CS3110 Fall 2013 Lecture 25: OCaml SL as a Basis for Math and Logic

Robert Constable

1 Lecture Plan

1. Lessons from specifications of computing problems
   - Euclid, Goldbach, Integer Sqr Root
   - $\sqrt{2}$ is irrational, constructing irrationals
   - Halting Problem, Divergence Problem
   - Equality of reals
2. Traditional "foundations of mathematics and CS"
   - Logic and truth
   - Mathematics as a logical system
     - Peano Arithmetic, Higher-Order Arithmetic (HOL)
     - Set Theory (ZFC)
3. Brouwer's intuitionistic foundations
4. The minimal type theory for founding mathematics
   - Examples
   - Relation to OCaml SL
5. Proof Rules (if time, otherwise Lect. 26)

2 Lessons from specifications

Euclid example – what is the lesson?

```
n : nat -> (p : nat where n < p && prime(p))
prime : nat -> bool
```

$\overline{\underline{\text{vs}}}$

$\forall n : \mathbb{N}. \exists p : \mathbb{N}. n < p \;\&\; Prime(p)$
$Prime(p)$

Goldbach example

What is the lesson?

$\sqrt{2}$ is irrational

`r : rat -> (r * r` $\neq$ `2)`
`q : rat ->` q^2 `< 2 |` q^2 `> 2`
how to define $\sqrt{2}$?

Constructing an irrational

```
(r : real * q : rat -> q < r | q > r)
```

Halting Problem Specification in OCaml SL

Let expressions be well formed members of symbols*, all finite strings of symbols.

Let terms be expressions that have types.

We need these types and functions.

`unit_term = (u : term where type_of(u) = unit)`
`eval n t = t'`   evaluating `t` for `n` steps results in term `t'`.

```
halt : (unit_term -> bool) ->
    u : unit_term -> (halt u = true ⇔
                      (n : nat where eval n u = () ))
                       &&
                      (halt u = false ⇔
                      (n : nat -> (m : nat where
                                   eval m u ≠ ()
                                      &&
                                    n < m)))
```

Do we know that any inhabited OCaml SL specification is "true" in mathematics?

> This can be shown in a strong sense as our examples suggest. We'll examine this below.

Do we know that any specification we could write down in mathematics or logic can be expressed as an OCaml SL specification?

> What about this "true" statement in mathematics?
>
> $\forall u : \mathit{term\ where\ type}\ u = \mathit{unit}.$
> $\quad \exists n : \mathbb{N}.\ \mathit{eval}\ n\ u = \texttt{()}\ \mathit{or\ not}\, (\exists n : \mathbb{N}.\ \mathit{eval}\ n\ u = \texttt{()})?$
>
> That is "any expression u of type *unit* either converges to `()` or it diverges."
>
> This can't be expressed in OCaml SL using the standard OCaml SL *or* construct. We need to use a weaker form of *or* defined by Gödel and Kolmogorov. They use $\sim\sim(\alpha \mid\ \sim \alpha)$ for $\alpha \mid \sim \alpha$ where $\sim \alpha$ is defined to be $\alpha \to \mathit{void}$.

3 What is true in mathematics?

3.1 Traditional view

Logic is basic, defines *true propositions*

Math is based on logic

CS is a kind of mathematics

CS is based on logic

Therefore, in CS2800, teach logic first. It can then be reviewed in CS3110 – in about 4 lectures. This is the basis of program correctness in many programming courses. But in OCaml we can do much better, using dependent types as specifications.

3.2 Brouwer's "intuitionistic" view of mathematics

Math is based on typed computation ($\mathbb{N}, \mathbb{R}$ as types)

Logic is a branch of mathematics

Logic is based on typed computation

Therefore, teach typed computation first, as in CS3110 (on $\mathbb{N}$ and ?).

$\mathbb{R}$ was defined using $\mathbb{N}$

What's up?

Leads to deeper look at $\mathbb{R}$ – later in CS4400?

We need Brouwer's real numbers and his key lazy data type of spreads. We now call them *co-trees.*

4 The minimal type theory for founding mathematics

What is the minimal type theory/computation system, needed for mathematics?

$$\begin{array}{cccc}
\texttt{x}:\alpha \texttt{ * } \beta\texttt{(x)} & \texttt{x}:\alpha \texttt{ -> } \beta\texttt{(x)} & \alpha \mid \beta & \text{type} \\
\& & \Rightarrow & \mathit{or} & \\
\exists & \forall & &
\end{array}$$

int $\quad =_{\mathit{int}}$ $\quad \mathit{bool}$ $\quad \mathit{tt}, \mathit{ff}$ as constants

True is *unit* $\quad$ *False* is *void* $\quad \sim \alpha$ is $\alpha \rightarrow \mathit{False}$

That's it!

OCaml SL is enough if we include *type* as a type; to do this carefully Agda, Coq, and Nuprl use a hierarchy of types, $\mathit{type}_1, \mathit{type}_2, \cdots$.

5 Proofs and Proof Rules

Proofs are a "proven way" to find programs in a specification type. Proof rules organize a systematic search for programs and data *given a typing specification*, e.g. a computing task.

The idea of a proof rule in "top down" style.

solve $x : \alpha \rightarrow \beta(x)$

$\vdash x : \alpha \rightarrow \beta(x)$ $\quad$ by $\underline{\ \ ?\ \ }$

The turnstile symbol, $\vdash$, can be read as "solve" or "prove" the task posed by the type.

$$\begin{array}{rll} & \text{solve } x : \alpha \to \beta(x) & \\ & \vdash x : \alpha \to \beta(x) & \text{by } \lambda x \\ x : \alpha & \vdash \beta(x) & \end{array}$$

This suggests a *rule format* for the general case

$$x_1 : \alpha_1, \ldots, x_n : \alpha_n \ \vdash \gamma\,(x_1, \ldots, x_n) \quad \text{by } \underline{\hspace{2cm}}$$

For example

$$\begin{array}{rll} f : (\alpha \to \beta) & \vdash \gamma & \text{by apply } f \\ & \vdash \alpha & \\ f : (\alpha \to \beta),\, y : \beta & \vdash \gamma & \end{array}$$

This example shows that there are two kinds of rule for each type constructor.

Left side rule	Right side rule
"use rule"	"construction rule"
"elimination rule"	"introduction rule"

On the left hand side we typically show the hypotheses on each side of the type we are analyzing, e.g.

$$H,\ f : (\alpha \to \beta),\ H' \ \vdash \gamma$$

where H can be $x_1 : \alpha_1, \ldots, x_n : \alpha_n$ and H' is $x_{n+2} : \alpha_{n+2}, \ldots, x_{n+m} : \alpha_{n+m}$.

More Proof Rules

"and"

$$\begin{tikzcd}
H \ \vdash \alpha * \beta \ \ \text{by pair } (& \_\_ & , & \_\_ &) \\
H \ \vdash \alpha \ \ \text{by } a \arrow[ur, dashed] & & & & \\
H \ \vdash \beta \ \ \text{by } b \arrow[uurrr, dashed] & & & &
\end{tikzcd}$$

$$\begin{array}{ll} H,\, h : \alpha * \beta,\, H' & \vdash \gamma \quad \text{by spread pair } h \\ H,\, x : \alpha,\, y : \beta,\, H' & \vdash \gamma \end{array}$$

“or”

$H \quad \vdash \alpha \mid \beta$ by left	$H \quad \vdash \alpha \mid \beta$ by right
$H \quad \vdash \alpha$	$H \quad \vdash \beta$

$H,\, d : \alpha \mid \beta,\, H' \quad \vdash \gamma$ by decide d

$\quad H,\, x : \alpha,\, H' \quad \vdash \gamma$

$\quad H,\, x : \beta,\, H' \quad \vdash \gamma$