1 Developing a Theory of Polymorphic Programming Logics

We are using other logical theories, such as the Propositional Calculus, to guide our thinking about a precise formal theory of polymorphic programming logics. For instance, are there analogs of the concepts of consistency and completeness? To explore these questions, we need a precise semantics for the polymorphic type “assertions” such as

\[ \sim (\alpha \lor \sim \alpha) \]

\[ \alpha \ast \beta \Rightarrow \alpha \lor \beta. \]

We discussed the following ideas in lecture for developing such a theory. It will depend on a precise account of the syntax of programs and data and the rules of computation. Our programs are functional, so among the basic objects of the theory are functions. We use the Nuprl syntax for these, \( \lambda(x.b(x)) \). We say that functions are canonical values of the theory. This means that they are irreducible by the computation rules. We use functions by applying them. There is an operator for this, \( ap(f;a) \). We read this as “the application of function \( f \) to data element \( a \).” The logic will have this rule for evaluation,

\[ ap(\lambda(x.b(x));a) \text{ reduces to } b(a/x). \]

\[ ap(\lambda(x.b(x));a) \downarrow b(a/x) \]

We sometimes just write \( b(a) \). That is, we substitute the value \( a \) for all occurrences of \( x \) in \( b(x) \).

This method of evaluation is called lazy evaluation because we do not attempt to evaluate the expression \( a \) first. If the result of this evaluation is another application, say \( ap(\lambda(y.e(y));b) \), then we continue evaluating.

Another canonical value of our theory is the pair. We write it as \( pair(a;b) \). We might abbreviate it at times as \( < a,b > \). We need a corresponding operator to decompose a pair and access its components. We use the Nuprl operator

\[ \text{spread}(p;x,y.exp(x,y)). \]
Like \( \text{ap}(f; a) \), \( \text{spread}(p; x, y.\text{exp}(x, y)) \) is a noncanonical term of the logic. There is an evaluation rule for it:

\[
\text{spread}(\text{pair}(a; b); x, y.\text{exp}(x, y)) \text{ evaluates in one step to } \text{exp}(a, b)
\]

(We can also write this with the substitution operator, \( \text{exp}(a/x, b/y) \).)

The canonical values of \( \alpha \lor \beta \) (e.g. \( \alpha | \beta \) in variant notation) are \( \text{inl}(a) \), \( \text{inr}(b) \), where \( a \) is of type \( \alpha \) and \( b \) of type \( \beta \). The term “\text{inl}” stands for “in left,” and “\text{inr}” stands for “in right.”

The corresponding destructor is \( \text{decide}(d; l.\text{left}(l); r.\text{right}(r)) \). This expression is non-canonical. It evaluates as follows,

\[
\text{decide}(\text{inl}(a); l.\text{left}(l); r.\text{right}(r)) \text{ evaluates to } \text{left}(a)
\]

\[
\text{decide}(\text{inr}(b); l.\text{left}(l); r.\text{right}(r)) \text{ evaluates to } \text{right}(b).
\]

The expressions \( \text{left}(a) \) and \( \text{right}(b) \) might evaluate further. The expressions \( \text{left}(l) \) and \( \text{right}(r) \) are just mnemonic not canonical.

### 1.1 Consistency and Completeness

- A good analogue of consistency is the notion that a program in the polymorphic type is correct for any specific types substituted for the type variables, \( \alpha, \beta, \gamma, \ldots \).

- An analogue of completeness is that we can find a program in any polymorphic type which has concrete instances, e.g. for which there are programs when we use concrete types such as \( \text{int}, \text{bool}, \text{char} \), etc. for the type variables.

### 2 Relating Smullyan’s account of First-Order Logic to Refinement Logic for FOL

First-Order Logic (FOL) is about predicates, also called relations, on an arbitrary non-empty domain \( D \) of individual elements. Concrete examples we will study are based on the following domains.

- \( \mathbb{N} \) The natural numbers: 0, 1, 2, \ldots
- \( \mathbb{Z} \) the integers: 0, ±1, ±2, \ldots
- \( \mathbb{Q} \) the rational numbers
- \( \mathbb{R} \) the real numbers
• Sets the collection of all sets
• \( \mathbb{U}_i \) the universes of types \( \mathbb{U}_1, \mathbb{U}_2, \ldots \)

Over the domain of natural numbers \( \mathbb{N} \) we will use these predicates (relations):

• \( eq(x,y) \) a binary relation of equality
• \( add(x,y,z) \) the relation \( x + y = z \)
• \( mult(x,y,z) \) the relation \( x \ast y = z \)
• \( zero(x) \) the predicate “\( x \) is zero”

In FOL we do not have variables for arbitrary relations. These are allowed in Higher-Order Logic, HOL. In some versions of HOL there are also variables for function on the domain \( D \), e.g. \( f : D \rightarrow D, f : D \times D \rightarrow D, \ldots \)

### 2.1 FOL Refinement Rules

1. Exists Construction
   \[
   H \vdash \exists x. B(x) \quad \text{by} \quad \text{pair}(d; \text{slot}_b(d))
   \]
   \[
   \vdash d \in D \quad \text{by} \quad \text{obj}(d)
   \]
   \[
   \vdash B(d) \quad \text{by} \quad \text{slot}_b(d)
   \]

2. Exists Decomposition
   \[
   H, e : \exists y. B(y), \quad H' \vdash G \quad \text{by} \quad \text{spread}(e; d, r. \text{slot}_g(d, r)) \quad \text{new } d, r
   \]
   \[
   H, d : D, r : B(d), \quad H' \vdash G \quad \text{by} \quad \text{slot}_g(d, r)
   \]

3. All Construction
   \[
   H \vdash \forall x. B(x) \quad \text{by} \quad \lambda(x. \text{slot}_b(x)) \quad \text{new } x
   \]
   \[
   H, x : D \vdash B(x) \quad \text{by} \quad \text{slot}_b(x)
   \]

4. All Decomposition
   \[
   H, f : \forall x. B(x), \quad H' \vdash G \quad \text{by} \quad \text{ap}(f; d)
   \]
   \[
   \vdash d \in D \quad \text{obj}(d)
   \]
   \[
   H, \text{`` } \text{``}, H', v : B(d) \vdash G \quad \text{by} \quad \text{slot}_g(v) \quad \text{note, } v = \text{ap}(f; d)
   \]
Here is a proof of an FOL proposition from page 56 of Smullyan.

1. \( \exists x. (P(x) \lor Q(x)) \Rightarrow (\exists x. P(x) \lor \exists x. Q(x)) \)

\[ \vdash \exists x : D. (P(x) \lor Q(x)) \Rightarrow (\exists x : D. P(x) \lor \exists x : D. Q(x)) \text{ by } \lambda (h. \_ \_ \_ ) \]

h: \( \exists x : D. (P(x) \lor Q(x)) \vdash \exists x : D. P(x) \lor \exists x : D. Q(x) \) spread(h; d; or.____)

d: D, or.(P(d) \lor Q(x)) \vdash \exists x : D. P(x) \lor \exists x : D. Q(x) \) decide(or; l.____ ; r.____ )

l: P(d) \vdash \exists x : D. P(x) \lor \exists x : D. Q(x) \) \( \text{inl( } ) \)

l: P(d) \vdash \exists x : D. P(x) < d, l >

r: Q(d) \vdash \exists x : D. P(x) \lor \exists x : D. Q(x) \) \( \text{inr( } ) \)

r: Q(d) \vdash \exists x : D. Q(x) < d, r >

The “program” or proof term or extract is this lambda term

\( \lambda (h. \text{spread}(h; d, \text{or.decide}(or; l.\text{inl}( < d, l >); r.\text{inr}( < d, r >)))) \).

We can translate this proof term or extract into a program in any functional programming language.

**Exercise:** Write the proof term in OCaml of Haskell of Lisp or Java, your favorite programming language with functions as data.

2. \( \vdash \exists x. (P(v) \Rightarrow C) \Rightarrow (\forall x. P(x) \Rightarrow C) \) by \( \lambda (h. \_ \_ \_ ) \)

h: \( \exists x. (P(x) \Rightarrow C) \vdash \forall x P(x) \Rightarrow C \) spread(h; d, i.____)

d: D, i : P(d) \Rightarrow C \vdash \forall x P(x) \Rightarrow C \) \( \lambda (a. \text{ap}(i; \text{ap}(i; \text{ap}(a; d)))) ) \)

a : \( \forall x. P(x) \vdash C \) \( \text{ap}(a; d) = a' \)

\( a' : P(d) \vdash C \) \( \text{ap}(i; a') \quad v = \text{ap}(i; a') \)

\( \vdash P(d) \) by \( a' \)

v : C \vdash C

\( \lambda (h. \text{spread}(h; d, i. \lambda (a. \text{ap}(i; \text{ap}(i; \text{ap}(a; d)))) )) \)