Getting to the Box of it: proof-theory for meta-programming and modal type theory

Murdoch J. Gabbay and Aleksander Nanevski

Thanks to Stephane Lengrand and Roy Dyckhoff

November 18, 2011

Congratulations Roy

We are all delighted to see you retire!

(In a nice way, of course.)

I can only try to imagine how it feels. Good luck, live long, and prosper.

Modal types

Let's look at modal types:

$$A ::= \mathbb{B} \mid \mathbb{N} \mid A \to A \mid \Box A$$

We can interpret $\Box$ in a useful variety of ways: 'the interior of'; 'in the future'; 'necessarily'; 'we can justify that'; and so on.

Aleksander Nanevski had the great idea of interpreting $\Box A$ as 'code for something of type A' or 'an intension of something for type A'.

Let's run through some plausible axioms for this interpretation:

Modal axioms

- $\Box A \to A$. Given code/intension for A, we can compute a value/extension of A. This is known as Axiom T.
- $\Box A \to \Box\Box A$. Given code/intension for A, we can form code that generates code for A by just returning that code. This is known as Axiom 4.
- $\Box(A \to B) \to \Box A \to \Box B$. Given code of a function from A to B, and code for A, we can form code for B. This is known as Axiom K.

Now let's look at a term language:

A term syntax

We fix sets of constants C, atoms a, and unknowns X and define terms by:

$$r ::= C \mid a \mid X_{@} \mid \lambda a{:}A.r \mid rr \mid \Box r \mid \mathit{let}\ X{=}r\ \mathit{in}\ r$$

Constants are just there for things like numbers and addition and stuff.

- $\Box r$ means 'r as code; do not evaluate this'. If $r : A$ then $\Box r : \Box A$.
- $X_{@}$ means 'unbox X and evaluate the code'. If $X : \Box A$ then $X_{@} : A$.

Two levels of variables

Note the two levels of variable (cf. nominal terms).

Why two levels?

- $\lambda X.\Box\Box X_@$ means 'input some code X, unbox it, and box it twice' (type $\Box A \to \Box\Box A$).
- $\lambda a.\Box\Box a_@$ means 'input an argument a, then calculate the syntax "$\Box\Box a_@$"'.

Not the same thing!

We use as to write code and do computation.

We use Xs specifically to make holes in code.

A variable a can have a modal type $\Box A$, and a variable X must have a modal type. We still need the Xs so that we can assemble quoted code.

Jamie, your slides have a mistake

Thank you! There is indeed no λX in the syntax.

$\lambda X.\Box\Box X_{@}$ should read $\lambda a{:}\Box A.\mathit{let}\, X{=}a\ \mathit{in}\, \Box\Box X_{@}$.

Also $\lambda a.\Box\Box a_{@}$ is not syntax because I omitted the type.

Just checking if you're paying attention.

Why is this interesting?

We need to generate and reason on code. Thus code becomes a first-class datum alongside numbers.

The mathematical foundations of our field may be too poor. We could use a foundation in which things like code are taken as primitive.

Code contains names (variable symbols). Nominal techniques help us to handle names in semantics, so it is reasonable to expect nominal techniques to be enriched by applications in meta-programming.

I wanted to understand an existing meta-programming system. Modal type theories appeal to me as a logician. So I looked at the modal type system.

Let's look at some typing rules.

Typing rules

$$\frac{}{\Gamma, a : A \vdash a : A}\ (\textbf{Hyp}) \qquad \frac{}{\Gamma \vdash C : \mathit{type}(C)}\ (\textbf{Const})$$

$$\frac{\Gamma, a{:}A \vdash r : B}{\Gamma \vdash (\lambda a{:}A.r) : A \to B}\ (\to\textbf{I}) \qquad \frac{\Gamma \vdash r' : A \to B \quad \Gamma \vdash r : A}{\Gamma \vdash r'\,r : B}\ (\to\textbf{E})$$

$$\frac{\Gamma \vdash r : A \quad (\mathit{fa}(r){=}\varnothing)}{\Gamma \vdash \Box r : \Box A}\ (\Box\textbf{I}) \qquad \frac{\Gamma, X{:}\Box A \vdash r{:}B \quad \Gamma \vdash s{:}\Box A}{\Gamma \vdash \mathit{let}\ X{=}s\ \mathit{in}\ r : B}\ (\Box\textbf{E})$$

$$\frac{}{\Gamma, X : \Box A \vdash X_{@} : A}\ (\textbf{Ext})$$

Types

This is a simple and clear presentation of the modal type theory which underlies the work of Nanevski, Pientka, Pfenning, and others.

Let's look at it. Things are mostly bog standard. ($\Box\mathbf{I}$) introduces closed code $\Box r$; it can still mention X (because these are used to make holes in code), but it must have no free a. ($\Box\mathbf{E}$) is just a let-construct. (**Ext**) is there to unbox code; so intuitively we have the reduction

$$(\Box r)_@ \to r.$$

Semantics

Here are our types again:

$$A ::= \mathbb{B} \mid \mathbb{N} \mid A \to A \mid \Box A$$

We interpret $\Box$ as 'closed code'.

How are we to give this a denotational semantics?

We can try this:

$$[\![\mathbb{B}]\!] = \{\bot, \top\} \qquad [\![\mathbb{N}]\!] = \{0, 1, 2, \dots\}$$
$$[\![A \to B]\!] = [\![A]\!] \to [\![B]\!] \qquad [\![\Box A]\!] = \{\Box r \mid \varnothing \vdash r : A\}$$

Modal types are interpreted as sets of closed code. Looks good!

Semantics

Look at the semantics of terms which flow from the definition above.

$$r ::= C \mid a \mid X_{@} \mid \lambda a{:}A.r \mid rr \mid \Box r \mid \mathit{let}\, X{=}r\ \mathit{in}\, r$$

Assume valuations ς mapping a to an element of $[\![\mathit{type}(a)]\!]$ and X to an element of $[\![\mathit{type}(X)]\!]$.

$$\begin{aligned}
[\![\top]\!]_\varsigma &= \top \\
[\![\bot]\!]_\varsigma &= \bot \\
[\![a]\!]_\varsigma &= \varsigma(a) \\
[\![\lambda a{:}A.r]\!]_\varsigma &= (x{\in}[\![A]\!] \mapsto [\![r]\!]_{\varsigma[a:=x]}) \\
[\![r'\,r]\!]_\varsigma &= [\![r']\!]_\varsigma\, [\![r]\!]_\varsigma
\end{aligned}$$

Semantics of $\Box r$

We can give $\Box r$ the most wonderful semantics:

$$[\![\Box r]\!]_\varsigma = (\Box r)\varsigma|_u$$

Here $\varsigma|_u$ is a substitution mapping X to $\Box r$ where $\varsigma(X) = \Box r$.

(I.e. we throw away $\varsigma(a)$.)

In order to type, $\Box r$ has no free atoms; so we just take the valuation and apply it to $\Box r$ as if it were a substitution.

Semantics of $X_{@}$

But there is a problem.

Then we need this clause for $X_{@}$:

$$[\![X_{@}]\!]_{\varsigma} = [\![r]\!]_{\varnothing} \quad \text{where} \quad \varsigma(X) = \Box r$$

Do you see? X ranges over boxed code, so the meaning of $X_{@}$ is 'unbox $\varsigma(X)$ and calculate the denotation of that code'.

But we have no control over the complexity of $\varsigma(X)$. The definition above is not inductive. The 'runtime' has infected the 'compile time' of our semantics.

We cannot impose bounds on the complexity of $\varsigma(X)$, since this can be updated by *let* $X{=}r$ *in* s.

Semantics (revised)

Proposed solution:

$$\begin{aligned}
[\![\mathbb{B}]\!] &= \{\top, \bot\} \\
[\![\mathbb{N}]\!] &= \{0, 1, 2, \dots\} \\
[\![A \to B]\!] &= [\![A]\!] \to [\![B]\!] \\
[\![\Box A]\!] &= \{\Box r \mid \varnothing \vdash \Box r : \Box A\} \times [\![A]\!]
\end{aligned}$$

Look: $[\![\Box A]\!]$ is a pair $(\Box r, x)$ of some syntax, and a purported denotation.

Look again: there is no restriction that x is actually equal to a denotation for r. We overgenerate. Just as in $[\![A]\!] \to [\![B]\!]$ there is no restriction that a function be computable or even representable by some term.

Semantics of terms

$$
\begin{aligned}
[\![\top]\!]_\varsigma &= \top \\
[\![\bot]\!]_\varsigma &= \bot \\
[\![a]\!]_\varsigma &= \varsigma(a) \\
[\![\lambda a{:}A.r]\!]_\varsigma &= (x{\in}[\![A]\!] \mapsto [\![r]\!]_{\varsigma[a:=x]}) \\
[\![r'\,r]\!]_\varsigma &= [\![r']\!]_\varsigma\,[\![r]\!]_\varsigma \\
[\![\Box r]\!]_\varsigma &= (\Box(r\varsigma|_u)) :: [\![r]\!]_\varsigma \\
[\![X_@]\!]_\varsigma &= tl(\varsigma(X)) \\
[\![let\ X{=}s\ \ in\ r]\!]_\varsigma &= [\![r]\!]_{\varsigma[X:=[\![s]\!]_\varsigma]} \\
[\![\mathsf{isapp_A}]\!]_\varsigma(\Box(\mathsf{r}'\mathsf{r}'')) &= \top \\
[\![\mathsf{isapp_A}]\!]_\varsigma(\Box(\mathsf{r})) &= \bot \quad (\forall r', r''.r \neq r'\,r'')
\end{aligned}
$$

isapp is just an example constant to show how we can operate intensionally on syntax.

A small theorem

Theorem. There is no injection from $[\![\mathbb{N} \to \mathbb{N}]\!]$ to $[\![\Box(\mathbb{N} \to \mathbb{N})]\!]$ that is injective on the first component.

Proof. The set $[\![\mathbb{N} \to \mathbb{N}]\!]$ is equal to the function-space $\mathbb{N} \to \mathbb{N}$, which is uncountable. The first component of $[\![\Box(\mathbb{N} \to \mathbb{N})]\!]$ is equal to $\{r \mid \varnothing \vdash r : \Box(\mathbb{N}\to\mathbb{N})\}$, which is countable.

Conclusions

See a forthcoming paper by myself and Alex. We also the more powerful (and complicated) contextual modal type system, in which types look like this:

$$A ::= \mathbb{B} \mid \mathbb{N} \mid A \to A \mid [A_1, \ldots, A_n]A$$

There are no additional serious difficulties.

I think that the semantics we have constructed is new, simple, elegant, and interesting in itself.

Conclusions

What could we do with it? Well:

- We bridge the gap between nominal techniques and Nanevski et al.s work. I can finally give a precise answer to the question: "so what does this have to do with CMTT?".
- We can develop new syntaxes. CMTT is no longer a type system which many have called scary; it is henceforth also a semantic idea. I hope this will benefit everybody.
- We can generalise the semantics. Use nominal sets instead of ordinary sets. Build a version of the modal type system in which $\Box A$ admits open code. We would have dynamic binding, as possibly open unboxed code migrates under λ-abstractions.

Conclusions

This links to other work.

The "Nominal Henkin Semantics" and accompanying term language (Gabbay & Mulligan, LFMTP 2011) is, in a sense, a partial generalisation of the syntax and semantics I have just presented for CMTT.

A calculus I am developing with Stéphane Lengrand is a candidate to receive this kind of semantics; an updated version of our "Lambda-context calculus".

We have three bits of a puzzle here. They don't quite fit together yet. But we are getting there.

Along the way, some really nice structures are emerging.