)
\end{array}$$

If we view these types as axioms in a logic

$$\begin{array}{l}
\vdash \Box A \supset A \\
\vdash \Box A \supset \Box\Box A \\
\vdash \Box(A \supset B) \supset \Box A \supset \Box B
\end{array}$$

then together with the rule of necessitation these are characteristic of the intuitionistic modal logic S4 [PD01]. This is not an accident and we will elaborate further on the connection between logics and type systems in the next lecture.

One limitation: while pattern matching is convenient, we cannot match against the structure of expressions underneath the **box** constructor. Allowing this requires yet another big leap (see, for example, the work on

Beluga [PC15]). Not being able to do this allows us to implement runtime code generation efficiently, because we compile a value **box** e of type $\Box\,\tau$ to a *code generator* for e. Then substitution for expression variables $[\![e/u]\!]e'$ composes code generators, and using an expression variable u outside a **box** will finally call the code generator then jump to the code it produces (see, for example, [WLP98]).

5 Lifting

Not all programs can be restaged in the neat way of the exponentiation function, but there are many examples that work more or less well. Here are some hinted at that can be found in the literature:

$$\mathit{parse} : \mathit{grammar} \to \Box(\mathit{string} \to \mathit{tree}\ \mathbf{option})$$

The staged version of a parser is a *parser generator* which takes a grammar and returns a parsing function from strings to parse trees (when they exist).

$$\mathit{mvmult} : \mathit{matrix} \to \Box(\mathit{vector} \to \mathit{vector})$$

The staged version that multiplies a matrix with a vector returns the source of a function that embodies the matrix values. This is generally a bad idea (the code could be very large) unless we know that the matrix is sparse. For spare matrices, however, it can pay off because we can eliminate multiplication by 0 and potentially get code that approximates the efficiency of specialized code for sparse matrix multiplication.

In general, in these example we sometimes have include *observable values* from one stage into the next stage, for example, integers. We recall from earlier that *purely positive types* have observable values. Ignoring universal and existential types, we have

$$\text{Purely Positive Types}\quad \tau^+ \quad ::= \quad 1 \mid \tau_1^+ \otimes \tau_2^+ \mid 0 \mid \tau_1^+ + \tau_2^+ \mid \rho\alpha^+.\,\tau^+ \mid \alpha^+$$

Also positive, but with a negative type underneath, is $(\Box\tau^-)^+$. For positive types, we can define functions by (nested) pattern matching, but not for negative types (which have the form $\tau_1^+ \to \tau_2^-$ and $\tau_1^- \;\&\; \tau_2^-$). We can also define a family of functions

$$\mathit{lift}_{\tau^+} \quad : \quad \tau^+ \to \Box\tau^+$$

but it would be defined differently for different types τ^+. In other words, the *lift* function would not be parametric! However, when included as a

primitive (justified because it is definable at every positive type) we may be able to rescue some parametricity property. As an example, we consider lifting natural numbers.

$$\begin{array}{lcl} \mathit{lift}_{nat} & : & \mathit{nat} \to \Box\, \mathit{nat} \\ \mathit{lift}_{nat}\ Z & = & \mathbf{box}\ Z \\ \mathit{lift}_{nat}\ (S\, n) & = & \mathbf{case}\ \mathit{lift}_{nat}\ n\ \{\mathbf{box}\ u \Rightarrow \mathbf{box}\ (S\, u)\} \end{array}$$

It is straightforward but tedious to translate this definition into one using only the language primitives directly.

References

[DP01] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, May 2001.

[Kis14] Oleg Kiselyov. The design and implementation of BER MetaOCaml. In M. Codish and E. Sumii, editors, *12th International Symposium on Functional and Logic Programming (FLOPS 2014)*, pages 86–102. Springer LNCS 8475, 2014.

[KWM+08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot client compiler for java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):7:1–7:32, May 2008.

[LL98] Mark Leone and Peter Lee. Dynamic specialization in the Fabius system. *Computing Surveys*, 30(3es), 1998. Published electronically.

[NPP08] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 9(3), 2008.

[PC15] Brigitte Pientka and Andrew Cave. Inductive Beluga: Programming proofs. In A. Felty and A. Middeldorp, editors, *25th International Conference on Automated Deduction (CADE 2015)*, pages 272–281, Berlin, Germany, August 2015. Springer LNCS 9195.

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*,

11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[WLP98] Philip Wickline, Peter Lee, and Frank Pfenning. Run-time code generation and modal-ML. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 224–235, Montreal, Canada, June 1998. ACM Press.