Denotation of contextual modal type theory

Murdoch J. Gabbay (joint work with Aleksandar Nanevski)

Thanks to César Sánchez and Manuel Hermenegildo

23 April 2013

Modal types

Let's combine the modal logic S4 with the simple types of the λ-calculus:

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

We can interpret $\Box$ variously as:

- 'the interior of';
- 'in the future';
- 'necessarily';
- 'we can justify that';

and so on.

Modal axioms

Aleksandar had the great idea of interpreting $\Box A$ as 'closed syntax of type A'.

Let's run through some plausible axioms for this:

Modal axioms

- Axiom T $\quad \Box A \to A$.
  Given syntax for A, we can compute that syntax to get an extension in A.
- Axiom 4 $\quad \Box A \to \Box\Box A$.
  Given syntax s for A, form code that generates code for A by just returning s.
- Axiom K $\quad \Box(A \to B) \to \Box A \to \Box B$.
  Given syntax of a function from A to B, and syntax for A, we can form syntax for B.

So if we can build a simply-typed λ-calculus with combinators of these types, and appropriate reductions, then we have converted modal logic into a meta-programming system.

Modal axioms

Modalities are important in logic, topology, representation theory, philosophy, and implementation.

For instance, LTL (linear temporal logic) is 'just' an extension of S4 and is the basis of the Sugar specification system used by IBM.

Using an S4 modality to analyse the notion of 'code of' makes a lot of sense.

A social interlude

Me and Aleksandar come from quite similar backgrounds. Even so, communication was a significant barrier.

Aleksandar was not sensitive to some important distinctions in my mind, and I was certainly ignorant of many important distinctions in his mind.

Just because we speak the same language does not mean we speak the same language!

I find this minefield fascinating and I want to devote a minute to recalling it. For instance . . .

A social interlude

'Closed syntax of A' does not mean 'code of A', 'values of A', 'computations of A', or 'intension of A'.

- 'Code of A' is often understood as compiled (byte)code, rather than original syntax.
- 'Values of A' can be synonymous with 'normal forms of A' (syntactic notion) or 'denotations of A' (non-syntactic notion).
  Also, mathematicians may silently assume that denotations can always be silently added to syntax as constants!
- 'Computation of A' means . . . whatever you want it to mean.
- 'Intension of A' resembles 'syntax of A', but need not be syntactic and sugests that $\Box\Box A$ be identical to $\Box A$, since taking an intension twice should reveal no further internal structure (cf. partial equivalence relations).

A term syntax

Back to the maths. We have a modal type system. Let's build terms for it.

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.

The terminology atoms/unknowns is taken from nominal techniques, but the idea of two classes of variables exists independently in the CMTT literature.

I think we're going to see a lot more of such languages.

A term syntax

- $\Box r$ 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?

- $\lambda X.\Box X_@$ means 'input syntax X, unbox it, then rebox' (type $\Box A \to \Box A$; a fancy way of writing the identity).
- $\lambda a.\Box a_@$ means 'input a, then throw away this value and output the syntax "$a_@$"'.

Not the same thing!

We use as to compute. We use Xs to move quoted syntax around without evaluating it.

A variable a can have a modal type $\Box A$, but a variable X must have a modal type.

Jamie, your slides have a mistake

Thank you! There is no λX in the syntax.

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

Also $\lambda a.\Box a_{@}$ is not syntax at all; you can't box a free variable (but see the contextual version).

Just checking if you're paying attention.

Computer science vs. maths

It's quite interesting to collect differences between computer scientists (CS) and mathematicians (Mat)—you might prefer other tags but the ideas stand:

- A Mat will assume a canonical notion of equality and an oracle that decides it in constant time. A CS will not assume equality is computable, and will not even assume a single canonical notion of equality.
- A Mat will assume that any set can be arbitrarily partitioned into two halves, and all sets-isomorphic partitions are equal. A CS will not (e.g. partitions might be uncomputable, such as 'the set of numbers representing halting programs', and this is very different from 'the set of prime numbers', itself different from 'the set of even numbers').
- A Mat will assume that if two languages can express the same functions, then they are equivalent. A CS will not (e.g. the translation might cost $O(n!)$, or just fail to be compositional).

Computer science vs. maths

I propose one more such split:

- In maths, the only interesting base datatype is numbers. In CS there is another interesting base datatype: syntax.

Here is another one:

- Mat are typically very interested in symmetry. CS are generally speaking not as interested in this. More on that tomorrow.

Existing programming languages are, frankly, lousy at programming on syntax. I think this is because they are descended from designs by mathematicians circa 1920-1950.

This will have to be fixed. Returning to CMTT, I do not mean to imply that this is 'the answer'. However, by studying it we can better understand what good answers should look like.

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 syntax $\Box r$; 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 syntax'.

How are we to give this a denotational semantics?

We can try this:

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

Modal types are interpreted as sets of closed syntax.

This looks good but it has a serious problem. Let's see why ...

Semantics

Look at the semantics 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 $[\![\mathit{type}(X)]\!]$.

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

$\varsigma|_u$ is a substitution $X \mapsto \Box r$ where $\varsigma(X) = \Box r$. We just take the valuation ς and apply it to $\Box r$ as if it were a substitution.

Semantics of $X_{@}$

The problem is, we need this clause for $X_{@}$:

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

So the meaning of $X_{@}$ is 'unbox $\varsigma(X)$ and calculate the denotation of that code'. We have no control over the complexity of $\varsigma(X)$.

Our definition is not inductive!

$\varsigma(X)$ can be updated by *let* $X{=}r$ *in* s. Therefore, we cannot impose bounds on the complexity of $\varsigma(X)$; we can compute syntax of arbitrary size and just plonk it into $\varsigma(X)$ any time we fancy.

Semantics (revised)

Proposed solution:

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

- $[\![\Box A]\!]$ is a pair $(\Box r, x)$ of syntax + a purported denotation.
- There is no restriction that x is actually equal to a denotation for r.

For instance, $(\Box(1+1), 2) \in [\![\Box\mathbb{N}]\!]$ but also $(\Box(1+1), 7) \in [\![\Box\mathbb{N}]\!]$.

Key idea: we overgenerate. This is fine. 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 my paper with Aleks.

The more powerful (and complicated) contextual modal type system looks like this:

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

From a high level, the semantics of this more complex system is much the same.

Conclusions

The problem is giving a semantics to a modal type $\Box A$ representing 'syntax of type A'. Our solution is simple and elegant. It rests on two ideas:

- Overgeneration.
- Two levels of variable.

Conclusions

The main outstanding problem is to relax box types to permit open syntax, if we can. So $\Box A$ behaves a bit like a coproduct of $[]A$, $[A_1]A$, $[A_1, A_2]A$, ...

This appears to be quite difficult!

Still, I have a number of such calculi under my belt now:

- CMTT semantics.
- The lambda-context calculus.
- Two-level nominal sets.
- (Permissive-)nominal terms and Permissive-nominal logic.
- Hierarchical nominal terms.

These display certain behaviour, and by mixing and matching it may be possible to construct a calculus to a given specification.

Conclusions

A larger problem, which is far harder to address, is that there is no agreed canon of standard problems to address in meta-programming.

Logic has standard results like soundness, completeness, cut-elimination, representation, duality, interpolation, conservative extension.

Metaprogramming is not anywhere as well-organised. It's not even really a single problem; it's more like an amalgamation of various difficulties, some of which arise from the relative insufficiency of ZF foundations.

For this reason, I think meta-programming is considered as a bit of a swamp. I'd be happy to bring some additional element of solid ground to this, even if only a little bit.