Syntax For Free: Representing Syntax with Binding using Parametricity

Robert Atkey

bob.atkey@ed.ac.uk
School of Informatics, University of Edinburgh

Abstract. We show that, in a parametric model of polymorphism, the type $\forall\alpha.((\alpha \to \alpha) \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha$ is isomorphic to closed de Bruijn terms. That is, the type of closed higher-order abstract syntax terms is isomorphic to a concrete representation. To demonstrate the proof we have constructed a model of parametric polymorphism inside the Coq proof assistant. The proof of the theorem requires parametricity over Kripke relations. We also investigate some variants of this representation.

1 Introduction

Representing, computing with, and reasoning about syntax with binding has been of interest to computer scientists for the last 30 or 40 years. The crucial point that makes these activities difficult is the notion of α-equivalence, the obvious idea that if we have two terms equal up to the swapping of the names of their bound variables, e.g. $\lambda x.x$ and $\lambda y.y$, then the terms should be treated equally. Unfortunately, the obvious representation of binders as a pair of a variable name and a subterm does not respect α-equivalence, so operations on such data must be carefully written in order to respect it.

In this paper, we look at two solutions that have been put forward to deal with this (we do not look at the third major approach: nominal sets [7]): de Bruijn indicies and higher-order abstract syntax, and relate the two.

The de Bruijn index approach [5], approaches the problem by removing the names of bound variables altogether. Bound variables are represented by pointers to the construct that binds them. For instance, the λ-term $\lambda x.\lambda y.xy$ is represented as $\lambda.\lambda.1\ 0$. The bound variable x has been replaced by a pointer to the binder one step away from the occurrence, and the bound variable y has been replaced by a binder zero steps away. The advantage of this representation is that α-equivalent terms are now structurally equal. The disadvantage is the complicated definitions of common operations such as substitution, where non-intuitive shifting operations are required to maintain the correct pointers.

Another common approach is to use *higher-order abstract syntax* [13]. In this approach, we use the binding structure of the meta-language to represent binding in the object-language. For the untyped λ-calculus, we suppose that there is a type tm and operations $\mathsf{lam} : (\mathsf{tm} \to \mathsf{tm}) \to \mathsf{tm}$ and $\mathsf{app} : \mathsf{tm} \to \mathsf{tm} \to \mathsf{tm}$.

The object-level term $\lambda x.\lambda y.xy$ is thus represented as the meta-language term $\mathsf{lam}\ (\lambda x.\ \mathsf{lam}\ (\lambda y.\ \mathsf{app}\ x\ y))$. The key advantage of this approach is that, since object-level variables are represented using meta-level variables, substitution becomes very easy to define. A disadvantage of this representation is the need to make sure that we do not allow too many terms into our type tm. Proving that we have not done so is called *adequacy* [8], and is usually performed by reasoning on the canonical forms of some weak type theory such as LF.

The key to higher-order abstract syntax is that the meta-level variables that are used to represent object-level variables are *only* used as variables, and cannot be further analysed. Washburn and Weirich [18] noted that parametric type abstraction, as available in System F, is a viable way of ensuring that represented terms are well behaved. They consider the type

$$\forall\alpha.((\alpha \to \alpha) \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha$$

and derive a *fold* operator and some reasoning principles from it. This type captures the two operations of higher-order abstract syntax, the *lam* and the *app*, but abstracts over the carrier type. Washburn and Weirich claim that this type represents exactly the terms of the untyped λ-calculus, but do not provide a proof. Coquand and Huet [4] also state that this type represents untyped lambda terms, also without proof. In this paper we provide such a proof.

The reason that this approach works is that System F terms of type $\forall\alpha.\tau$ must act *parametrically* in α, that is, they cannot reflect on what actual instantiation of α they have been provided with. Reynolds [16] formalised this idea by stating that for any two instantiations of α, parametric terms must preserve all relations between them.

We take this idea, and extend it to use Kripke relations [15]. Kripke relations are relations R indexed by some preorder W, such that if $w \leq w'$ in W, then $Rwxy$ implies $Rw'xy$. By requiring that all terms of polymorphic type preserve all Kripke logical relations, we can prove that the denotation of the type given by Washburn and Weirich is isomorphic to the type of closed de Bruijn terms: de Bruijn terms that do not have dangling pointers. The preorder-indexing of the relations is used to handle the expansion of the number of meta-variables being used as object-variables as we go under binders.

Traditionally, parametric models of System F have been hard to come by, and have generally involved fiddly constructions with PERs. We make life easier for ourselves by starting with a meta-theory[1] that already has impredicative polymorphism and construct a parametric model of System F inside it. We use a version of Coq with impredicative polymorphism for this purpose, and we have formalised most of our results here[2].

Overview In the next section we introduce our model of System F inside the Coq type theory. Following that, in Section 3, we present our main result, the

[1] Or meta-meta-theory, if one is pedantic.

[2] The formal development is available from: `http://homepages.inf.ed.ac.uk/ratkey/parametricity`.

isomorphism between the Washburn-Weirich HOAS type and de Bruijn terms. In Section 4 we investigate two alternative representations that take different views on how variables are represented. In Section 5, we show how the computational aspect of System F can be integrated into our object-level representations, and prove that a simplified version of the Haskell ST monad can be represented using de Bruijn-style terms. Finally, Section 6 concludes with a discussion of related work.

2 A model of parametric polymorphism

To state and prove our main results, we construct, inside the Coq proof assistant, a denotational model of System F that supports parametricity. For simplicity, we want to let System F types be denoted by objects of sort Set; we can then express denotations of terms as normal Coq functions that preserve all Kripke relations.

2.1 Preparing the meta-theory

In order to use Sets as denotations of System F types, we require impredicativity. The denotation of the type $\forall\alpha.\tau$ quantifies over all denotations of types (i.e. Sets). By default, Coq's type theory is predicative for Set (although it is impredicative in the type of propositions, Prop), so one cannot construct a new object of sort Set by quantifying over all objects of sort Set. Fortunately, Coq supports a command line option `-impredicative-set` that allows us to proceed.

We also require three axioms to be added to Coq's theory. The first of these is proof irrelevance, which states that all proofs of a given proposition are equal:

$$\forall P : \mathsf{Prop}.\ \forall p_1, p_2 : P.\ p_1 = p_2.$$

We also require extensionality for functions, which states that two functions are equal if they are equal for all inputs:

$$\forall A : \mathsf{Type}, B : A \to \mathsf{Type}, f, g : (\forall a.Ba).\ (\forall x.\ fx = gx) \to f = g$$

Extensionality for functions allows our denotational model to support the η-equality rules of System F. We also require propositional extensionality, which will allow us to treat equivalent propositions as equal:

$$\forall P, Q : \mathsf{Prop},\ (P \leftrightarrow Q) \to P = Q$$

These axioms allow us to define data with embedded proofs that are equal if their computational contents are equal, which will aid us in proving equalities between denotations of System F types.

We informally justify our use of these axioms, plus impredicativity, by the existence of models of CIC in intuitionistic set theory. In the remainder of the paper, we use informal set theoretic notation and do not explicitly highlight the uses of these axioms. Note that everywhere we use the word "set", we are referring to Coq objects of sort Set.

2.2 Denotational semantics of System F

The syntax of System F types is standard:

$$\tau \ ::= \quad \alpha \quad | \quad \tau_1 \to \tau_2 \quad | \quad \forall\alpha.\tau$$

where α is taken from a countably infinite set of variables, and $\forall\alpha.\tau$ binds α in τ. We actually use a de Bruijn representation of types (and terms) of System F in our Coq development, but we will use the usual concrete representation for exposition.

As we mentioned in the introduction, in order to prove the isomorphisms below involving syntax with binding, we require that the denotation of $\forall\alpha.\tau$ be parametric over all Kripke relations over all preorders. Preorders consist of a carrier $W : \mathsf{Type}$ and a binary relation $\leq_W : W \to W \to \mathsf{Prop}$ that is reflexive and transitive. For a given preorder W, a W-Kripke logical relation over sets $A, B : \mathsf{Set}$ is a predicate $R : W \to A \to B \to \mathsf{Prop}$, such that

$$\forall w, w', a, b.\ w \leq_W w' \to Rwab \to Rw'ab.$$

For brevity, we write the collection of all W-Kripke relations over A, B as $\mathrm{KRel}(W, A, B)$. Note that, even though we are using W-indexed Kripke relations, we do not use sets indexed by any particular W as denotations of System F types—we are not constructing a model of System F in the presheaf category for some preorder W. We will require multiple instantiations of W in our proofs.

Type environments γ are mappings from type variables to sets. For a preorder W and a pair of type environments γ_1, γ_2, a relation environment ρ is a mapping from type variables α to W-Kripke relations over $\gamma_1(\alpha), \gamma_2(\alpha)$. For any type environment γ and preorder W, there is a relation environment Δ_γ^W that maps all type variables to the equality relation.

We now define the denotations of types and the induced Kripke relations between them. The mapping $\mathcal{T}[\![-]\!]$ maps types with type environments to sets and the mapping $\mathcal{R}[\![-]\!]$ maps types τ, preorders W and relation environments over type environments γ_1, γ_2 to W-Kripke relations over $\mathcal{T}[\![\tau]\!]\gamma_1$, $\mathcal{T}[\![\tau]\!]\gamma_2$. These mappings are mutually defined over the structure of types:

$$\begin{aligned}
\mathcal{T}[\![\alpha]\!]\gamma &= \gamma(\alpha) \\
\mathcal{T}[\![\tau_1 \to \tau_2]\!]\gamma &= \mathcal{T}[\![\tau_1]\!]\gamma \to \mathcal{T}[\![\tau_2]\!]\gamma \\
\mathcal{T}[\![\forall\alpha.\tau]\!]\gamma &= \{\ x : \forall A : \mathsf{Set}.\ \mathcal{T}[\![\tau]\!](\gamma[\alpha \mapsto A]) \\
&\qquad\quad |\ \forall W, A_1, A_2, R : \mathrm{KRel}(W, A_1, A_2), w : W. \\
&\qquad\qquad \mathcal{R}[\![\tau]\!]^W(\Delta_\gamma^W[\alpha \mapsto R])\ w\ (x\ A_1)\ (x\ A_2)\ \} \\
\mathcal{R}[\![\alpha]\!]^W \rho\ w\ x\ y &= \rho(\alpha)\ w\ x\ y \\
\mathcal{R}[\![\tau_1 \to \tau_2]\!]^W \rho\ w\ f\ g &= \forall w' : W, x : \mathcal{T}[\![\tau_1]\!]\gamma_1, y : \mathcal{T}[\![\tau_1]\!]\gamma_1.\ w \leq_W w' \to \\
&\qquad\qquad \mathcal{R}[\![\tau_1]\!]^W \rho\ w'\ x\ y \to \mathcal{R}[\![\tau_2]\!]^W \rho\ w'\ (fx)\ (gy) \\
\mathcal{R}[\![\forall\alpha.\tau]\!]^W \rho\ w\ x\ y &= \forall A_1, A_2, R : \mathrm{KRel}(W, A_1, A_2). \\
&\qquad\qquad \mathcal{R}[\![\tau]\!]^W(\rho[\alpha \mapsto R])\ w\ (x\ A_1)\ (y\ A_2)
\end{aligned}$$

These clauses are mostly straightforward for Kripke logical relations, but we draw the reader's attention to the clause for $\mathcal{T}[\![\forall\alpha.\tau]\!]$. We have used impredicative quantification over all sets here. We also constrain the denotations of polymorphic types to be those that preserve all W-Kripke relations, for all preorders W. It is this parametricity property that we will use to prove the isomorphisms in Section 3.

Lemma 1. *The following hold, for all τ and preorders W:*

1. *For all γ_1, γ_2 and ρ, $\mathcal{R}[\![\tau]\!]^W\rho$ is a W-Kripke relation over $\mathcal{T}[\![\tau]\!]\gamma_1, \mathcal{T}[\![\tau]\!]\gamma_2$.*
2. *For all γ and w, $\mathcal{R}[\![\tau]\!]^W \Delta_\gamma^W\ w\ x\ y$ iff $x = y$.*

Proof. Both by induction over the structure of τ.

Note that this denotational semantics of types validates the usual representations of inductive types in System F, e.g. $\mathcal{T}[\![\forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha]\!]\gamma \cong \mathbb{N}$ etc.

Denotations of System F terms We also define a denotation for every well-typed System F term, but we have elided these for lack of space. Please see the formal development for more details. The main result is that every well-typed System F term has a meaning in the model as a function from the denotation of the context to the denotation of the result type, such that all Kripke relations over any preorder are preserved by this function.

3 Representing λ-terms using parametricity

We will show that, in our model, the denotation of the type

$$\tau_H = \forall\alpha.((\alpha \to \alpha) \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha$$

is isomorphic to the set of closed de Bruijn terms. This task is not so straightforward as producing two functions and showing that they are mutually inverse: we must show that the function from the above type to de Bruijn terms actually does give a well-formed closed de Bruijn term.

We define the set of well-formed de Bruijn terms as a natural number-indexed inductively defined set $\mathsf{Term} : \mathbb{N} \to \mathsf{Set}$ with constructors:

$$\begin{aligned}
\mathsf{Var} &: \{i : \mathbb{N} \mid i < n\} \to \mathsf{Term}(n) \\
\mathsf{Lam} &: \mathsf{Term}(n+1) \to \mathsf{Term}(n) \\
\mathsf{App} &: \mathsf{Term}(n) \to \mathsf{Term}(n) \to \mathsf{Term}(n)
\end{aligned}$$

The set of all closed de Bruijn terms is hence given by $\mathsf{Term}(0)$. This definition admits the following recursion principle[3]:

$$\begin{aligned}
\mathsf{term\_rec} : {} & \forall P : \mathbb{N} \to \mathsf{Set}. \\
& \quad (\forall n.\{i : \mathbb{N} \mid i < n\} \to P(n)) \to \\
& \quad (\forall n.P(n+1) \to P(n)) \to \\
& \quad (\forall n.P(n) \to P(n) \to P(n)) \to \\
& \quad \forall n.\mathsf{Term}(n) \to P(n)
\end{aligned}$$

[3] This is less general than the one Coq provides, but suffices for our purposes.

We will also need the set of "pre-de Bruijn" terms—terms that are not necessarily known to be well-formed—as an intermediate staging ground. The set $\mathsf{preTerm}$ is defined inductively with the following constructors:

$$\begin{aligned}
\mathsf{preVar} &: \mathbb{N} \to \mathsf{preTerm} \\
\mathsf{preLam} &: \mathsf{preTerm} \to \mathsf{preTerm} \\
\mathsf{preApp} &: \mathsf{preTerm} \to \mathsf{preTerm} \to \mathsf{preTerm}
\end{aligned}$$

There is an obvious relation $n \vdash t$ relating context sizes to $\mathsf{preTerm}$s well-formed in that context, and an isomorphism between $\mathsf{Term}(n)$ and $\{t : \mathsf{preTerm} \mid n \vdash t\}$.

Note that the type $\mathsf{preTerm}$ is a normal inductive type and is therefore representable in parametric System F. The mapping from τ_H to $\mathsf{preTerm}$ that we give is also expressible in pure System F.

We are now ready to define this mapping from denotations of the type τ_H to $\mathsf{Term}(0)$. We do this first by mapping to $\mathsf{preTerm}$ and then showing that the produced term satisfies $0 \vdash t$. By the definition of $\mathcal{T}[\![\tau_H]\!]$, the underlying set for this type is $\forall A : \mathsf{Set}.((A \to A) \to A) \to (A \to A \to A) \to A$. We define $\phi(t) = t\ (\mathbb{N} \to \mathsf{preTerm})\ \mathit{lam}\ \mathit{app}\ 0$, where:

$$\begin{aligned}
\mathit{lam} &= \lambda f.\lambda i.\mathsf{preLam}\ (f\ (\lambda j.\mathsf{preVar}\ (j - (i+1)))\ (i+1)) \\
\mathit{app} &= \lambda x.\lambda y.\lambda i.\mathsf{preApp}\ (x\ i)\ (y\ i)
\end{aligned}$$

We instantiate a value of type τ_H with the set $\mathbb{N} \to \mathsf{preTerm}$, intending that applying a function of this type to a number n will produce a term well-formed in the context of size n. Inside the definition of these functions, the argument represents the depth of context (or the number of binders) surrounding the current term. In the case for *app*, we do not go under a binder, so we do not increase the depth when applying it to the sub-terms. In the case for *lam*, given a function f of type $(\mathbb{N} \to \mathsf{preTerm}) \to (\mathbb{N} \to \mathsf{preTerm})$, and a depth i, we apply f to an argument that will evaluate to a bound variable for a future depth j. The arithmetic computes the distance between the bound variable and its binder. Crucially, it is always the case that $j > i$, since we only ever count upwards in the depth of terms. This is the meat of the following:

Lemma 2. *For all* $t : \mathcal{T}[\![\tau_H]\!]\gamma$, $0 \vdash \phi(t)$.

Proof. We use the parametricity of the denotation of τ_H. Unfolding the definition of $\mathcal{R}[\![\tau_H]\!]$, this tells us that the following property holds of all $t : \mathcal{T}[\![\tau_H]\!]\gamma$:

$$\begin{array}{l}
\forall W, A_1, A_2, R : \mathrm{KRel}(W, A_1, A_2), w : W. \\
\quad (\forall w_1 \geq w, \mathit{lam}_1 : (A_1 \to A_1) \to A_1, \mathit{lam}_2 : (A_2 \to A_2) \to A_2. \\
\qquad (\forall w_2 \geq w_1, f_1 : A_1 \to A_1, f_2 : A_2 \to A_2. \\
\qquad\quad (\forall w_3 \geq w_2, x : A_1, y : A_2. R\ w_3\ x\ y \to R\ w_3\ (f_1\ x)\ (f_2\ y)) \to \\
\qquad\quad R\ w_2\ (\mathit{lam}_1\ f_1)\ (\mathit{lam}_2\ f_2)) \to \\
\qquad (\forall w_4 \geq w_1, \mathit{app}_1 : A_1 \to A_1 \to A_1, \mathit{app}_2 : A_2 \to A_2 \to A_2. \\
\qquad\quad (\forall w_5 \geq w_4, x_1 : A_1, x_2 : A_2. R\ w_5\ x_1\ x_2 \to \\
\qquad\qquad (\forall w_6 \geq w_5, y_1 : A_1, y_2 : A_2. R\ w_6\ y_1\ y_2 \to \\
\qquad\qquad\quad R\ w_6\ (\mathit{app}_1\ x_1\ y_1)\ (\mathit{app}_2\ x_2\ y_2))) \to \\
\qquad\quad R\ w_4\ (t\ A_1\ \mathit{lam}_1\ \mathit{app}_1)\ (t\ A_2\ \mathit{lam}_2\ \mathit{app}_2)))
\end{array}$$

We let W be $\mathbb{N}$ with the usual ordering. We will not need to use both type arguments for this proof, so we set $A_1 = \mathbb{N} \to \mathsf{preTerm}$ and $A_2 = 1$, the one element set (we use dummy implementations of *lam* and *app* for this type). We set $R\,n\,x\,y$ iff $\forall n' \geq n. n' \vdash x(n')$. It is easy to verify that this is a Kripke relation. This relation will suffice to prove our lemma, provided we can prove that our implementations of *lam* and *app* in the definition of ϕ satisfy the requirements of t's parametricity property.

For *lam*, we must prove that at all depths $n \geq 0$, if we are given a functional argument $f : (\mathbb{N} \to \mathsf{preTerm}) \to (\mathbb{N} \to \mathsf{preTerm})$ satisfying the property at all $n' \geq n$, then for all $n'' \geq n$, we have

$$n'' \vdash \mathsf{preLam}\ (f(\lambda j.\mathsf{preVar}\ (j - (n''+1)))\ (n''+1))$$

This is true if

$$n''+1 \vdash f(\lambda j.\mathsf{preVar}\ (j - (n''+1)))\ (n''+1)$$

Since f preserves R, we need only show that the argument $\lambda j.\mathsf{preVar}\ (j - (n''+1))$ satisfies R at all $n''' \geq n''+1$. This amounts to showing that

$$n''' \vdash \mathsf{preVar}(n''' - (n''+1))$$

which is trivial.

The case for *app* is easier and is a straightforward application of the required property being satisfied by the two arguments.

This proof is very similar to the Kripke logical relations proof employed by Rhiger [17] to prove that a single language embedded using higher-order abstract syntax always gives well-formed terms. We have extended this by allowing multiple languages to be embedded in a single meta-language. Rhiger also considers the use of type constructors to embed typed languages, something we cannot do in our System F setting. We also note that the proofs here are very similar in structure to the proofs used for proving adequacy of higher-order syntax encodings in LF [8].

Corollary 1. *The map ϕ can be seen as a map from $\mathcal{T}[\![\tau_H]\!]\gamma$ to* $\mathsf{Term}(0)$.

The map ϕ^{-1} from closed de Bruijn terms is defined by recursion over the structure of terms. We make use of an auxiliary data structure of vectors $\mathsf{vec}\ A\ n$, representing lists of elements of type $A : \mathsf{Set}$ of length n. These have two constructors:

$$\begin{aligned} \mathsf{vecNil} &: \mathsf{vec}\ A\ 0 \\ \mathsf{vecCons} &: A \to \mathsf{vec}\ A\ n \to \mathsf{vec}\ A\ (n+1) \end{aligned}$$

and a look-up function $\mathsf{lookup} : \mathsf{vec}\ A\ n \to \{i : \mathbb{N} \mid i < n\} \to A$.

The mapping $\phi^{-1} : \mathsf{Term}(0) \to \mathcal{T}[\![\tau_H]\!]\gamma$ is defined as:

$$\begin{aligned}\phi^{-1}(t) = \lambda A : \mathsf{Set}.\lambda lam.\lambda app.\ &\mathsf{term\_rec}\ (\lambda n.\mathsf{vec}\ A\ n \to A)\\ &(\lambda n, i, env.\ \mathsf{lookup}\ env\ i)\\ &(\lambda n, h, env.\ lam\ (\lambda x.\ h\ (\mathsf{vecCons}\ x\ env)))\\ &(\lambda n, x, y, env.\ app\ (x\ env)\ (y\ env))\\ &0\ t\ \mathsf{vecNil}\end{aligned}$$

The basic idea is to recurse down the term, maintaining a vector of representations of bound variables. Every time we go under a binder, we extend the vector by the object provided by the implementation of *lam*. For this mapping to be well-defined, we must prove the following:

Lemma 3. *For all $t : \mathsf{Term}(0)$, $\phi^{-1}(t)$ is parametric.*

Proof. We must prove, essentially, that for any preorder W, pair of sets A_1, A_2 and W-Kripke relation R over A_1, A_2, then if lam_1, lam_2 and app_1, app_2 are related pairs of functions, then the bodies of ϕ^{-1} are related by R at some index w. We strengthen the statement from talking about terms in $\mathsf{Term}(0)$ with empty starting environments to: for all n and $t : \mathsf{Term}(n)$, $v_1 : \mathsf{vec}\ A_1\ n$, $v_2 : \mathsf{vec}\ A_2\ n$ and $w' \geq w$,

$$\forall i : \{i : \mathbb{N} \mid i < n\}, w'' \geq w'.\ R\ w''\ (\mathsf{lookup}\ v_1\ i)\ (\mathsf{lookup}\ v_2\ i)$$

implies $R\ w'\ (\mathsf{term\_rec}\ ...\ t\ v_1)\ (\mathsf{term\_rec}\ ...\ t\ v_2)$. This is easily proved by induction on t, and implies the lemma statement.

We now prove that our two mappings are mutually inverse. We first do the direction that does not require parametricity:

Lemma 4. *For all $t : \mathsf{Term}(0)$, $\phi(\phi^{-1}(t)) = t$.*

Proof. As with the previous proof, we strengthen the statement to prove that for all n, $t : \mathsf{Term}(n)$ and $v : \mathsf{vec}\ (\mathbb{N} \to \mathsf{preTerm})\ n$,

$$\forall i \leq n, n'.\ n \leq n' \to (\mathsf{lookup}\ v\ i)\ n' = \mathsf{Var}(i + (n - n'))$$

implies $\mathsf{term\_rec}\ ...\ t\ v\ n = t$. This is easily proved by induction on t, and implies the lemma statement.

The other direction requires the use of parametricity:

Lemma 5. *For all $t : \mathcal{T}[\![\tau_H]\!]$, $\phi^{-1}(\phi(t) = t$.*

Proof. We are given a set A and operations *lam* and *app*. We apply the parametricity property of t (as given in the proof of Lemma 2) with the following data. The preorder W consists of lists of elements of A with the prefix ordering. The set A_1 is set to $\mathbb{N} \to \mathsf{preTerm}$, and A_2 is set to A. We set the relation R to be $R\ env\ x\ y$ iff:

$$\forall env' \sqsupseteq env.\ \mathsf{term\_rec}\ ...\ (x\ (\mathsf{length}\ env'))\ (\mathsf{toVec}\ env') = y$$

where length gives the length of a list, and toVec maps lists l of As to a value of type $\mathsf{vec}\ A\ (\mathsf{length}\ l)$. It is easy to prove that is is a Kripke relation. The proof then proceeds in a very similar way to the proof of Lemma 2.

Summing up, we have:

Theorem 1. $\mathsf{Term}(0) \cong \mathcal{T}[\![\tau_H]\!]\gamma$.

4 Alternative representations of variables

Washburn and Weirich [18] also consider terms with a fixed maximum number of free variables by using types of the form:

$$\tau_H^n = \forall\alpha.((\alpha \to \alpha) \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha^n$$

where $\alpha^0 = \alpha$ and $\alpha^{n+1} = \alpha \to \alpha^n$. By extending the proof in the previous section, we have been able to prove $\mathcal{T}[\![\tau_H^n]\!]\gamma \cong \mathsf{Term}(n)$ for various n, but unfortunately we have not been able to formally prove this for all n.

Washburn and Weirich further claim ([18], in the definition of `iterList`) that the type

$$\forall\alpha.((\alpha \to \alpha) \to \alpha) \to (\alpha \to \alpha \to \alpha) \to [\alpha] \to \alpha$$

represents terms with arbitrary numbers of free variables, where $[\alpha]$ is shorthand for lists of α. However, it is easy to see that this is not the case. Consider the following inhabitant of this type:

$$\Lambda\alpha.\lambda lam.\lambda app.\lambda env.\ \mathsf{match}\ env\ \mathsf{with}\ \mathsf{nil}\ \Rightarrow lam(\lambda x.x)\ |\ \mathsf{cons}(x,t) \Rightarrow x$$

(where we allow ourselves some syntactic sugar for lists in System F). This "term" represents $\lambda x.x$ when the free variable list is empty, and the first available free variable otherwise. This does not correspond to any single λ-term.

We now look at two other representations of variables in higher-order abstract syntax and evaluate them in the light of the techniques of Section 3.

4.1 Parameterised and weak higher-order abstract syntax

In [6] the authors note that the normal higher-order abstract syntax type cannot be directly translated to an inductive type in Coq due to the negative occurrence in the case for λ-abstraction. They propose *weak* higher-order abstract syntax, defined by an inductive type parameterised by a type of variables. We can represent this type in System F like so, using the normal encoding of inductive types:

$$\tau_{WH}(\nu) = \forall\alpha.(\nu \to \alpha) \to ((\nu \to \alpha) \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha$$

Choosing something obvious for ν, like natural numbers, results in inhabitants of this type that do not represent λ-terms (because they can inspect the variable names they are given). The solution is to keep the type ν abstract, so that inhabitants cannot inspect their variables. Hofmann [9] analysed this construction in the setting of presheaves, using a presheaf of variables for ν.

Following on from [6], Chlipala [3] noticed that, if the meta language has parametric polymorphism, then the type $\forall\nu.\tau_{WH}(\nu)$ can be used to represent λ-terms, but he did not have a proof. He called this technique parameterised higher-order abstract syntax. We can supply such a proof:

Theorem 2. $\mathcal{T}[\![\tau_H]\!]\gamma \cong \mathcal{T}[\![\forall\nu.\tau_{WH}(\nu)]\!]\gamma$ $(\cong \mathsf{Term}(0))$.

Proof. Define (in System F) $\phi : \tau_H \to \forall\nu.\tau_{WH}(\nu)$ and $\phi^{-1} : (\forall\nu.\tau_{WH}(\nu)) \to \tau_H$ by:

$$\begin{aligned}\phi &= \lambda t.\Lambda\nu.\Lambda\alpha.\lambda var.\lambda lam.\lambda app.\ t\ [\alpha]\ (\lambda f.\ lam\ (\lambda x.\ f\ (var\ x)))\ app\\ \phi^{-1} &= \lambda t.\Lambda\alpha.\lambda lam.\lambda app.\ t\ [\alpha]\ [\alpha]\ (\lambda x.x)\ lam\ app\end{aligned}$$

Since these functions are terms of System F, the parametricity properties automatically hold. The $\phi^{-1}(\phi(t))$ direction is particularly easy to prove:

$$\begin{aligned}&\phi^{-1}(\Lambda\nu.\Lambda\alpha.\lambda var.\lambda lam.\lambda app.\ t\ [\alpha]\ (\lambda f.\ lam\ (\lambda x.\ f\ (var\ x)))\ app)\\ &= \Lambda\alpha.\lambda lam.\lambda app.\ t\ [\alpha]\ (\lambda f.\ lam\ (\lambda x.\ f\ ((\lambda x.\ x)\ x)))\ app\\ &= \Lambda\alpha.\lambda lam.\lambda app.\ t\ [\alpha]\ lam\ app\\ &= t\end{aligned}$$

In the reverse direction we can prove $\phi(\phi^{-1}(t)) = t$ by applying parametricity over ordinary relations (Kripke relations are not needed here). If we have sets V for ν and A for α, the key idea is to relate the A and V by Rxy iff $x = var\ y$ and relate A and A by the equality relation.

4.2 Locally higher-order abstract syntax

We now consider explicitly representing free variables in terms using any data type we choose, but representing bound variables using higher-order abstract syntax. This approach is inspired by locally nameless representations using de Bruijn indicies only for bound variables [1]. We consider the type:

$$\tau_{LH}(\nu) = \forall\alpha.(\nu \to \alpha) \to ((\alpha \to \alpha) \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha$$

This type has three "constructors", one for injecting free variables of type ν into terms, and the two higher-order abstract syntax constructors. We are free to choose any type we like for ν, such as natural numbers or strings. Selecting naturals, we can define the following combinators:

$$\begin{aligned}&var\ :\ \mathbb{N} \to \tau_{LH}(\mathbb{N})\\ &var = \lambda x.\Lambda\alpha.\lambda v.\lambda l.\lambda a.\ v\ x\\ &app\ :\ \tau_{LH}(\mathbb{N}) \to \tau_{LH}(\mathbb{N}) \to \tau_{LH}(\mathbb{N})\\ &app = \lambda xy.\Lambda\alpha.\lambda v.\lambda l.\lambda a.\ a\ x\ y\\ &lam\ :\ \mathbb{N} \to \tau_{LH}(\mathbb{N}) \to \tau_{LH}(\mathbb{N})\\ &lam = \lambda xt.\Lambda\alpha.\lambda v.\lambda l.\ l\ (\lambda y.\ t\ [\alpha]\ (\lambda x'.\mathsf{if}\ x = x'\ \mathsf{then}\ y\ \mathsf{else}\ v\ x')\ l\ a)\end{aligned}$$

The *var* combinator constructs a term with a single free variable, and *app* constructs the object-level application of two terms. The *lam* combinator is more complicated: for free variable x and term t, it creates a new object-level λ-abstraction, with the body being t with x substituted for the variable bound by the object-level λ-abstraction.

It is also possible to define a pattern matching combinator of type:

$$\tau_{LH}(\mathbb{N}) \to \mathbb{N} + (\tau_{LH}(\mathbb{N}) \times \tau_{LH}(\mathbb{N})) + (\tau_{LH}(\mathbb{N}) \to \tau_{LH}(\mathbb{N}))$$

that analyses a term in our representation, and returns either a free variable, the pair of terms involved in an application, or a term abstracted over another term in the case of object-level λ-abstraction. We cannot give this term here due to lack of space: please see the OCaml files contained with the Coq development.

By using the techniques of Section 3 we can prove that this representation is actually equivalent to a representation using de Bruijn terms. We define such a representation $\mathsf{LNTerm}(A, n)$ inductively by the following constructors:

$$\begin{aligned}
\mathsf{freeVar} &: A \to \mathsf{LNTerm}(A, n)\\
\mathsf{boundVar} &: \{i : \mathbb{N} \mid i < n\} \to \mathsf{LNTerm}(A, n)\\
\mathsf{Lam} &: \mathsf{LNTerm}(A, n+1) \to \mathsf{LNTerm}(A, n)\\
\mathsf{App} &: \mathsf{LNTerm}(A, n) \to \mathsf{LNTerm}(A, n) \to \mathsf{LNTerm}(A, n)
\end{aligned}$$

Theorem 3. *For closed types τ, $\mathsf{LNTerm}(\mathcal{T}[\![\tau]\!]\gamma, 0) \cong \mathcal{T}[\![\tau_{LH}(\tau)]\!]\gamma$.*

The significance of this theorem arises from the fact that we can use a language with parametric polymorphism to represent locally nameless λ-terms; a type that would normally seem to require some kind of indexed types to represent. We speculate that it would be possible to build a convenient (if inefficient) library for manipulating syntax with binders in OCaml using this representation.

5 Mixing computation and representation

We now go beyond the representation of pure syntax to embed the computational power of System F in abstract syntax trees. Licata, Zeilberger and Harper [11] define a system based on a proof theoretic analysis of focusing that allows for a mixing of computational and representational data. Note that the locally higher-order abstract syntax example from the previous section already demonstrates this in action: the $\nu \to \alpha$ constructor for free variables is computational in the sense that it can inspect the values it is given.

5.1 Arithmetic expressions

Our first example is from Licata *et al* [11], that of the abstract syntax of arithmetic expressions with embedded “semantic” binary operations. Binding structure is introduced into the type by a “let” construct. We make the following definition, assuming some primitive type of integers `int`:

$$\begin{aligned}\tau_A = \forall\alpha.\,(\texttt{int} \to \alpha) \to ((\texttt{int} \to \texttt{int} \to \texttt{int}) \to \alpha \to \alpha \to \alpha) \to \\ (\alpha \to (\alpha \to \alpha) \to \alpha) \to \alpha\end{aligned}$$

From the type, we have three "constructors": one to introduce integers into terms, one for terms representing binary operations, with a function expressing the actual operation to perform, and one to handle lets, using the normal higher-order abstract syntax representation for binding. We can write an evaluator for expressions in this type very easily:

$$\mathsf{eval}(t) = t\ [\texttt{int}]\ (\lambda x.\ x)\ (\lambda fxy.\ fxy)\ (\lambda xf.\ fx)$$

A de Bruijn-style representation for these arithmetic expressions is given by the following constructors for an indexed type $\mathsf{AExp} : \mathbb{N} \to \mathsf{Set}$:

$$\begin{aligned}\mathsf{Num} &: \texttt{int} \to \mathsf{AExp}(n)\\ \mathsf{Binop} &: (\texttt{int} \to \texttt{int} \to \texttt{int}) \to \mathsf{AExp}(n) \to \mathsf{AExp}(n) \to \mathsf{AExp}(n)\\ \mathsf{Let} &: \mathsf{AExp}(n) \to \mathsf{AExp}(n+1) \to \mathsf{AExp}(n)\end{aligned}$$

Again, using the same method as Section 3 we can prove:

Theorem 4. *$\mathsf{AExp}(0) \cong \mathcal{T}[\![\tau_A]\!]\gamma$.*

5.2 Encapsulated side-effects with dynamic allocation

The Haskell programming languages contains a monad called ST, that is used to represent encapsulated side-effects with dynamic allocation. A simplified version of this monad, with a single type of data stored in references σ, a type of references ρ and result type τ is given by the following data: a family of types $\mathsf{ST}\ \tau\ \sigma\ \rho$, with associated monadic *return* and *bind* operations, plus three operations:

$$\begin{aligned}\mathit{new}_{\sigma\rho} &: \sigma \to \mathsf{ST}\ \rho\ \sigma\ \rho\\ \mathit{upd}_{\sigma\rho} &: \rho \to \sigma \to \mathsf{ST}\ 1\ \sigma\ \rho\\ \mathit{lkup}_{\sigma\rho} &: \rho \to \mathsf{ST}\ \sigma\ \sigma\ \rho\end{aligned}$$

corresponding to dynamic allocation of a new memory cell, updating a memory cell and looking up the value of a memory cell. This monad has an associated function $\mathsf{runST} : \forall\tau.\forall\sigma.(\forall\rho.\tau_{ST}(\tau,\sigma,\rho)) \to \tau$ that takes a computation and runs it, producing a final result value of type τ. The intention is that the nested quantification over ρ prevents references leaking or entering from outside the computation.

Moggi and Sabry [12] used operational techniques to prove the safety of the full ST monad with typed references. They represent values of the monadic type using a polymorphic type. Simplified to the System F setting with a single type for stored data, this type can be given as:

$$\begin{aligned}\tau_{ST}(\tau,\sigma,\rho) = \forall\alpha.\,(\tau\to\alpha)\to(\sigma\to(\rho\to\alpha)\to\alpha)\to\\ (\rho\to\sigma\to\alpha\to\alpha)\to(\rho\to(\sigma\to\alpha)\to\alpha)\to\alpha\end{aligned}$$

We can make this family of types into a monad with the following definitions:

$$\begin{aligned}
\mathit{return}_{\tau\sigma\rho} &: \tau \to \tau_{ST}(\tau,\sigma,\rho)\\
\mathit{return}_{\tau\sigma\rho} &= \lambda x.\Lambda\alpha.\lambda \mathit{ret}\ \mathit{new}\ \mathit{upd}\ \mathit{lkup}.\ \mathit{ret}\ x\\
\mathit{bind}_{\tau_1\tau_2\sigma\rho} &: \tau_{ST}(\tau_1,\sigma,\rho)\to(\tau_1\to\tau_{ST}(\tau_2,\sigma,\rho))\to\tau_{ST}(\tau_2,\sigma,\rho)\\
\mathit{bind}_{\tau_1\tau_2\sigma\rho} &= \lambda cf.\Lambda\alpha.\lambda \mathit{ret}\ \mathit{new}\ \mathit{upd}\ \mathit{lkup}.\,c[\alpha](\lambda x.\ fx[\alpha]\mathit{ret}\ \mathit{new}\ \mathit{upd}\ \mathit{lkup})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{new}\ \mathit{upd}\ \mathit{lkup}
\end{aligned}$$

Note that, unlike Moggi and Sabry, we have not included a "constructor" in our type to represent *bind*, it can already be defined from the *ret* "constructor". We define the operations of the monad like so:

$$\begin{aligned}
\mathit{new}_{\sigma\rho} &= \lambda s.\Lambda\alpha.\lambda \mathit{ret}\ \mathit{new}\ \mathit{upd}\ \mathit{lkup}.\ \mathit{new}\ s\ (\lambda r.\ \mathit{ret}\ r)\\
\mathit{upd}_{\rho} &= \lambda rs.\Lambda\alpha.\lambda \mathit{ret}\ \mathit{new}\ \mathit{upd}\ \mathit{lkup}.\ \mathit{upd}\ r\ s\ (\mathit{ret}\ *)\\
\mathit{lkup}_{\sigma\rho} &= \lambda r.\Lambda\alpha.\lambda \mathit{ret}\ \mathit{new}\ \mathit{upd}\ \mathit{lkup}.\ \mathit{lkup}\ r\ (\lambda s.\ \mathit{ret}\ s)
\end{aligned}$$

Using these combinators we can write programs in monadic style that issue commands to dynamically allocate new memory cells via the *new* operation and access them using the *upd* and *lkup* operations.

Moggi and Sabry note that (their version of) the type $\tau_{ST}(\tau,\sigma,\rho)$ almost fits the schema for the polymorphic representation of an inductive type in System F, were it not for the negative occurrence of ρ in the *new* "constructor". Using the techniques of Section 3, we can show that this type actually does correspond to an inductively defined type using de Bruijn representation for variables. The appropriate type is given by by the following constructors for an indexed type $\mathsf{ST}(A,S,-):\mathbb{N}\to\mathsf{Set}$, for sets A and S.

$$\begin{aligned}
\mathsf{Ret} &: A\to\mathsf{ST}(A,S,n)\\
\mathsf{New} &: S\to\mathsf{ST}(A,S,n+1)\to\mathsf{ST}(A,S,n)\\
\mathsf{Update} &: \{i:\mathbb{N}\mid i<n\}\to S\to\mathsf{AExp}(A,S,n)\to\mathsf{AExp}(A,S,n)\\
\mathsf{Lookup} &: \{i:\mathbb{N}\mid i<n\}\to(S\to\mathsf{AExp}(A,S,n))\to\mathsf{AExp}(A,S,n)
\end{aligned}$$

Theorem 5. *For closed types τ and σ,*

$$\mathsf{ST}(\mathcal{T}[\![\tau]\!]\gamma,\mathcal{T}[\![\sigma]\!]\gamma,0)\cong\mathcal{T}[\![\forall\rho.\tau_{ST}(\tau,\sigma,\rho)]\!]\gamma.$$

An obvious question now is whether this result extends to the case with typed references. Following Moggi and Sabry, we would expect that the F_ω type

$$\begin{aligned}
\lambda\tau.\forall\rho:*\to*.\forall\alpha.\,&(\tau\to\alpha)\to\\
&(\forall\sigma.\ \sigma\to(\rho[\sigma]\to\alpha)\to\alpha)\to\\
&(\forall\sigma.\ \rho[\sigma]\to\sigma\to\alpha\to\alpha)\to\\
&(\forall\sigma.\ \rho[\sigma]\to(\sigma\to\alpha)\to\alpha)\to\\
&\alpha
\end{aligned}$$

should have a de Bruijn-style representation similar to ST above. However, there is a problem with proceeding naively here. Consider the following program written in this monad (using Haskell's do notation):

$$\begin{array}{l} \mathtt{do}\ x \leftarrow \mathit{new}\ (\lambda().\ \mathit{return}\ ()) \\ \phantom{\mathtt{do}\ }\mathit{upd}\ x\ (\lambda().\ \mathtt{do}\ \{y \leftarrow \mathit{lkup}\ x;\ y\ ()\}) \\ \phantom{\mathtt{do}\ }y \leftarrow \mathit{lkup}\ x \\ \phantom{\mathtt{do}\ }y\ () \end{array}$$

which uses “Landin's knot” to represent a non-terminating computation using mutable references. However, the “obvious” de Bruijn-style type (using a context consisting of lists of types) does not admit the translation of this term.

6 Related Work and Conclusions

Aside from the work of Washburn and Weirich [18], the closest work to ours is that of Rhiger [17], who shows that a higher-order abstract syntax encoding for a single typed object-language is sound and complete in a simply-typed meta-language with a type constructor $\mathsf{Exp} : * \to *$. We have extended his work by allowing multiple embedded languages. The use of System F also allows the use of iteration constructs to access terms from the outside, as demonstrated by Washburn and Weirich.

Also related is the work of Carette *et al* [2]. They use the same method as Rhiger to embed languages inside an existing typed language (OCaml in this case). They abstract over the carrier type and actual implementations of *lam* and *app*, as we do here, but do not make the connection to concrete terms explicit.

It seems obvious, though we have not yet formally proved it, that there is a natural extension of the representation of inductive types in System F as polymorphic types $\forall\alpha.(F[\alpha] \to \alpha) \to \alpha$, where α is positive in F to ones, where we allow negative a occurrences, and the represented type is some kind of abstract syntax with binding. We leave formulating and proving a general theorem of this kind to future work, but we suspect that it will be a straightforward application of the ideas in Section 3, the key idea being the use of Kripke logical relations.

In future work we also wish to consider more powerful type theories than System F for use as the meta-language. An obvious first step is the use of System F_ω, which will allow the use of type parameters to represent object languages with type systems that are subsets of the meta-language type system, although the case of the multi-typed ST monad from Section 5.2 shows that this extension may not be straightforward. Pfenning and Lee [14] have considered the use of F_ω as a meta language, using a form of weak higher-order abstract syntax, but did not prove the close connection between representation and syntax that we have here. A yet more powerful route may be to consider the combination of dependent types and parametric polymorphism, so that representations of logics in the same style as the Logical Framework approach maybe used, combined with powerful ways of computing with them. The work of Izumi [10] on parametricity in dependent types may be useful here.

Acknowledgements Thanks to Randy Pollack, Sam Staton and Jeremy Yallop for comments on this work. This work was funded by the ReQueST grant (EP/C537068) from the Engineering and Physical Sciences Research Council.

References

1. Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In George C. Necula and Philip Wadler, editors, *POPL*, pages 3–15. ACM, 2008.
2. Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated. In Zhong Shao, editor, *APLAS*, volume 4807 of *LNCS*, pages 222–238. Springer, 2007.
3. Adam J. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP*, pages 143–156. ACM, 2008.
4. Thierry Coquand and Gérard Huet. Constructions: A higher order proof system for mechanizing mathematics. In *EUROCAL85*, volume 203 of *LNCS*, pages 151–184, 1985.
5. N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34:381–392, 1972.
6. Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz. Higher-Order Abstract Syntax in Coq. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *TLCA*, volume 902 of *LNCS*, pages 124–138. Springer, 1995.
7. Murdoch Gabbay and Andrew M. Pitts. A New Approach to Abstract Syntax Involving Binders. In *LICS*, pages 214–224, 1999.
8. Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, 2007.
9. Martin Hofmann. Semantical Analysis of Higher-Order Abstract Syntax. In *LICS*, pages 204–213, 1999.
10. Takeuti Izumi. The Theory of Parametricity in Lambda Cube. Technical Report 1217, RIMS Kokyuroku, 2001.
11. Daniel R. Licata, Noam Zeilberger, and Robert Harper. Focusing on Binding and Computation. In *LICS*, pages 241–252. IEEE Computer Society, 2008.
12. Eugenio Moggi and Amr Sabry. Monadic encapsulation of effects: a revised approach (extended version). *J. Funct. Program.*, 11(6):591–627, 2001.
13. Frank Pfenning and Conal Elliott. Higher-Order Abstract Syntax. In *PLDI*, pages 199–208, 1988.
14. Frank Pfenning and Peter Lee. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science*, 89:137–159, 1991.
15. Gordon D. Plotkin. Lambda-Definability in the Full Type Hierarchy. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, 1980.
16. John C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*, pages 513–523, 1983.
17. Morten Rhiger. A foundation for embedded languages. *ACM Trans. Program. Lang. Syst.*, 25(3):291–315, 2003.
18. Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *J. Funct. Program.*, 18(1):87–140, 2008.