CS 4860

Lecture 3

August 30, 2016

1 Introduction

When comparing the propositional connectives of Boolean propositional logic with types in functional programming laguages, especially for OCaml, Coq, and Nuprl, we have seen similarities:

- function types and implementation, especially $(\alpha \rightarrow \beta)$ and $(A \supset B)$
- product types and conjunction $(\alpha * \beta)$ and $(\alpha \ \& \ \beta)$
- variant types (disjoint unions) and disjunction $(L\alpha \mid R\beta)$ and $(A \vee B)$
- the empty type void and the proposition False
- the unit type and the proposition True

We saw that some laws of propositional (Boolean) logic are not yet "expressible" using types such as $\sim (A \ \& \ B) \Rightarrow \sim A \vee \sim B$. We define $\sim A$ to mean $(A \supset False)$, and we can express $\sim (\alpha * \beta)$ as $(\alpha * \beta) \rightarrow void$. There is a program in $void \rightarrow void$, namely $fun\ x \rightarrow x$. So if we claim there is a program in $\alpha * \beta \rightarrow void$, then in some sense we know that $\alpha * \beta$ must be the empty type (or equivalent to it). That means one of α or β must be empty, possibly both. But we have not yet discussed how to make this idea practical.

2 Type Directed Programming

One of the long term lessons in the history of programming languages is that a type system helps people understand what a program is doing or should accomplish. A good type system also helps prevent coding errors. A popular way of saying this is to claim that "if the program type checks, then it will run correctly." Even better, it will accomplish the specified task. So a mainstream trend in designing programs is to *seek a very expressive type system* with a fast type checking algorithm that runs before evaluation - either in the "top loop" or in the compiler. We are looking at the logic inherent in the type systems of functional programming languages because:

a. it is conceptually rich and logical

b. *the logic can be explained in terms of computation*, and it is more general and expressive than the Boolean logics of the programming language.

c. these "programming logics" turned out to be nearly identical to logics discovered by mathematicians and logicians in the early 1900s and finally "implemented" by computer scientists in the 1980s and 90s. These logics are at the core of modern type theories: Nuprl, Coq, Agda, F*, RedPRL, more to come.

d. Computer scientists predict that the next generation functional style programming languages will use these very rich type systems as the compiler technology improves.

3 Propositional Programming

Proposition	Type	Program (OCaml)
$A \Rightarrow A$	$\alpha \rightarrow \alpha$	$fun\ x \rightarrow x$
$A\ \&\ B \Rightarrow A$	$\alpha * \beta \rightarrow \alpha$	$fun\ x \rightarrow let(a,b) = x\ in\ a$
$A\ \&\ B\ \Rightarrow A \vee B$	$\alpha * \beta \rightarrow (\alpha, \gamma)$union	type ('a,'b) union = L of 'a | R of 'b $fun\ x \rightarrow let(a,b) = x\ in\ L\ a$
$A\ \&\ B\ \Rightarrow B\ \&\ A$	$\alpha * \beta \rightarrow \beta * \alpha$	$fun\ x \rightarrow let(a,b) = x\ in\ (b,a)$
$A \Rightarrow (B \Rightarrow (A\ \&\ B))$	$\alpha \rightarrow \beta \rightarrow (\alpha * \beta)$	$fun\ x \rightarrow fun\ y \rightarrow (x,y)$
$A \Rightarrow A \vee B$	$\alpha \rightarrow (\alpha, \beta)$union	$fun\ x \rightarrow L\ x$
$(A \vee B) \Rightarrow (B \vee A)$	(α, β)union $\rightarrow (\beta, \alpha)$union	
$((A\ \&\ B) \Rightarrow C) \Rightarrow A \Rightarrow (B \Rightarrow C))$	$(\alpha * \beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \beta \rightarrow \gamma$	$fun\ f \rightarrow fun\ x \rightarrow fun\ y \rightarrow f(x,y)$
$False \Rightarrow True$	$void \rightarrow unit$	$fun\ x \rightarrow (\ )$

Note that ( ) is the unique element of unit. We can also see that $True \Rightarrow False$ is impossible to program because there is no element in the type void.

You can type check these programs using Try OCaml online.

3.1 Example 1

We cannot program $\sim(\alpha * \beta) \Rightarrow \sim\alpha \vee \sim\beta$. We prove this in Lecture 4. But if we add the hypothesis $(\sim\alpha \vee \sim\beta)$ we can. Let's see how, first intuitively using logic, then as an OCaml program:

$$(\alpha \vee \sim\alpha) \Rightarrow (\sim(\alpha * \beta) \Rightarrow \sim\alpha \vee \sim\beta)$$

We start by assuming $(\alpha \vee \sim\alpha)$ and proceed by cases.

- In the $\sim\ \alpha$ case, we assume we have a function $na \in (\alpha \rightarrow void)$. So we can return $L\ na$ as the value in $(\sim\alpha \vee \sim\beta)$. Note, OCaml calls this union type a *variant* type.
- In the α case, we can show $\sim\beta$ as follows. If we assume we have $b \in \beta$, then we have the pair (a,b) as input to $f : (\alpha * \beta) \rightarrow void$. Hence $f(a,b) \in void$. Thus we know in the α case that $\sim\beta$ holds.

The above informal argument leads to this OCaml program,

$fun\ da \rightarrow fun\ nab \rightarrow$ match da with $|L\ a \rightarrow R\ fun\ b \rightarrow nab(a,b)$
$|R\ na \rightarrow L\ na$;;

This is the Nuprl program,

$\lambda(da.\lambda(nab.decide(da; a.inr(\lambda(x.nab(a,x)); na.inl(na)))))$.
We will gradually introduce ideas from Nuprl.

4 Truth Tables

Why can't we use the "truth table" method on the OCaml type? We know $\alpha \to \alpha$ since $fun\ x \to x$ is in it. Can we know $\alpha \vee (\alpha \to void)$ in some similar way?

If $\alpha = void$ then we know $void \to void$, it has $fun\ x \to x$ in it. But we need a program that works for *any* α as in $\alpha \to \alpha$. The program will be *polymorphic*, it will not depend on the particular type.

4.1 Example 1

Show $(\alpha \Rightarrow \beta) \Rightarrow (\sim \beta \Rightarrow \sim \alpha)$ using *ex falso quod libet*, $void \to \alpha$ for any α:

$$fun\ f \to fun\ nb \to fun\ a \to any(nb(f(a)))\ .$$

Alternatively we can also write

$$fun\ f \to fun\ g \to fun\ x \to any(g(f(x)))).$$

Claim that adding $any()$ to OCaml as a primitive is OK beacuse we never get an element of void. So the map $any : (void \to \alpha)$ is never used in computation, just in "simulating" logic, when the computation is vacuous but the typing is more "complete."

The above logical rule is called "ex falso quod libet," meaning "from false anything follows." The implementation of $any(x)$ has no computational content, just the typing $any : void \to \alpha$. *A computation can never happen*. We could add a rule based on a defined exception, $any(t) \in exn$. We might discuss this later.

CS exercise: add the *any exception* in a "nice" way.

4.2 Example 2

Now we consider $(A \vee \sim A) \Rightarrow \sim (A\ \&\ B) \Rightarrow\ (\sim A \vee \sim B)$ as a Boolean logic problem. We can just look at the simple truth table:

A	B	$A \vee \sim A$	$\sim (A\ \&\ B)$	$(\sim A \vee \sim B)$	$(A \vee \sim A) \Rightarrow \sim (A\ \&\ B) \Rightarrow (\sim A \vee \sim B)$
t	t	t	f	f	t
t	f	t	t	t	t
f	t	t	t	t	t
f	f	t	t	t	t

Is there a simpler proof method? We can try to force $(A \vee \sim A) \Rightarrow \sim (A\ \&\ B) \Rightarrow (\sim A \vee \sim B)$ to be false. This is false only if $(A \vee \sim A)$ is true (which it always is) and $\sim (A\ \&\ B) \Rightarrow (\sim A \vee \sim B)$ is false:

(1) $\mathbf{F}\ (\sim (A\ \&\ B) \Rightarrow (\sim A \vee \sim B))$
(2) $\mathbf{T}\ (\sim (A\ \&\ B))$ from 1
(3) $\mathbf{F}\ (\sim A \vee \sim B)$ from 1
(4) $\mathbf{F}\ (A\ \&\ B)$ from 2

$\mathbf{F}\ A$ from 4	$\mathbf{F}\ B$ from 4
$\mathbf{F} \sim A$ from 3	$\mathbf{F} \sim A$ from 3
$\mathbf{F} \sim B$ from 3	$\mathbf{F} \sim B$ from 3
$\mathbf{T}\ A$	$\mathbf{T}\ B$
#	#

This can be made to look very much like a proof.