A Type for Lambda – DRAFT

Polymorphically Typed Analysis of Simply Typed Programs

Barry Jay

Centre for Artificial Intelligence
University of Technology Sydney
Barry.Jay@uts.edu.au

Abstract. Abstraction calculus represents variables, environments and lambda-abstractions by combinations of six fundamental operators, including the abstraction operator, called A or lambda. When applied to an environment s and a body t it produces a closure Ast. The type of lambda uses three kinds of function type: that of t is from a type context to a simple type; that of s is from a type context to another such; the type of A is higher-order.
The resulting combinations can be analysed using a further six operators, namely: the traditional combinators S and K; a recursion operator R, the factorisation operator F which is used to decompose compounds into their components; and two operators E and D for recognising other operators when selecting from programs or updating them. In this manner, the calculus can support polymorphically-typed analyses of simply-typed programs.

Keywords: typed lambda calculus, program analysis, intensional computation

1 Introduction

Abstraction calculus represents variables, λ-abstractions, and environments as combinations of a small set of operators, whose intensionality cannot be expressed in traditional combinatory logic. The β-reduction rule is given by

$$Astu \longrightarrow Bust$$

which is to be understood as follows. A is the abstraction operator that binds the zeroth index in the body t of the abstraction, all in the environment given by the substitution s. Reduction creates a new environment Bus in which the zeroth index is bound to u, and applies this to t. A type for A (also called λ) is thus

$$\lambda : (\Gamma \to \Delta) \to (U * \Delta \to T) \to (\Gamma \to U \to T).$$

That is, if s produces an environment of type Δ in the context Γ and t produces a term of type T in the context $U * \Delta$ then λst produces a term of type $U \to T$ in context Γ. Similarly, we have

$$B : (\Gamma \to U) \to (\Gamma \to \Delta) \to (\Gamma \to U * \Delta)\ .$$

The use of B makes abstraction calculus a form of *explicit substitution calculus* [1, 11] but the motivation is new.

In a way, this is all quite straightforward: terms are combinations of operators; and types are built from product and function types. However, there are three sorts of application in the β-reduction rule, each of which is typed differently. The application of λst to u uses a fixed context Γ. The application of Bus to t changes the context from $U * \Delta$ to Γ. The other applications are higher-order and do not concern themselves with contexts at all. Happily, this can all be managed by introducing four *kinds*:

$$\kappa ::= s \mid c \mid h \mid q$$

which represent the *simple types*, *type contexts*, *higher types* and *quantified types*, respectively. Then the types are given by

$$U, T, \Gamma, \Delta ::= (X : \kappa) \mid \mathbf{1} \mid U * \Gamma \mid U \to T \mid \forall (X : \kappa).T$$

for (kinded) type variables, the unit type (for the empty context), product types (for enlarging the context), function types and quantified types.

The terms are built from operators by application. A minimal set of operators is given by the *closure operators*

$$J \mid N \mid H \mid A \mid I \mid B\ .$$

J and N play the role of zero and successor in building the indices (in the de Bruijn style) that play the role of variables. H is for *holding* or *freezing* an application wihch adds some necessary fine control over reduction. A is for abstraction. I is the empty environment or substitution, which grows by using B to add bindings.

These operators are enough to resolve a long-standing problem concerning the relationship between λ-calculus [2] and combinatory logic (built using the operators S and K) [5]. Although there is a reduction-preserving translation of combinatory logic to pure λ-calculus, the translation in the other direction does not preserve reduction. Rather, in the process of analyzing the abstraction body, it breaks every redex that appears. Here, λ becomes a free-standing operator, despite its role in binding, and the translation from λ-calculus to abstraction calculus is an embedding that preserves reduction. More precisely, there is a variant of λ-calculus called *closure calculus* (in press) which can be embedded into abstraction calculus.

That done, we can consider the possibilities for program analysis. Traditionally, analysis begins by quoting the program, to obtain a syntax tree, e.g. to develp a typed self-interpreter [4]. Here, analysis will be within the calculus itself, *without* the need for quotation, or other meta-programming. This has previously been achieved in SF-calculus [9], where recursive application of the factorisation operator F can be used to define program analyses [8], including a typed self-interpreter [10]. In particular, the Gödel number of a program can be so defined, so that any Turing computable analysis is definable. This approach has suffered from two limitations. First, SF-calculus does not provide native support for λ-abstraction, and it is a major effort to add it directly [7]. Second, the typing has been fragile, due in part to the difficulties of polymorphic typing in a combinational setting. Thse limitations are overcome by adding some more operators

$$S \mid K \mid R \mid F \mid E \mid D$$

which together support the pattern-matching functions necessary to perform interesting analyses.

S and K are used to provide indirect support for abstraction at higher kinds. R builds fixpoint functions in a manner similar to the traditional fixpoint operator Y but it requires two arguments before reducing. In this manner, a fixpoint function Rf is normal if f is, and we can insist that all programs, even recursive ones, be in normal form. F is used to decompose normal forms into their components, using the rule

$$F(pq)tu \longrightarrow upq \text{ if } pq \text{ is a compound.}$$

where the compounds turn out to be the partially applied operators. Its type is

$$F : U \to T \to (\forall(Z : h).(Z \to U) \to (Z \to T)) \to T\ .$$

Note that its third argument must be a polymorphic function since there is no way to infer the type of q from that of pq. E is the equality operator, of type

$$E : U \to V \to T \to T \to T\ .$$

It compares arguments of arbitrary type to produce a bolean, expressed as a function $T \to T \to T$. These operators are enough to define functions that select or filter sub-terms from a structure, but is unable to type updates that modify a structure without changing its type. This requires *discriminators* of the form DO . If the operator O has principal type scheme $\mathsf{Ty}[O]$ then DO has type

$$DO : (T \to T) \to T \to \mathsf{Ty}[O] \to T\ .$$

That reduction preserves typing has been verified using the Coq proof assistant [**?**].

Together, these operators are enough to support an interesting class of pattern-matching functions, in the style of pattern calculus [6]. Examples include: the test for being in the image of closure calculus; an update that forces evaluation of frozen applications. Indeed, there is a pattern-matching account of an eager self-interpreter which purports to be of type $\forall(X : h).X \to X$, , though this has yet to be formally verified.

The sections of the paper are as follows. The terms of abstraction calculus are introduced in Section 2. The types of abstraction calculus are introduced in Section 3. Section 4 introduces lambda-abstraction at higher types. Section 5 introduces pattern-matching at higher types. Section 6 proposes a typed self-interpreter. Section 7 draws conclusion. All named theorems have been verified in Coq.

2 Terms

The *operators* O and *combinations* M, N of *abstraction calculus* are given by

$$\begin{aligned} O &::= J \mid N \mid H \mid A \mid I \mid B \mid S \mid K \mid R \mid F \mid E \mid D \\ i, s, t, u &::= O \mid tu \ . \end{aligned}$$

Each operator has an *arity* given by $1, 2, 3, 3, 1, 3, 3, 2, 2, 3, 2$ and 4 respectively. The resulting calculus is easy to understand, in that each term is a combination built from a finite set of operators. However, to type D is beyond the expressive power of the types that will be used here, as it is polymorphic in the choice of operator, whose type are themselves polymorphic. The first difficulty is eliminated by excluding D as a (typed) term, but allowing DO to have a type for any operator O. This all becomes a little clumsy, so let us recast the operators as a recursive class

$$O ::= J \mid N \mid H \mid A \mid I \mid B \mid S \mid K \mid R \mid F \mid E \mid DO$$

in which D is rquired to take an operator as argument. Now there is an infinite collection of operators: this is conceptually troublesome but is easy to work with in practice, so we shall adopt this approach. The operator arities are as before, except that DO now has arity 3.

The *atoms* are the operators. The *compounds* are the partial applications pq, i.e. where the head operator of p is applied to fewer arguments than its arity. The terms p and q are then the *left-* and *right-hand components* of pq. The *variables* are the terms of the form J or Ni. The *lambda forms* are those terms of the form

$$J, Ni, Htu, Bus,\ Ast\ .$$

We may write λst for Ast. These will be used to represent terms of λ-calculus. All other compounds are *true compounds.* The *factorable forms* are the atoms and the compounds. The number of possible terms forms that are factorable is given by the sum of the arities, which is 28.

The reduction rules are given in Figure 1. The operators J, N, H, A, I, S, K and R are extensional in that their reduction is given by a single rule that does not inspect the internal structure of the arguments. The operators B, F, E and D are intensional: B queries its third argument; F queries its first argument, while E and D compare two of their arguments.

As written, there are 23 reduction rules with side-conditions. However, to eliminate the side-conditions, the rules for F can be expanded from 2 to 28. Similarly, the rules for B expand from 7 to 28 while those for D and E expand to $28 * 28 = 784$ rules! Thus, the total number of rules without side-conditions is $8 + 2 * 28 + 2 * 784 = 1632$ rules! Fortunately, the proofs, including the formal proofs in Coq, are able to work with the conditional rules.

The rules can be understood as follows. The representation of λ-abstraction is driven by the rules for applying a (non-trivial) substitution Bus. If the argument is a fully applied operator then it should be reduced before substitution

$$\begin{aligned}
Jt &\longrightarrow HJt \\
Nit &\longrightarrow H(Ni)t \\
Htuv &\longrightarrow H(Htu)v \\
Astu &\longrightarrow Bust \\
It &\longrightarrow t \\
BusJ &\longrightarrow u \\
Bus(Ni) &\longrightarrow si \\
Bus(Htv) &\longrightarrow (Bust)(Busv) \\
Bus(Bvt) &\longrightarrow B(Busv)(Bust) \\
Bus(Avt) &\longrightarrow A(Busv)t \\
BusO &\longrightarrow O \quad (O \neq J \text{ is an operator}) \\
Bus(pq) &\longrightarrow Busp(Busq) \quad (pq \text{ is a true compound}) \\
Stuv &\longrightarrow tv(uv) \\
Ktu &\longrightarrow t \\
Rtu &\longrightarrow t(Rt)u \\
FOtu &\longrightarrow t \quad (O \text{ is an operator}) \\
F(pq)tu &\longrightarrow upq \quad (pq \text{ a compound}) \\
EOO &\longrightarrow K \quad (O \text{ is an operator}) \\
EOu &\longrightarrow K(SKK) \quad (O \text{ is an operator, } u \neq O \text{ is factorable}) \\
EtO &\longrightarrow K(SKK) \quad (t \text{ is a compound, } O \text{ is an operator}) \\
E(tu)(pq) &\longrightarrow Etp(Euq)(K(SKK)) \quad (tu \text{ and } pq \text{ are compounds}) \\
DOtOv &\longrightarrow v \quad (O \text{ is an operator}) \\
DOtpv &\longrightarrow tp \quad (p \neq O \text{ is factorable})
\end{aligned}$$

Fig. 1. Reduction rules of abstraction calculus

is performed. This prevents there being any conflict between reduction rules; there are no critical pairs. If the argument is the zeroth index J then substitution yields the zeroth term u in the substitution Bus. If the argument is some other index Ni then apply s to i. In this manner, the implicit indices in the substitution are reduced by one, just as Ni is reduced to i.

Now consider where the argument is the application of a variable. In standard accountsof λ-calculus, an application $xv_1, \ldots v_n$ of a variable is head normal. If we stay with this approach then the corresponding rule for substitutions would be

$$Bus(xv_1 \ldots v_n) \longrightarrow (Busx)(Busv_1) \ldots (Busv_n)?$$

This, of course, would be a countable family of rules, indexed by n which is unpleasant. In practice, one must decompose the application to find its head, so we may as well record this work in reduction rules, by tagging, or freezing, or holding these applications by the reductions

$$xv_1 v_2 \ldots v_n \longrightarrow x@v_1 v_2 \ldots v_n \longrightarrow x@v_1@v_2@ \ldots @v_n \ .$$

Formally, we write Htv for $t@v$. Now the intended reductions above are given by the rules $Jt \longrightarrow HJt$ and $Nit \longrightarrow H(Ni)t$ and $Htuv \longrightarrow H(Htu)v$.

Of course, when u is substituted for x in $x@v$ then the resulting application of u should be thawed so that u may act on the (substitution of) v. This yields the rule for applying Bus to Htv.

When Bus is applied to another substitution Bvt then Bus is applied to v and to t. When Bus is applied to the empty environment I then the result is I. That is, the domain of a substitution does not grow under substitution.

The most interesting case arises when the argument of Bus is an abstraction Avt. Now Bus is applied to the environment v but not to the body t. That is, the environment shadows the body of the closure.

Now let us consider the second half of the operators. The reduction rules for S and K are taken from combinatory logic. The rule for R show that it creates fixpoint functions, but not fixpoint values. That is, Rt is a compound, but $Rtu \longrightarrow t(Rt)u$ shows that Rt is a fixpoint function. Note that R itself does not introduce any non-termination. In particular, the primitive recursive functions are srongly normalizing, and not merely weakly normalizing.

The rules for F are taken from SF-calculus. F branches after querying its first argument r: dividing it if r is a compound, and conquering if r is an atom.

The rules for E show that it is an equality operator, proceeding to divide-and-conquer both its arguments.

The rules for D show that it discriminates, according to whether its third arguments matches its first argument, which must be an operator. In an untyped setting, D duplicates the work of E but in a typed setting, E will be used to define queries and selections, while D will be used to define updates.

Theorem 1. *Reduction of abstraction calculus is confluent.*

Proof. There are no critical pairs.

The normal forms are given by the factorable forms whose sub-terms are all normal.

Theorem 2. *Every combination of abstraction calculus is either a normal form, or reduces.*

Proof. A straightforward induction on the structure of the term suffices.

3 Types

The traditional type derivation rule for a λ-abstraction is

$$\frac{x : U, \varGamma \vdash t : T}{\varGamma \vdash \lambda x.t : U \to T} \; .$$

That is, type judgments are of the form $\varGamma \vdash t : T$ where t is a term, T is a type and $\varGamma$ is a type context, given by a set or sequence of typings for term variables $x_i : U_i$. The rule for λ-abstraction derives $\varGamma \vdash \lambda x.t : U \to T$ from a derivation of $x : U, \varGamma \vdash t : T$.

By using de Bruijn indices i in place of variables x_i, this rule can be recast as

$$\frac{U * \varGamma \vdash t : T}{\varGamma \vdash \lambda t : U \to T}$$

where U in $U * \varGamma$ is the type of the variable indexed by zero. Further, if $\varGamma$ is, say $V * \mathbf{1}$ then V becomes the type of the variable indexed by 1 as required by the de Bruijn style.

Further, if the abstraction is given an environment or substitution s to form a *closure* then we have the rule

$$\frac{\varGamma \vdash s : \varDelta \quad U * \varDelta \vdash t : T}{\varGamma \vdash \lambda st : U \to T}$$

in which the context $\varDelta$ is the range of the substitution s. Now that the type contexts are given by tuples of types, with no reference to term variables, we can declare them to be types, and place them on the right, to get

$$\frac{s : \varGamma \to \varDelta \quad t : U * \varDelta \to T}{\lambda st : \varGamma \to U \to T} \; .$$

Finally, by currying, we have a type for λ itself, namely

$$\lambda : (\varGamma \to \varDelta) \to (U * \varDelta \to T) \to (\varGamma \to U \to T).$$

Certainly, this typing by product and function types reminds us of cartesian closed categories, but there are some subtleties. Consider the β-reduction rule $Astu \longrightarrow Bust$ where A is our operator for λ. Given $s : \varGamma \to \varDelta$ then $As : (U * \varDelta \to T) \to (\varGamma \to U \to T)$ just as usual. Similarly, given $t : U * \varDelta \to T$ then

$Ast : \mathit{\Gamma} \to U \to T$. However, the argument of this abstraction is a term u whose type is U in context $\mathit{\Gamma}$. That is, $u : \mathit{\Gamma} \to U$. This application *cannot* be typed in the usual manner. Rather, we require that

$$\frac{Ast : \mathit{\Gamma} \to U \to T \quad u : \mathit{\Gamma} \to U}{Astu : \mathit{\Gamma} \to T}$$

so that the context can be shared. Further, the β-redex must be typed by

$$\frac{Bus : \mathit{\Gamma} \to U * \mathit{\Delta} \quad t : U * \mathit{\Delta} \to T}{Bust : \mathit{\Gamma} \to T} \; .$$

All up, we have three different rules for typing applications, according to whether the function maps: a context to a simple type; a context to a context; or one higher type to another.

The solution adopted is to introduce a primitive system of *kinds* given by

$$\kappa ::= s \mid c \mid h \mid q \; .$$

They characterise the *simple types*, the *context types*, the *higher types*, and the *quantified types* respectively.

Based on the account above, it is not yet clear why the quantified types will need a separate kind. If this does not trouble you, then feel free to move on to the next paragraph. In brief, the argument is as follows. First, program analysis will require factorisation, which splits a compound pq into its component p and q. In general, the type of q is not derivable from that of pq so the type of F will require a universally quantified argument type. That is, we require the full power of System F. Second, type-level operations, such as type instantiation, are made explicit in the terms by, say applying a term t to a type U to get tU then this is not easy to incorporate into the combinatory style, or to decide how such compounds might be factored. Hence, type-level operations will be implicit. Third, implicit operations make proofs about type derivation more difficult. To impose some control, terms may take types of the form $t : (\forall X.U) \to T$ but T may not be quantified. This can be applied to $u : U$ with the generalization of U to $\forall X.U$ being implicit. Note that there is no side-condition on the occurrence of X in the context, since there is no context outside of U. Fourth, if tu above were a compound then factorisation would not be typable without finding some way to incorporate the generalization of U to $\forall X.U$. Hence, we must ensure that such tu are never compounds but are always fully applied operators.

The *types* are given by

$$T, U ::= (X : \kappa) \mid \mathbf{1} \mid U * T \mid U \to T \mid \forall (X : \kappa).T \; .$$

We may write X for $X : \kappa$ where the kind κ is understood from the circumstances. We may use the meta-variables $\mathit{\Gamma}$ and $\mathit{\Delta}$ to represent context types. Substitution $\{U/X\}T$ of a type $U : \kappa$ for a variable $X : \kappa$ in a type T is routine.

The *kinding rules* are given in Figure 2. That is, simple types are built from variables of simple kind to form function types. Some type constants could be added here, but will not feature in the sequel. The context types are given by products $U * V * \ldots * \Gamma$ where Γ is either a variable of the unit type. The higher types are either functions from contexts to (simple types or contexts) or are functions from higher types to higher types. The quantified types are generated by (the variables) and the higher types, and closed under universal quantification.

$$\frac{U : s \quad V : s}{U \to V : s} \qquad \frac{}{\mathbf{1} : c} \qquad \frac{U : s \quad \Gamma : c}{U * \Gamma : c}$$

$$\frac{\Gamma : c \quad U : s}{\Gamma \to U : h} \qquad \frac{\Gamma : c \quad \Delta : c}{\Gamma \to \Delta : h} \qquad \frac{U : h \quad T : g}{U \to T : h}$$

$$\frac{T : h}{T : q} \qquad \frac{}{(X : \kappa) : \kappa} \qquad \frac{T : h}{\forall (X : \kappa).T : h}$$

Fig. 2. Kinding rules

The *substitution* $\{U/X : \kappa\}T$ of a type $U : \kappa$ for a variable $X : \kappa$ is defined in the standard manner. The *instantiation relation* between quantified types is the reflexive, transitive closure of the rule

$$\forall (X : \kappa).T < \{U/X : \kappa\}T \ .$$

We may write $T \prec U$ for $T < U$ if U is of higher kind. Derived kinds are stable under substitution. The *generalization relation* $U > V$ is the reflexive, transitive closure of the rule

$$U > \forall (X : \kappa).U$$

which adds quantifiers.

Each operator O has a *principal type scheme* $\mathsf{Ty}[O]$ of quantified kind, as given in Figure 3. The types for the operators used to model λ-calculus make heavy use of simple type and contexts. The other operators act on higher types only. No variables of quantified kind have been used. Of special interest is the type of DO which requires an argument of type $\mathsf{Ty}[O]$.

The rules for building *well-typed combinations* are given in Figure 4. Note that the derived type is always of higher kind, and never of quantified kind. Hence, when typing an operator, it is necessary to instantiate its principal type. Conversely, when a function $t : V \to T$ has an argument type V that is a quantified type, then the type U of the argument u must generalize to V. Traditionally, the ability to introduce a quantifier was conditional on the state of the type context. For example, one might infer $\Gamma \vdash t : \forall X.T$ from $\Gamma \vdash t : T$ if X was not free in Γ. Here, however, we have $t : \Gamma \to T$ so that generalization produces $\forall X.\Gamma \to T$ and no constraint is required. The last type derivation rule is for *weakening* the context, to replace the empty context by any other such. This

$$\begin{aligned}
J &: \forall(X:c, Y:s).Y * X \to Y\\
N &: \forall(X:c, Y:s, Z:s).(X \to Y) \to (Z * X \to Y)\\
H &: \forall(X:c, Y:s, Z:s).(X \to Y \to Z) \to (X \to Y) \to (X \to Z)\\
A &: \forall(W:c, X:c, Y:s, Z:s).(W \to X) \to (Y * X \to Z) \to (W \to Y \to Z)\\
I &: \forall(X:c).X \to \mathbf{1}\\
B &: \forall(X:c, Y:s, Z:c).(X \to Y) \to (X \to Z) \to (X \to Y * Z)\\
S &: \forall(X:h, Y:h, Z:h).(X \to Y \to Z) \to (X \to Y) \to (X \to Z)\\
K &: \forall(X:h, Y:h).X \to Y \to X\\
R &: \forall(X:h, Y:h), ((X \to Y) \to (X \to Y)) \to (X \to Y)\\
F &: \forall(X:h, Y:h).X \to Y \to (\forall(Z:h).(Z \to X) \to (Z \to Y)) \to Y\\
E &: \forall(X:h, Y:h, Z:h).X \to Y \to Z \to Z \to Z\\
DO &: \forall(X:h).(X \to X) \to X \to \mathsf{Ty}[O] \to X\ .
\end{aligned}$$

Fig. 3. Principal types of the operators

is required to ensure that the reduction $It \longrightarrow t$ preserves typing. Note that derived types are stable under substitution.

$$\frac{}{O:T}\ \mathsf{Ty}[O] \prec T \qquad \frac{t:\Gamma \to (U \to V : s) \quad u:\Gamma \to U}{tu:\Gamma \to V}$$

$$\frac{s:\Gamma \to (\Delta : c) \quad u:\Delta \to U}{su:\Gamma \to U} \qquad \frac{t:V \to (T:g) \quad u:U}{tu:T}\ U > V$$

$$\frac{t:\mathbf{1} \to T}{t:(\Gamma : c) \to T}$$

Fig. 4. Typing rules

For example, the identity function AIJ is typed by

$$\dfrac{\dfrac{\dfrac{}{A:(\mathbf{1} \to \mathbf{1}) \to (U * \mathbf{1} \to U) \to \mathbf{1} \to U \to U} \qquad \dfrac{}{I:\mathbf{1} \to \mathbf{1}}}{AI:(U * \mathbf{1} \to U) \to (\mathbf{1} \to U \to U)} \qquad \dfrac{}{J:U * \mathbf{1} \to U}}{AIJ:\mathbf{1} \to U \to U}$$

Lemma 1 (compound_types). *If $pq : T$ is a compound then there is a type U such that $p : U \to T$ and $q : U$.*

Proof. The proof is by case analysis on the possible compounds.

Theorem 3 (subject_reduction). *Reduction preserves typing.*

Proof. The proof is long, but routine. It has been verified in Coq.

4 Abstraction of Terms Having Higher Types

Program analyses will be defined as pattern-matching functions where the patterns and bodies of the cases are of higher type. The simplest form of pattern is a variable, where the corresponding case becomes an abstraction. However, we do not have variables or indices of higher type, much less their abstraction, so must resort to the standard combinatorial techniques. First , extend the terms with a class of (typed) variables $x, y, z, \ldots$ and then define $\lambda^* x.t$ by

$$\begin{aligned}
\lambda^* x.x &= SKK \\
\lambda^* x.y &= Ky \\
\lambda^* x.O &= KO \\
\lambda^* x.tu &= (\lambda^* x.t)(\lambda^* x.u)\ .
\end{aligned}$$

Note that if $t : T : h$ and $x : U : h$ then $\lambda^* x.t : U \to T$.

For example, consider the problem of recognising an index, of the form $N^i J$ by some term $\mathsf{is\_index}$. Given a candidate x first factorise it by F. If it is an operator, then it must be J as determined by EJx. If it is a compound yz then y must be R and z must be an index, as determined by $ERy(\mathsf{is\_index}\ z)(K(SKK))$. Note how K is used to represent truth and $K(SKK)$ is used to represent falsehood. Putting these together yields

$$\lambda^* x.Fx(EJx)(\lambda^* y.\lambda^* z.ERy(\mathsf{is\_index}\ z)(K(SKK)))\ .$$

Then the recursion is handled by replacing $\mathsf{is\_index}$ by a variable f, binding f and applying R to get

$$\mathsf{is\_index} = R(\lambda^* f.\lambda^* x.Fx(EJx)(\lambda^* y.\lambda^* z.ERx(fy)(K(SKK))))\ .$$

This has type $\forall(X : h, Y : h).X \to Y \to Y \to Y$. That is, it is a boolean test for being an index.

5 Pattern-Matching

Intensional calculi are built to support pattern-matching. A pattern-matching function can be represented as an *extension* of the form $p \Rightarrow s \mid r$ where $p \Rightarrow s$ is the special case and r is the *default function.* Often r will also be an extension, but the ultimate default case must be able to act on anything left over. In general, the pattern p may require reduction before matching can occur, but for simplicity, let us assume that the pattern p is a normal form, so that the patterns are static. Now consider the structure of p.

If p is a variable x then the extension is given by $\lambda^*x.s$.

If p is a compound, of the form p_1p_2 then use F to factorise u. If u is an atom then matching fails. If u is a compound u_1u_2 then match p_1 against u_1 and p_2 against u_2.

The delicate situation arises when p is an operator O. Consider how to type

$$O \Rightarrow s \mid r : U \to T \ .$$

Since r is the default function, it must have type $U \to T$.

The simplest case is when s has type T. Then the extension can be given by $\lambda^*x.EOxs(tx)$. This is enough to type all queries that return a fixed type, regardless of their argument, such as is_index above. In the same spirit, it can be used to type the function is_cc that recognises the normal forms that are in the image of closure calculus. Let $b_1 \wedge b_2 = b_1b_2(K(SKK))$ be the conjunction of booleans. Then is_cc is defined by

```
is_cc $t$ =
match $t$ with
 | $J \Rightarrow K$
 | $Ri \Rightarrow$ is_index $i$
 | $Htu \Rightarrow$ is_cc $t \wedge$ is_cc $u$
 | $Ast \Rightarrow$ is_cc $s \wedge$ is_cc $t$
 | $I \Rightarrow K$
 | $Bus \Rightarrow$ is_cc $u \wedge$ is_cc $s$
 | $pq \Rightarrow$ is_cc $p \wedge$ is_cc $q$
 | _ $\Rightarrow K(SKK)$
```

where the tests for being J is given by EJ etc.

The other key case arises when the extension is used to perform an update. In this setting, the extension has type $U \to U$ or even $X \to X$. Now s must be able to take any type that O does. This is ensured by using DO so that the extension $O \Rightarrow s \mid r$ is here interpreted by

$$\lambda^*x.DOrxs \ .$$

Of course, by changing the order of arguments in the reduction rule for D, this term could be replaced by $DOsr$ but then factoring the compound DOs would break the typing, since we cannot infer that s has quantified type $\mathsf{Ty}[O]$. That is, it is important that the argument with the quantified type come last.

Further generality would require a variant of D, say $D^?$ of type

$$D^?O : \{\mathsf{Ty}[O] \prec X\}Y \to (X \to Y) \to (X \to Y)$$

in which the type system would be required to support type constraints, such as the requirement that X be an instantiation of $\mathsf{Ty}[O]$. However, the ability to query and update supported by E and D is enough for many purposes, so there is no rush to add further complexity.

An example of updating is the function thaw which replaces H by the identitiy function SKK. It is given by the pattern-matching function

```
thaw $t$ =
match $t$ with
 | $xy \Rightarrow$ thaw $x$ (thaw $y$)
 | $H \Rightarrow SKK$
 | $x \Rightarrow x$
```

which desugars to

$$R(\lambda^* f.\lambda^* x.Fx(DH(SKK)x(SKK))(\lambda^* y.\lambda^* z.fy(fz)))$$

and has type $(X : h) \to X$.

6 Typed Self-Interpretation

A good example of an update is the self-interpreter given in Figure 5, of type $(X \to Y) \to (X \to Y)$. Evaluation proceeds by pattern-matching against its argument $p : X \to Y$ and then, perhaps, by matching against then next argument $x : X$. The evaluation is eager, in that, for example, the evaluation of Stu applied to x evaluates the application of u to x before that of tx to ux. In essence, the interpreter is built by recasting the reduction rules in Figure **??** as cases of the function. However, some rules are not mentioned explicitly. For example, to evaluate Kt applied to u it is enough to use the default case, which yields Ktu and thus t as desired. That is, some of the interpretation is meta-circular.

```
eval $p$ =
match $p$ with
 | $Stu \Rightarrow \lambda^* x.$eval (eval $t$ $x$)(eval $u$ $x$))
 | $Rt \Rightarrow$ eval (eval $t$ $(Rt)$)
 | $Fut \Rightarrow \lambda^* x.Fut(S(K$eval)(eval $x$))
 | $DOtu \Rightarrow DO$ (eval $t$) $u$
 | $Ast \Rightarrow \lambda^* x.$eval $(Bxs)$ $t$
 | $Bus \Rightarrow$
   let $b$ = eval $(Bus)$ in
   $(\lambda^* x.$match $x$ with
     | $J \Rightarrow u$
     | $Ni \Rightarrow si$
     | $Htv \Rightarrow$ eval $(b\ t)(b\ v)$
     | $Bvt \Rightarrow B(b\ v)(b\ t)$
     | $Avt \Rightarrow A(b\ v)t)$
     | $O \Rightarrow O$
     | $yz \Rightarrow (b\ y)(b\ z))$
 | _ $\Rightarrow p$
```

Fig. 5. A Self-Interpreter

7 Conclusions

There has always been some tension between λ-calculus and combinatory logic. One one side, λ-calculus provides an account of functions that is intuitively appealing a function is a rule that is to be applied to its argument. On the other side, there are the conceptual and practical challenges associated with binding. Schönfinkel created combinatory logic, his calculus of functions, to capture the expressive power of predicate calculus without having to describe the domain of x in $\forall x P(x)$. In practice, the difficulties of variable renaming, of substituting under the λ and all the associated meta-theory create headaches for implementers and slow down performance.

Abstraction calculus combines the best of both worlds. One can freely buid abstractions that bind variables, using terms of the form Ast. And yet there is no meta-theory to consider: no need to compute free variables, rename variables, raise or lower de Bruijn indices, or perform implicit substitutions.

Further, abstraction calculus can be typed using little more than function types, product types and quantified types. The only novlty is in the kinding system, which is used to support three ways of typing applications, according to whether the function uses a fixed type context, changes the type context, or is higher order.

Creating the type system required many choices. Other choices may be more general, or aesthetically pleasing. Here, the emphasis has been on simplicity and familiarity. In particular, if a program has type T in context Γ then T is a simple type, built from variables and function types only (or perhaps some type constants for integers, etc), and Γ is a product of simple types: there is no polymorphism at this level. Future work may introduce such polymorphism, e.g. in the Hindley-Milner style. The operators themselves are polymorphic, however. Indeed, the full power of System F is required to type F. Further, the kind system, though easy to understand, looks a little ad hoc. Perhaps there is a pure type system to be found. As it stands, the system introduces a minimum of novelty to the type system, basically through its kinding rules, to achieve polymorphic analyses of simply-typed programs.

The resulting calculus is able to support typed pattern-matching functions that are able to select sub-terms of interest, or to update programs. Only a few examples are given here, and their typing has not been formally verified. nevertheless, the prospect of typed self-interpreters and other program analyses looks promising.

More generally, abstraction calculus may provide the core calculus for a typed programming language. Like **bondi**[3] it would support all of the usual programming styles, such as object-oriented, query-based and functional programming styles. In addition, it would be able to analyze its own programs, even during program execution, and this without any need for quotation or other meta-theory.

References

1. Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990. URL: citeseer.nj.nec.com/abadi91explicit.html.
2. H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984. revised edition.
3. bondi programming language. bondi.it.uts.edu.au/, 2014. URL: bondi.it.uts.edu.au/.
4. Matt Brown and Jens Palsberg. Typed self-evaluation via intensional type functions. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 415–428. ACM, 2017.
5. R. Hindley and J.P. Seldin. *Introduction to Combinators and Lambda-calculus*. Cambridge University Press, 1986.
6. Barry Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer, 2009.
7. Barry Jay. Self-quotation in a typed, intensional lambda-calculus. *Electronic Notes in Theoretical Computer Science*, 2017. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII).
8. Barry Jay. Recursive programs in normal form (short paper). In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '18, pages 67–73. ACM, 2018.
9. Barry Jay and Thomas Given-Wilson. A combinatory account of internal structure. *Journal of Symbolic Logic*, 76(3):807–826, 2011.
10. Barry Jay and Jens Palsberg. Typed self-interpretation by pattern matching. In *Proceedings of the 2011 ACM Sigplan International Conference on Functional Programming*, pages 247–58, 2011.
11. Delia Kesner. The theory of calculi with explicit substitutions revisited. In *International Workshop on Computer Science Logic*, pages 238–252. Springer, 2007.