Polymorphic Logic

Mark Bickford & Robert Constable

July 1, 2011

Abstract

A system where the logic is defined in type theory allows us to treat each type constructor as a logical operator. The intersection type can be seen as a new form of universal quantification and the subtype relation as a new form of implication. Members of these types are witnesses for the truth of the corresponding logical propositions. Witnesses of the intersection form of a universal and subtype form of an implication resemble witnesses of the standard universal and implication, but they are polymorphic with respect to their input parameters.

Theorems stated in terms of the usual universal quantifier and implication can sometimes be restated with the corresponding polymorphic versions and given new proofs that construct more uniform, polymorphic, witnesses that are also more efficient.

We illustrate this idea with a proof of a lemma from Smullyan's First Order Logic.[1]

1 Introduction

2 Universal quantification

The standard universal quantifier $\forall x\colon T.\, P(x)$ is defined to be the Π-type, $\Pi x\colon T.\, P(x)$, which we prefer to call the *dependent function type* and write as $x\colon T \to P(x)$. A witness $f \in \forall x\colon T.\, P(x)$ is therefore a function $f \in x\colon T \to P(x)$ that maps any $x \in T$ to a witness for $P(x)$.

In some cases, there may be a single p that is a uniform witness for $P(x)$ for any $x \in T$. In this case, p is a member of the *intersection type*, $\bigcap_{x\colon T} P(x)$. Such a p is not a function with input $x \in T$, but is instead a witness for $P(x)$, *polymorphic* or *uniform* over all $x \in T$.

We define the polymorphic universal quantifier $\forall[x\colon T].\, P(x)$ to be $\bigcap_{x\colon T} P(x)$. The brackets around the bound variable indicate that the witness does not "use" the parameter x. Classically, $\forall x\colon T.\, P(x)$ and $\forall[x\colon T].\, P(x)$ have the same meaning, but constructively they differ. A witness p for the proposition with the polymorphic

[1] This is a draft for the 2011 Oregon Programming Languages Summer School.

quantifier is likely to be more efficient since it does not need to be given an input $x \in T$.

In an extensional, computational type theory, like NuPrl, types are members of a hierarchy of universes, $\mathbb{U}_i$, $i \in \{0, 1, 2, \dots\}$. When the universe level i is unimportant or can be inferred from context, we write *Type* for $\mathbb{U}_i$. Since propositions are defined to be types, we define $\mathbb{P}_i = \mathbb{U}_i$ and write $\mathbb{P}$ when the level is unimportant or can be inferred from context. $\mathbb{P}$ is the type of propositions which we can think of as truth values. A false proposition is an empty type, so it is extensionally equal to *False* = *Void*. A true proposition is a non-empty type and the members of the type are the witnesses for the truth of the proposition.

3 Rules for $\forall[x\colon T].\ P(x)$

The rules for proving $\forall[x\colon T].\ P(x)$ are the rules for proving $\bigcap_{x\colon T} P(x)$. These make use of contexts with *hidden declarations*. To prove $\Gamma \vdash \bigcap_{x\colon T} P(x)$ we must prove $\Gamma, [x : T] \vdash P(x)$. The brackets on the declaration $[x : T]$ added to the context Γ indicate that it is hidden.

To prove this sequent, we use whatever rules are appropriate for proving $P(x)$, and no rules use hidden declarations. The hidden declarations are automatically unhidden once the sequent is refined to one with a conclusion of the form $t_1 = t_2 \in T$. Because the rules for proving an equality proposition all extract a fixed witness term Ax (because we consider equality propositions to have no constructive content) the extract of any proof of $\Gamma, [x : T] \vdash P(x)$ will not include the hidden parameter x.

In particular, the proposition $t \in T$ is simply an abbreviation for $t = t \in T$, so when proving a typing judgement, the hidden declarations are unhidden and may be used.

4 An induction principle

The principle of complete induction over the natural numbers, $\mathbb{N}$, can be written in higher-order logic as

$$\forall P\colon \mathbb{N} \to \mathbb{P}.\ (\forall n\colon \mathbb{N}.\ (\forall m\colon \mathbb{N}_n.\ P(m)) \Rightarrow P(n)) \Rightarrow (\forall n\colon \mathbb{N}.\ P(n))$$

Here, the type $\mathbb{N}_n$ is the set type $\{m\colon \mathbb{N} \mid m < n\}$ whose members are the natural numbers less than n.

A witness for the induction principle is a member *Ind* of the corresponding dependent function type

$$P\colon (\mathbb{N} \to \mathbb{P}) \to (n\colon \mathbb{N} \to (m\colon \mathbb{N}_n \to P(m)) \to P(n)) \to (n\colon \mathbb{N} \to P(n))$$

The witness *Ind* will have the form $\lambda P.\ \lambda G.\ \lambda n.\ \dots$. It takes inputs P, G, and n, where G has type $(n\colon \mathbb{N} \to (m\colon \mathbb{N}_n \to P(m)) \to P(n))$, and produces a witness, *Ind*(P, G, n), for $P(n)$.

If we restate the induction principle using the polymorphic universal quantifier, we get

$$\forall[P\colon \mathbb{N} \to \mathbb{P}].\ (\forall[n\colon \mathbb{N}].\ (\forall[m\colon \mathbb{N}_n].\ P(m)) \Rightarrow P(n)) \Rightarrow (\forall[n\colon \mathbb{N}].\ P(n))$$

Proving this is equivalent to the construction of a witness W of type

$$\bigcap_{P\colon (\mathbb{N}\to\mathbb{P})} (\bigcap_{n\colon \mathbb{N}} (\bigcap_{m\colon \mathbb{N}_n} P(m)) \to P(n)) \to (\bigcap_{n\colon \mathbb{N}} P(n))$$

W will have the form $\lambda F.\ \ldots$ and take an $F \in (\bigcap_{n\colon \mathbb{N}}(\bigcap_{m\colon \mathbb{N}_n} P(m)) \to P(n))$ and produce a member, $W(F)$, of $(\bigcap_{n\colon \mathbb{N}} P(n))$. The input F is a function that takes an $x \in (\bigcap_{m\colon \mathbb{N}_n} P(m))$ and produces a witness, $F(x)$ for $P(n)$. The result $W(F)$ is a uniform witness for all the $P(n)$, $n \in \mathbb{N}$.

Such a W appears to be a fixed point operator, and we can, in fact, prove the polymorphic induction principle using any fixed point combinator **fix** that satisfies

$$\mathbf{fix}(F) \sim F(\mathbf{fix}(F))$$

The relation $\sim$ is the symmetric-transitive closure of $\mapsto$, where $t_1 \mapsto t_2$ if a single primitive computation step such as β-reduction, expanding definitions (δ-reduction), or reducing another primitive ($+$, $*$, ..., on numbers, projections on pairs, etc.) transforms t_1 into t_2. In computational type theory all types are closed under $\sim$, so we have *subject reduction* :

$$x \in T, x \sim y \vdash y \in T$$

Lemma 1.

$$\forall[P\colon \mathbb{N} \to \mathbb{P}].\ (\forall[n\colon \mathbb{N}].\ (\forall[m\colon \mathbb{N}_n].\ P(m)) \Rightarrow P(n)) \Rightarrow (\forall[n\colon \mathbb{N}].\ P(n))$$

Proof. Given $[P \in \mathbb{N} \to \mathbb{P}]$ and $f : \forall[n\colon \mathbb{N}].\ (\forall[m\colon \mathbb{N}_n].\ P(m)) \Rightarrow P(n)$ we must construct a member of $(\forall[n\colon \mathbb{N}].\ P(n))$ (without using P).

We show that $\mathbf{fix}(f)$ (which is independent of P) is in $(\forall[n\colon \mathbb{N}].\ P(n))$. Since this is a proof of a typing judgement, we may now use the declarations that were formerly hidden.

Let Γ be the context $P : \mathbb{N} \to \mathbb{P},\ f : \bigcap_{n\colon \mathbb{N}}(\bigcap_{m\colon \mathbb{N}_n} P(m)) \Rightarrow P(n)$. We must show $\Gamma,\ n : \mathbb{N} \vdash \mathbf{fix}(f) \in P(n)$ and we use the complete induction principle on n. Thus, we show that $\mathbf{fix}(f) \in P(n)$ follows from the assumptions

$$\Gamma, n : \mathbb{N},\ \forall m\colon \mathbb{N}_n.\ \mathbf{fix}(f) \in P(m)$$

But this implies $\mathbf{fix}(f) \in (\bigcap_{m\colon \mathbb{N}_n} P(m))$, and therefore, using the polymorphic type of f, $f(\mathbf{fix}(f)) \in P(n)$. Since $f(\mathbf{fix}(f)) \sim \mathbf{fix}(f)$, we have $\mathbf{fix}(f) \in P(n)$. □

We carried out this proof in NuPrl using for the fixed point combinator the Y-combinator,

$$Y = \lambda f(\lambda x(f(xx)))(\lambda x(f(xx)))$$

The extract of the proof, computed by the system, is simply the term Y.

5 An application: constructing valuations

Raymond Smullyan's "First order logic" [2] begins with a definition of propositional logic. A key concept is the notion of a valuation of a propositional formula given an assignment to its propositional variables, and Smullyan gives constructive proofs of the existence and uniqueness of valuations. We would like to construct the valuations by extraction from the proof of a proposition in our logic, and we want the extracted algorithm to be efficient. We can attain these goals, while remaining faithful to Smullyan's proofs, by expressing an intermediate subgoal of the existence theorem using the polymorphic universal quantifier.

The existence of valuations is expressed in the standard way:

$$\forall x\colon \mathit{form}.\ \forall v_0\colon \mathit{Var}(x) \to \mathbb{B}.\ \exists f\colon \mathit{Sub}(x) \to \mathbb{B}.\ \mathit{valuation}(x, v_0, f)$$

The formal definitions are straightforward. We define a datatype for the formulas of propositional logic by:

$$\mathit{form} := var(Atom) \mid \mathbf{not}\ (\mathit{form}) \mid \mathit{form}\ \mathbf{and}\ \mathit{form} \mid \mathit{form}\ \mathbf{or}\ \mathit{form} \mid \mathit{form}\ \mathbf{implies}\ \mathit{form}$$

This defines the type *form* together with constructors, destructors, and recognizers for each case, and also an induction principle and an induction operator that witnesses the induction principle.

Using the induction operator we define the sub-formula relation $q \subseteq p$ on formulas, and show that it is reflexive and transitive. The types *Var*(x) and *Sub*(x) are then defined using the set type:

$$\begin{array}{rcl} \mathit{Sub}(x) & = & \{v\colon \mathit{form} \mid v \subseteq x\} \\ \mathit{Var}(x) & = & \{v\colon \mathit{form} \mid v \subseteq x \ \wedge\ var?(v)\} \end{array}$$

The induction operator for the type can also be used non-inductively as a simple case operator, and we use this to define the value of a formula p given an assignment v_0 and a function g defined on the proper sub-formulas of p.

$$\begin{array}{rcl} extend(v_0, g, p) & = & case(p) \\ & & var(v) \Rightarrow v_0(v) \\ & & \mathbf{not}\ q \Rightarrow bnot(g(q)) \\ & & q_1\ \mathbf{and}\ q_2 \Rightarrow band(g(q_1), g(q_2)) \\ & & q_1\ \mathbf{or}\ q_2 \Rightarrow bor(g(q_1), g(q_2)) \\ & & q_1\ \mathbf{implies}\ q_2 \Rightarrow bimp(g(q_1), g(q_2)) \end{array}$$

Here, $bnot$, $band$, bor, and $bimp$ are the obvious functions defined on $\mathbb{B}$, the Boolean values.

For a function f of type *Sub*$(x) \to \mathbb{B}$ to be a valuation of x given the assignment v_0 of type *Var*$(x) \to \mathbb{B}$ it must satisfy the constraint *valuation*(x, v_0, f) defined by

$$\mathit{valuation}(x, v_0, f) \Leftrightarrow \forall p\colon \mathit{Sub}(x).\ f(p) = extend(v_0, f, p)$$

This defines a valuation as a function that correctly extends itself.

Lemma 2.

$$\forall x\colon \textit{form}.\ \forall v_0\colon \textit{Var}(x) \to \mathbb{B}.\ \exists f\colon \textit{Sub}(x) \to \mathbb{B}.\ \textit{valuation}(x, v_0, f)$$

Proof. We use the induction operator on formulas to define a rank function $|x|$ with range $\mathbb{N}$ that decreases on proper sub-formulas and assigns variables rank 0. Then we define a bounded valuation by

$$bddval(n, x, v_0, f) \Leftrightarrow \forall p\colon \textit{Sub}(x).\ |x| < n \Rightarrow f(p) = extend(f, v_0, p)$$

Given the context $\Gamma = x : \textit{form}, v_0 : \textit{Var}(x) \to \mathbb{B}$ we must show $\Gamma \vdash \exists f\colon \textit{Sub}(x) \to \mathbb{B}.\ \textit{valuation}(x, v_0, f)$. We use the "cut" rule to assert the (polymorphically quantified)

$$\forall [n\colon \mathbb{N}].\ \exists f\colon \textit{Sub}(x) \to \mathbb{B}.\ \textit{bddval}(n, x, v_0, f)$$

From the assertion we easily complete the proof by choosing n to be $|x| + 1$. To prove the assertion we use the induction principle in Lemma 1. We must then prove that from $n : \mathbb{N}$, $\bigcap_{m\colon \mathbb{N}_n} \exists f\colon \textit{Sub}(x) \to \mathbb{B}.\ \textit{bddval}(m, x, v_0, f)$ it follows that $\exists f\colon \textit{Sub}(x) \to \mathbb{B}.\ \textit{bddval}(n, x, v_0, f)$. For this we let f be a member of the type in the induction hypothesis, and then use $\lambda p.extend(v_0, f, p)$. □

This existence proof is essentially the proof given by Smullyan. We carried out this proof in NuPrl and the extract of the lemma, constructed by the system, is the term

$$\lambda x, v_0.(Y(\lambda f, p.extend(v_0, f, p)))$$

.

These results extend unpublished work [1] attempting to faithfully find the computational content in Smullyan's treatment of Boolean evaluation. The article *Expressing and Implementing the Computational Content Implicit in Smullyan's Account of Boolean Valuations*, is available in pdf under Publications at www.nuprl.org.

References

[1] Stuart Allen, Robert Constable, and Matthew Fluet. Expressing and implementing the computational content implicit in Smullyan's account of Boolean valuations. Draft article, 2003.

[2] R. M. Smullyan. *First–Order Logic*. Springer-Verlag, New York, 1968.