Free Variable Types[1]

Edwin Westbrook
ewestbro@cse.wustl.edu

Department of Computer Science and Engineering,
Washington University in Saint Louis
Saint Louis, MO USA

Abstract

Higher-order abstract syntax (HOAS) has been shown to be useful in formal systems for reasoning about programming languages. How to recurse over HOAS representations, however, is a well-known problem. This is because, in most current systems, it is generally impossible to recurse on the body of a λ-abstraction without creating a fresh variable, which must itself be bound. This in turn makes it impossible, for instance, to extract first-order data from higher-order data. One previous approach to this problem has been to allow the creation of fresh variables that are not bound to pass to λ-abstractions as long as they can be shown not to escape their scope. Current systems cannot show these do not escape their scope without a runtime check. In this paper, we propose a novel type system that allows explicit reasoning about free variables. This type system can ensure these fresh variables do not escape their scopes without runtime checks. The usefulness of this approach is shown with a simple example.

1 INTRODUCTION

Higher-order abstract syntax (HOAS) has been shown to be useful in formal systems for reasoning about programming languages. It allows the user to represent variable bindings in the object language by using λ-abstractions in the meta language. This in turn means the user need not reason directly about name-related problems such as α-equivalence, as these are already handled by the meta language.

Meta language variables, however, are slippery things. Being variables, they can be replaced at any time via a substitution, and can thus not be used in any way in which substitution would be ill-typed. In addition, the meta language must ensure that its variables do not show up where they are not expected. This is because under the Curry-Howard isomorphism a free variable is a proof by assumption of the proposition for its type. Unexpected free variables then become unexpected assumptions, making the logic of a formal system inconsistent. Thus the meta language must be restrictive about the use of its variables and can only let the user use them when it can ensure they will not end up free outside of a λ that binds them.

[1]This work was partially supported by NSF grant CCF-0448275

For most current languages supporting HOAS, this restriction is conservative: variables are not allowed outside the scope of a λ. This is a burden because, in order to access the body of a λ, the user must pass that λ a term of the appropriate type. The only general way to make a dummy term of the appropriate type is to use a fresh variable, which, again, must be bound. This means the user cannot in general perform any operations on a λ outside the scope of some other λ of the same type. This in turn means it is impossible to write many simple functions on HOAS data, such as the function that computes the size of of a piece of data, as the size of the bodies of λs cannot escape to the top level.

In this paper we investigate a meta language called the λ^{FV}-calculus that relaxes this condition. The λ^{FV}-calculus allows the user to create fresh variables to pass to λ abstractions, using the construct $\nu x : T.M$. This construct creates a fresh variable x of type T for use in M and then returns M. (The name ν is pnemonic for a "new" variable. It comes from the ∇-calculus of Schürmann et. al. [12].) The fresh variable x can then be used to access the body of a λ in M.

In order for a program written in such a system to be safe, it must satisfy three conditions above and beyond standard type checking[1]:

1. *Variable Safety*: No variable created by a ν can escape the scope of that ν.

2. *Coverage*: Pattern matches must cover all possible forms of the argument they match against so that it is not possible for an input to "fall through" a pattern.[2] Coverage of a pattern match depends on the types of free variables in its argument, since it must consider all possible ways in which free variables could be used in that argument.

3. *Per-Variable Data*: Many functions on HOAS data require every free variable of some type to have data of some other type associated with it. (This condition is called "regular worlds" in Twelf[7].) An example of why per-variable data is necessary is given below.

To ensure these conditions, the λ^{FV}-calculus uses a novel type system that can reason about free variables. The λ^{FV}-calculus contains a free variable type modality that captures all of the variables of a given type that might occur free in the normal form of a term. This modality specifies a list, L, that must contain all such variables. Using this modality, the type system can ensure that a program meets the three conditions above. For variable safety, the type system can require that the variable, x, introduced by $\nu x : T.M$ not be in any lists specified by the type modalities of M. For coverage, the type system can ensure that all the ways free variables in the argument to a pattern match can be used in that argument are covered by some pattern. For per-variable data, the free variable type modality also

[1]We do not consider here a fourth condition, termination, that is required for programs to be considered as proofs since it is mostly orthogonal to the conditions given here

[2]If an input falls through a pattern in many implementations of ML, for instance, the program immediately terminates with an error condition.

allows these lists to contain additional data associated with the variables, so that a program can always look the variable up in L to find this data.

Note that this is a work in progress. Although the author does have an operational semantics for the calculus (which is not given here for space reasons), the proof of type safety is not yet complete. The author does not believe there should be any serious problems in completing it.

The rest of this document is organized as follows. In Section 2, we consider an example motivating the need to introduce new free variables. In Section 3, we introduce the free variable modality we will need for the λ^{FV}-calculus and show how it enforces safety conditions 1 and 3 above. In Section 4, we discuss how the free variable type modality of the λ^{FV}-calculus enforces coverage. In Section 5, we formalize these ideas and give the full static semantics. Finally, we look at related work in Section 6 and conclude and give future work in Section 7.

2 MOTIVATING EXAMPLE

In this Section, we consider the task of converting a lambda term from a HOAS representation to a first-order representation using deBruijn indices. These two representations are given by two different LF types, hterm and dbterm, with the following constructors:

$$\begin{array}{ll}
\mathsf{happ} : \mathsf{hterm} \Rightarrow \mathsf{hterm} \Rightarrow \mathsf{hterm} & \mathsf{dbvar} : \mathsf{nat} \Rightarrow \mathsf{dbterm} \\
\mathsf{hlam} : (\mathsf{hterm} \Rightarrow \mathsf{hterm}) \Rightarrow \mathsf{hterm} & \mathsf{dbapp} : \mathsf{dbterm} \Rightarrow \mathsf{dbterm} \Rightarrow \mathsf{dbterm} \\
 & \mathsf{dblam} : \mathsf{dbterm} \Rightarrow \mathsf{dbterm}
\end{array}$$

hterm uses HOAS to represent λ-abstraction, since hlam takes a meta-level function (and thus the object-level variables are represented by meta-level variables), whereas dbterm does not.

To convert an hterm to a dbterm, we need to recurse on the structure of the hterm, keeping track of the *level*, or number of variables bound by an hlam above the current point in the recursion. When we recurse on the body of an hlam, we will need to pass it a new variable, so we can access its body. In addition, we will need to associate that new variable with the level of its binding site. Then, whenever we reach a variable, we can look up the level of the variable's binding site and subtract this from the level of the variable, yielding the correct deBruijn index for the variable.

The code to do this is given in Figure 1. The **fun** creates the recursive function convert, which takes three arguments: $term$, the hterm to be converted; $level$, the current binding level; and l, a list of records associating hterm variables with their binding levels. The notation $\{l_1 : T_1, \ldots, l_n : T_n\}$ is a dependent record type, where T_i can use the labels l_j for $j < i$ as free variables. See [9] for more on dependently-typed records. The type List T is the type of lists of elements of type T, using the notation $x :: L$ for adding x to the front of L and nil for the empty list.

A **fun** is a different sort of function from a λ-abstraction, as the bodies of λ-abstractions are fully evaluated, whereas the bodies of a **fun** may have a pattern

```
fun convert ($L$ : List {$var$ : hterm, $varlev$ : nat}, $term$ : hterm, $level$ : nat) $\longrightarrow$
 case $term$ of
   happ $t_1$ $t_2$ [$t_1$ : hterm, $t_2$ : hterm] $\longrightarrow$
    dbapp (convert $L$ $t_1$ $level$) (convert $L$ $t_2$ $level$)
 | hlam $F$ [$F$ : hterm $\Rightarrow$ hterm] $\longrightarrow$
    dblam $\nu x$ : hterm.(convert [$var = x, varlev = level$] :: $L$ ($F$ $x$) (succ $level$))
 | $u$ [$u$ : var hterm] $\longrightarrow$
    dbvar (minus (lookup$_{var}$ $L$ $u$).$varlev$ $n$)
```

FIGURE 1. Converting HOAS to deBruijn λ-terms

match which cannot be evalutated until the arguments are known. This distinction between "representational" and "computational" functions is present in some form in many dependently-typed languages. The λ^{FV}-calculus distinguishes the two by giving representational functions the type $\Pi x : S.T$ (abbreviated $S \Rightarrow T$ when T does not depend on x) and and computational functions the type $\Pi^c u : S.T$ (abbreviated $S \Rightarrow^c T$). The λ^{FV}-calculus also distinguishes between two sorts of variable, the computational variables, which can be bound by computational functions, and the representational variables, which can be bound by λ and ν. Except in the example below (where verbosity yields meaning), we use x and y for representational variables and u for computational variables.

The key difference between the two sorts of variable is that the computational functions can perform computation with free representational variables, but all computational variables must be bound at the time of computation. The former should not be substituted for until after computation has finished, whereas the latter must be bound before computation can proceed. Note that pattern contexts can only have function variable bindings, because these variables must be bound when the body of the pattern executes.

The body of convert pattern matches *term* against three possible patterns: an application, a λ-abstraction, and a variable. The square brackets after each pattern give the types of the pattern variables in it. The first case, for applications, simply recurses on the subterms. The second case, for λ-abstractions, uses the ν construct to create a fresh variable x of type hterm. This case then recurses on $F\ x$ (which is effectively the body of F, as x is fresh), incrementing the level and associating x with the current level in L. The final case, for a variable, looks up the variable in L using the function lookup$_{var}$ and subtracts the number associated with it in *varlev* (which was the level of its binding site) from the current level. In the type of the variable u, we use the keyword **var** to indicate that it should only match a meta-level variable.[3]

We intuitively know that convert satisfies the three safety conditions mentioned in the Introduction. First, it is variable safe because any variable introduced by a

[3]Thus the variable matched against is not u itself, rather u is a variable whose contents are a variable.

ν will eventually be matched against in a recursive call, and convert will return a variable-free dbvar term. Second, it satisfies coverage because the only ways of constructing terms of type hterm are either using the constructors happ and hlam or using some free variable of type hterm. Note that convert does *not* satisfy coverage when *term* has free variables of types other than hterm that can be used to construct terms of type hterm, like for instance hterm $\Rightarrow$ hterm. This is because free variables of such types can be used to construct terms of type hterm, and the pattern match does not consider such cases. Finally, every free variable matched against has per-variable data in L, which is necessary for the call to lookup_{var} to succeed. This is only true under the assumption that *term* in the top-level call to convert has no free variables, i.e. that convert is only called on closed terms or is called recursively by itself. Given this assumption, though, whenever convert introduces a new free variable in the hlam case it always adds an entry for it into L.

3 FREE VARIABLE MODALITY

To ensure variable safety, coverage, and per-variable data, the λ^{FV}-calculus contains a type modality that limits the free variables of certain types of the normal form of a term. For expository purposes, we consider simpler versions of this modality in this section. We first consider a version that just ensures variable safety, and later modify it to also ensure per-variable data. The full modality that ensures coverage is introduced in Section 4.

The simplest version of the modality is $\Box_S(L)$. For types S and T and list L of type List S, a term of type $\Box_S(L)T$ is a term that otherwise has type T but for which all the free variables of type S in its normal form are elements of the list L. In particular, the modality $\Box_S(\mathsf{nil})$ ensures the normal form of a term has no free variables of type S. We use $\Box$ here because of the relation between free variables and modal logic: if M has type $\Box_S(L)T$, then for all possible substitutions of type S for variables in L (read: all possible worlds), M will have type T. (For more on the connection to modal logic see Nanevski et. al. [6, 5].)

In general, terms can have more than one type of free variable, so these modalities can be stacked. For instance, a term with free variables x and y of types S_1 and S_2 respectively might have type $\Box_{S_1}(x :: \mathsf{nil})\Box_{S_2}(y :: \mathsf{nil})T$. To abbreviate multiple modalities, we use the notation ϕT, where ϕ can be any list of $\Box$ modalities. We will generally assume with this notation that ϕ contains all the modalities of the type, and T contains none (at least at the top level). Since the order of the free variables does not matter, we identify all ϕ up to permutations of the modalities and the lists.

Modalities ϕ limit only the free variables in the normal form of a term. This allows terms that might temporarily use a variable not mentioned in the modality as long as the variable is not in the normal form. This is a conservative approximation: if $M : \Box_S(L)T$, the normal form of M need not actually have all the variables listed in L free. This modality simply states that the normal form of M *can* have these variables free. More importantly, the modalities on a type list the *only* variables the normal form can have free that have the types mentioned in the modalities. Thus T is a supertype of $\Box_S(L)T$, as T does not impose any requirements on the free variables of a term, while $\Box_S(L)T$ does. Also $\Box_S(L)T$ is a subtype of $\Box_S(x :: L)T$, as the latter allows more variables to be free.

We also do not allow modalities inside modalities. For instance, the type $\Box_{\Box_S(L_1)T}(L_2)U$ is not well-formed. This is because we will always allow the substitution of any term M of type T for x in any term N of type U, no matter what the free variables of M are. Of course the modality of the resulting term will reflect the fact that it could have the variables of M in it as well as any N had. The type $\Box_{\Box_S(L_1)T}(L_2)U$ would limit the terms that could be substituted for variables in L_2 depending on the free variables of their normal forms. By the same token, it does not make sense to have a λ-abstraction with type $(\Box_S(L)T) \Rightarrow U$. On the other hand, if a λ-abstraction has a body whose normal form has some variables free in it, the normal form of that λ-abstraction itself has those variables free in it, so the types $T \Rightarrow (\Box_S(L)U)$ and $\Box_S(L)(T \Rightarrow U)$ are equivalent.

In contrast, computational functions like $\mathsf{convert}$ can affect the free variables of a term. In the case of $\mathsf{convert}$, term can have free variables of type hterm, but the returned value will not have these free. Because of this, the types of computational functions can mention modalities in both their arguments and their bodies. For instance, $\mathsf{convert}$ has type

$$\begin{array}{c}\Pi^c L : \{\mathit{var} : \mathsf{hterm}, \mathit{varlev} : \mathsf{nat}\}.\\ \Pi^c \mathit{term} : \Box_{\mathsf{hterm}}(\mathsf{proj}_{\mathsf{var}}\ L)\mathsf{hterm}.\\ \Pi^c \mathit{level} : \mathsf{nat}.\Box_{\mathsf{hterm}}(\mathsf{nil})\end{array}$$

where $\mathsf{proj}_{\mathsf{var}}\ L$ is the function that returns a list of var fields of the records in L.

Although the $\Box_S(L)$ modality is sufficient to ensure variable safety - since the typing rule for $\nu x : S.M$ can simply require M to have type $\Box_S(L)T$ for any L not containing x - it is not sufficient for our needs for three reasons. First, because of the dependencies that arise in HOAS functions, it is useful to be able to reason not only about all variables of one specific type, but also about all variables of types that are substitution instances of a general type. For instance, consider a type, sterm, for the terms of the simply typed lambda calculus. Such a type might be indexed by stype, i.e. $\mathsf{sterm}\ A$ would be a type for every $\mathsf{stype}\ A$. In this scenario, we might be interested in all free variables of type $\mathsf{sterm}\ x$ for any x. The second reason the current modality will not be sufficient is that because of per-variable data we will require some easy method of associating data with each variable. Finally, it will be useful below to specify multiple lists that together contain all the free variables of specified types.

We thus expand the $\Box$ modality to

$$\Box_{\Gamma \vdash S}(l_1;L_1|\ldots|l_n;L_n)$$

where Γ is a context (list of variables and their types), each l_i is a record label, and each L_i is a list of type $\mathsf{List}\ RT_i$ for some record type RT_i that contains the label

l_i. This modality specifies that every free variable whose type is a substitution instance of S (substituting for variables in Γ) occurs, for some i, in field l_i of some record R occuring in the list L_i. The vertical bars $|$ in the modality indicate the disjunctive nature of the list/label combinations. For example, if a function were to recurse on terms of type sterm with free variables of type $\mathsf{sterm}\ x$ for any stype, but had data of type nat associated with variables of type $\mathsf{sterm}\ A$, it would use the modality

$$\Box_{x:\mathsf{stype} \vdash \mathsf{sterm}\ x}(trm; L_1 | trm; L_2)$$

where L_1 would have type $\mathsf{List}\ \{tp : \mathsf{stype}, trm : \mathsf{sterm}\ tp\}$ and L_2 would have type $\mathsf{List}\ \{tp : \mathsf{stype}, trm : \mathsf{sterm}\ tp, data : \mathsf{nat}\}$. To specify that all free variables of type $\mathsf{sterm}\ A$ had data of type nat, we would have to nest the modalities:

$$\Box_{x:\mathsf{stype} \vdash \mathsf{sterm}\ x}(trm; L_1)\Box_{\cdot \vdash \mathsf{sterm}\ A}(trm; L_2)$$

so that all variables of type $\mathsf{sterm}\ A$ would also be in L_2. The first modality here would then also require that all variables of type $\mathsf{sterm}\ x$ be in L_1, including those of type $\mathsf{sterm}\ A$ in L_2. To allow variables of type $\mathsf{sterm}\ A$ to not appear in L_1, we would have to use the modality

$$\Box_{x:\mathsf{stype} \vdash \mathsf{sterm}\ x}(trm; L_1 | trm; L_2)\Box_{\cdot \vdash \mathsf{sterm}\ A}(trm; L_2)$$

to allow these variables which appear in L_2 to satisfy the first modality. Each modality must be satisfied irrespective of what other modalities are present. The full list of modalities can be considered as a conjunction of the individual modalities, while the set of lists inside a modality is a disjunction of the individual lists.

This expanded modality requires a new pattern matching construct to match against the data associated with a variable. The syntax

$$\hat{\ }(l; L; [\ldots, l = u, \ldots])$$

matches against a variable at label l in list L. The record given as the third parameter to ˆ matches against the record in L containing the free variable, which will be substituted for u when this pattern matches it. Having the whole record present in the pattern allows case splitting in pattern matches on the per-variable data. Note that the field l in this record must always contain a variable to hold the free variable matched against. (It is a variable that names a variable.) Since u matches a free variable, it can be assumed to satisfy the modalities this free variable would match, namely $\Box_S(u :: \mathsf{nil})$ (where S is the type of u) and any modalities that have types different than S.

Using the refined $\Box$ modality and the ˆ construct, the variable case for convert can be rewritten

$$\hat{\ }(var; L; [var = u, varlev = vl])\ [u : \mathsf{hterm}, vl : \mathsf{nat}] \to$$
$$\mathsf{dbvar}\ (\mathsf{minus}\ vl\ n)$$

to make explicit the association between the variable matched against and L. The resulting type of convert is

$$\begin{array}{c}\Pi^c L : \{var : \mathsf{hterm}, varlev : \mathsf{nat}\}.\\ \Pi^c term : \Box_{\cdot\vdash\mathsf{hterm}}(var; L)\mathsf{hterm}.\\ \Pi^c level : \mathsf{nat}.\Box_{\cdot\vdash\mathsf{hterm}}(\mathsf{nil}).\end{array}$$

Note that *level* cannot contain any free varables of type nat, as the minus function would not work in this case.

4 COVERAGE

In this section, we examine issues of coverage that come up when free variables are allowed. As mentioned in Section 1, coverage is the property that all possible forms of the argument of a pattern match are matched by some pattern. If a pattern match does not satisfy coverage, then the program containing it is in some sense not correct, as certain input data could cause the pattern match to fall through all of its cases and cause a run-time error. Since one of the points of using dependent types with HOAS is to have verified programs, this is an important property.

We give an informal description of coverage checking here. For more details, see [10].

Coverage is checked by case splitting on the computational variables in the argument of a pattern match. More specifically, if the argument to a pattern match does not already match one of the patterns (which is hardly the norm), the computational variables it contains are *split* into subcases, one for each constant or (λ-bound) variable that could possibly be used to construct a term of the type of that computational variable. For instance, checking coverage of convert requires splitting the variable *term* into cases for happ, hlam, and for free variables of type hterm. Since these are the only possible ways of constructing a term of type hterm, *term* must match one of them, and so if they are all covered by the pattern match (which they are in the case of convert), so is *term*. If checking coverage of a pattern match requires splitting some computational variable u of type T, the pattern match is said to split u over T. Either u or T can be elided, in which case the pattern match is said to split u or split over T.

Splitting over function and record types reduces to splitting over base types. Splitting over the type $\Pi x : S.T$ requires splitting over T with the extra variable x of type S, since a λ-abstraction can use the variable it introduces in its body. Splitting over a record type is straightforward, as only records can inhabit record types (using extensionality of records), and splitting over computational function types $\Pi^c u : S.T$ is not allowed, as there are (in almost all cases) infinitely many possible functions of type $\Pi^c u : S.T$.

How a variable u is split depends on the free variable modality of u. For example, when splitting *term* over the type hterm in convert, free variables of type hterm must be considered, as *term* could possibly be such a free variable, and the pattern match must consider this case. The problem is in fact more general, as splitting over any base type T must consider all free variables that could be used as the head of a term of type T, not just those of type T. (Splitting over record and function types reduces to splitting over base types by the previous paragraph.) The free variables that must be considered when splitting over base type T include: free variables of any base type T' unifiable with T; free variables of any function type $\Pi x_1 : U_1 . \dots \Pi x_m : U_m . T'$ where some substitution for x_1 through x_m in T' yields a substitution instance for computational variables of T; and free variables of any record type, the type of one of whose fields is a substitution instance of T.

Any of these types that could be used as the head of a term of type T are said to *generate* T. A computational variable is itself said to generate T if its type generates T. Formally, that S generates T is captured by the judgment $\Gamma \vdash S \rhd T$, where Γ is a context typing the variables occuring in S and T. It makes precise the intuitions given in the previous paragraph. We do not formalize this here for space reasons.

Since there are infinitely many types that generate any T and pattern matches are finite, any pattern match (except those with a catch-all pattern that matches anything) cannot cover all possible cases of free variables that could occur in its argument. Specifically, if a pattern match splits u over T, then the free variables that generate T in any argument passed in for u must have one of the types considered by the splitting. Otherwise, u could have a form not considered by the pattern match. To ensure this property, we modify the meaning of the free variable modality of the previous section. Its syntax remains the same, but a term with the modality

$$\Box_{\Gamma \vdash S}(l_1; L_1 | \dots | l_n; L_n)$$

must now have all of its free variables that generate S be listed in one of the L_i in field l_i. The other change to the modality is that S must be a base type. Given this refined modality it is then safe to only consider free variables listed in the L_i when splitting over any type that is a substitution instance of S.

The full free variable modality given above can be considered an explicit formulation of the regular worlds of Twelf [7]. Regular worlds specify what types of variable may be free when a function is used. They also specify that each variable of certain types have data associated with them by using blocks of variables that must come together. These blocks can be modelled by the lists of records in our free variable type modality. The difference here is that the type modality is now an explicit part of the type of the function, instead of it being externally specified and externally checked. In addition, the free variable modality allows finer control over which specific variables end up free.

5 STATIC SEMANTICS

In this section we make concrete the above ideas by giving a static semantics for the λ^{FV}-calculus. We start with a grammar for the elements of the language. This

$$
\begin{array}{lll}
\text{Kinds} & K ::= & \Pi x : T.K \parallel \mathsf{type} \\
\text{Types} & T ::= & \Pi x : T_1.T_2 \parallel \Pi^c x : \phi_1 T_1.\phi_2 T_2 \parallel a\, M_1 \ldots M_n \parallel RT \parallel \mathsf{List}\; T \\
\text{Record Types} & RT ::= & \{l : T, RT\} \parallel \{\} \\
\text{Objects} & M ::= & x \parallel c \parallel M_1\, M_2 \parallel \lambda x : T.M \parallel u \parallel \nu x : T.M \parallel \mathsf{nil} \parallel M_1 :: M_2 \\
 & & \parallel R \parallel M.l \parallel \textbf{fun}\, u\, (u_1 : \phi_1 T_1, \ldots, u_n : \phi_n T_n) \rightarrow M \\
 & & \parallel \textbf{case}\, M\, \textbf{of}\, P_1\, [\Gamma] \rightarrow M_1 | \ldots | P_n\, [\Gamma] \rightarrow M_n \\
\text{Records} & R ::= & [l = M, R] \parallel [] \\
\text{Record Patterns} & RP ::= & [l = P, RP] \parallel [] \\
\text{Patterns} & P ::= & x \parallel c \parallel P_1\, P_2 \parallel \lambda x : T.P \parallel u \parallel \mathsf{nil} \parallel P_1 :: P_2 \\
 & & \parallel RP \parallel P.l \parallel \Upsilon(l; M; RP) \\
\text{Signatures} & \Sigma ::= & \Sigma, c : T \parallel \Sigma, a : K \parallel \cdot \\
\text{Contexts} & \Gamma ::= & \Gamma, x : T \parallel \Gamma, u : \phi T \parallel \cdot \\
\text{Modalities} & \phi ::= & \Box_{\Gamma \vdash T}(LS)\phi \parallel \cdot \\
\text{List Specs.} & LS ::= & l; M | LS \parallel \cdot
\end{array}
$$

FIGURE 2. Grammar for the λ^{FV}-calculus

is given in Figure 2. Most of these constructs are either standard or we have seen them above. The first four object-level constructs along with the representational function types and the type constants are exactly LF [3]. The records, record types, and record projections are standard from dependently-typed records (see [9]). The List type, with its constructors nil and ::, is also straightforward. The language contains in addition computational functions (**fun**), computational variables, the ν construct, and **case** expressions.

The Figure also separates out the class of patterns. These are similar to the objects, but cannot contain computational functions or **case** expressions. They can contain expressions $\Upsilon(l; L; RP)$ to match variables, as discussed in Section 3. For preciseness, the forms of records allowed in patterns are also separated from normal records in the Figure.

In order to be used in a typing judgment, all signatures, contexts, and modalities must be well-formed, written $\vdash \Sigma$, $\vdash \Gamma$, and $\Gamma \vdash \phi$. These judgments will be implicit below, and all signatures, contexts, and modalities will be assumed to be well-formed. The well-formedness of signatures will in fact be implicit in the rules for that of contexts and modalities, and the well-formedness of contexts will also be assumed in the rules for that of modalities. The rules for $\vdash \Sigma$ are:

$$
\frac{}{\vdash \cdot} \qquad \frac{\Sigma; \cdot \vdash K}{\vdash \Sigma, a : K} \qquad \frac{\Sigma; \cdot \vdash T : \mathsf{type} \quad T \text{ does not contain } \Pi^c}{\vdash \Sigma, c : T}
$$

In addition to the kinds and types of constants needing to be well-formed, the types of constants are also not allowed to contain computational abstractions. This is because computational abstraction types can contain modalities which change the variables considered to be bound, but constants themselves cannot change which variables are bound.

The rules for $\vdash \Gamma$ are as follows:

$$\frac{}{\vdash \cdot} \qquad \frac{\Gamma \vdash T : \mathsf{type} \quad T \text{ does not contain } \Pi^c}{\vdash \Gamma, x : T} \qquad \frac{\Gamma \vdash T : \phi\mathsf{type} \quad \Gamma \vdash \phi}{\vdash \Gamma, u : \phi T}$$

A representational variable can also not contain computational abstraction types, for the same reason as for constants. Computational variables can contain computational abstraction types, since computational variables will always be bound to a value when evaluated. Computational variables are also listed with a modality in the context. This modality must be well-formed, and the type of the variable must also satisfy the modality.

The rules for $\Gamma \vdash \phi$ are as follows:

$$\frac{}{\Gamma \vdash \cdot} \qquad \frac{\Gamma, \Gamma' \vdash T : \mathsf{type} \quad \forall i(\Gamma \vdash L_i : \mathsf{List}\, RT_i) \quad \forall i(l_i \in \text{labels}(RT_i)) \quad \Gamma \vdash \phi}{\Gamma \vdash \Box_{\Gamma' \vdash T}(l_1; L_1 | \ldots | l_n; L_n)\phi}$$

These rules require the type specifier T in a modality to always be well-formed. They also require every L_i to be a valid list of records, where the type of these records must contain the label l_i.

For space reasons, we do not give the rules for well-formed kinds or well-kinded types. These are similar to their standard versions, except how they propagate, which is done in ways similar to the object-level typing rules.

The rules for typing objects are given in Figure 3. These are given by the judgment $\Gamma \vdash M : \phi T$, where Γ is the context typing the variables of M, T is the type of M, and ϕ limits the free variables of M. If $\Gamma \vdash M : \phi T$ holds for some Γ and T, M is said to *satisfy* ϕ. Many of the rules are similar to their standard forms except for the addition of ϕ. Constants, empty records, and empty lists do not have any free variables, and can thus satisfy any modality ϕ. Representational applications, non-empty records and lists, and record projections satisfy a modality only if all subterms satisfy the modality.

In Section 3, we argued that a term satisfying a modality ϕ should also satisfy any modality ϕ' where ϕ' allows for at least as many free variables. This is because ϕ' is a more conservative approximation than ϕ. In this case we say ϕ' is at least as permissive as ϕ, written $\phi \precsim \phi'$. If ϕ and ϕ' are just as permissive as each other, we write $\phi \sim \phi'$. The rule in Figure 3 allows any M satisfying ϕ to be typed with a more permissive modality ϕ'.

The rules for permissiveness are given in Figure 4 in terms of permissiveness rules for the list specifiers of modalities. These rules define $\sim$ as an equivalence relation and $\precsim$ as a transitive ordering. List specifiers are just as permissive when lists are permuted, when free variables of the lists are permuted, and when the empty list is added. Modalities are in addition just as permissive when their order is changed. A list specifier is more permissive than another when it adds a list specifier, adds a variable to a list, or makes the type of a modality more specific (and thus it specifies the freeness of less variables).

Returning to Figure 3, there are three rules for typing representational variables x. The first finds the type of x in Γ, but does not ascribe it any modality. The other

$$\frac{c : \phi T \in \Sigma}{\Gamma \vdash c : T} \qquad \frac{u : \phi T \in \Gamma}{\Gamma \vdash u : \phi T} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash M : \phi' T \quad \phi' \precsim \phi}{\Gamma \vdash M : \phi T}$$

$$\frac{\Gamma \vdash x : \phi T \quad \Gamma, \Gamma' \vdash T \not\triangleright T'}{\Gamma \vdash x : \Box_{\Gamma' \vdash T'}(l_1; M_1 | \ldots | l_n; M_n)\phi T} \qquad \frac{\Gamma \vdash x : \phi T \quad \Gamma, \Gamma' \vdash T \triangleright T'}{\Gamma \vdash x : \Box_{\Gamma' \vdash T'}(l; [\ldots, l = x, \ldots] :: M)\phi T}$$

$$\frac{\Gamma \vdash M_1 : \phi(\Pi x : T_1.T_2) \quad \Gamma \vdash M_2 : \phi T_1}{\Gamma \vdash M_1\ M_2 : \phi[M_2/x]T_2} \qquad \frac{\Gamma, x : T_1 \vdash M : \phi T_2}{\Gamma \vdash \lambda x : T_1.M : \phi^x(\Pi x : T_1.T_2)}$$

$$\frac{\Gamma, x : T_1 \vdash M : \phi T_2 \quad \Gamma \vdash \phi \quad \Gamma \vdash T_2 : \mathsf{type}}{\Gamma \vdash \nu x : T_1.M : \phi T_2} \qquad \frac{\Gamma \vdash M : \phi RT \quad \mathbf{RTSel}_l(M, RT) = T}{\Gamma \vdash M.l : \phi T}$$

$$\frac{}{\Gamma \vdash [] : \phi\{\}} \qquad \frac{\Gamma \vdash M : \phi T \quad \Gamma, l : T \vdash RT : \phi\mathsf{type} \quad \Gamma \vdash R : \phi[M/l]RT}{\Gamma \vdash [l = M, R] : \phi\{l : T, RT\}}$$

$$\frac{\Gamma \vdash T : \mathsf{type}}{\Gamma \vdash \mathsf{nil} : \phi(\mathsf{List}\ T)} \qquad \frac{\Gamma \vdash M_1 : \phi T \quad \Gamma \vdash M_2 : \phi(\mathsf{List}\ T)}{\Gamma \vdash M_1 :: M_2 : \phi(\mathsf{List}\ T)}$$

$$\frac{\Gamma \vdash M_1 : \phi(\Pi^c x : \phi_1 T_1.\phi_2 T_2) \quad \Gamma \vdash M_2 : \phi_1 T_1}{\Gamma \vdash M_1\ M_2 : \phi\phi_2[M_2/x]T_2}$$

$$\frac{\Gamma, u : \Pi^c u_1 : \phi_1 T_1 \ldots \Pi^c u_n : \phi_n T_n.\phi T, u_1 : \phi_1 T_1, \ldots, u_n : \phi_n T_n \vdash M : \phi T}{\Gamma \vdash \mathbf{fun}\ u\ (u_1 : \phi_1 T_1, \ldots, u_n : \phi_n T_n) \rightarrow M : \Pi^c u_1 : \phi_1 T_1 \ldots \Pi^c u_n : \phi_n T_n.\phi T}$$

$$\frac{\begin{array}{l} \Gamma \vdash L : \phi'(\mathsf{List}\ RT) \qquad\qquad\qquad\qquad \Gamma \vdash u : \phi T \\ \Gamma \vdash [l_1 = P_1, \ldots, l_i = P_i, l = u, l_{i+1} = P_{i+1}, \ldots, l_n = P_n] : RT \end{array}}{\Gamma \vdash \Upsilon(l; L; [l_1 = P_1, \ldots, l_i = P_i, l = u, l_{i+1} = P_{i+1}, \ldots, l_n = P_n]) : \phi T}$$

$$\frac{\begin{array}{c} \Gamma \vdash M : \phi' T' \qquad\qquad \forall i(\Gamma, \Gamma_i \vdash P_i : \phi_i T') \\ \forall i(\Gamma, \Gamma_i \vdash M_i : \phi T) \quad \forall i \forall x \in \mathrm{Dom}(\Gamma_i)(\vdash \mathbf{strict}(x, P_i, \Gamma)) \\ \vdash (\Gamma, \Gamma_1 \vdash P_1; \ldots; \Gamma, \Gamma_n \vdash P_n)\ \mathbf{covers}\ (\Gamma \vdash M) \end{array}}{\Gamma \vdash \mathbf{case}\ M\ \mathbf{of}\ P_1\ [\Gamma_1] \rightarrow M_1 | \ldots | P_n\ [\Gamma_n] \rightarrow M_n : \phi T}$$

FIGURE 3. Object-level typing rules for the λ^{FV}-calculus

$$\frac{}{\phi \sim \phi} \qquad \frac{\phi' \sim \phi}{\phi \sim \phi'} \qquad \frac{\phi \sim \phi' \quad \phi' \sim \phi''}{\phi \sim \phi''} \qquad \frac{\phi \sim \phi'}{\phi\phi'' \sim \phi'\phi''} \qquad \frac{}{\phi\phi' \sim \phi'\phi}$$

$$\frac{LS \sim LS'}{\Box_{\Gamma\vdash T}(LS) \sim \Box_{\Gamma\vdash T}(LS')} \qquad \frac{LS \sim LS'}{LS|LS'' \sim LS'|LS''} \qquad \frac{}{l;L|l';L'|LS \sim l';L'|l;L|LS}$$

$$\frac{}{l,\mathsf{nil} \sim \cdot} \qquad \frac{l;L \sim l;L'}{l;M :: L \sim l;M :: L'} \qquad \frac{}{l;M :: M' :: L \sim l;M' :: M :: L}$$

$$\frac{\phi \sim \phi'}{\phi \precsim \phi'} \qquad \frac{\phi \precsim \phi' \quad \phi' \precsim \phi''}{\phi \precsim \phi''} \qquad \frac{\phi \precsim \phi'}{\phi\phi'' \precsim \phi'\phi''} \qquad \frac{LS \precsim LS'}{\Box_{\Gamma,u:\phi T,\Gamma'\vdash T}(LS) \precsim \Box_{\Gamma,\Gamma'\vdash[M/u]T}(LS')}$$

$$\frac{}{l;L \precsim l;R :: L} \qquad \frac{}{\cdot \precsim l;L} \qquad \frac{R.l = x \quad R'.l' = x}{l;R :: L|l';L' \sim l;L|l';R' :: L'}$$

FIGURE 4. Context Permissiveness

two check whether it satisfies each individual modality in ϕ. Whether x satisfies $\Box_{\Gamma'\vdash T'}(l_1;L_1|\ldots|l_n;L_n)$ depends on whether its type T generates T'. If not (given by the judgment $\Gamma \vdash T \not\triangleright T'$ of Section 4), x will satisfy any such modality. Otherwise, x must be in field l_i of a record in L_i for some i. The rule for this specifies that x only satisfies modalities with one list that has x in the correct field at the head of that list. Using permissiveness, these modalities can be weakened so that x satisfies the full modality.

Since a variable cannot escape its λ binding, the typing rule for $\lambda x : T.M$ can safely remove x from the modality used to type M. This is the purpose of ϕ^x. Intuitively, this purges the variable x from ϕ. Formally, ϕ^x is defined as follows:

$$\begin{aligned}
(\phi\phi')^x &= \phi^x\phi'^x \\
(\Box_{\Gamma\vdash T}(l_1;L_1|\ldots|l_n;L_n))^x &= \Box_{\Gamma\vdash T}((l_1;L_1)^x|\ldots|(l_n;L_n)^x) \\
(l;\mathsf{nil})^x &= l;\mathsf{nil} \\
(l;R :: L)^x &= l;(R\backslash x) :: \mathsf{nil}|(l;L)^x \text{ if } R.l \neq x \\
(l;u)^x &= l;u \\
(l;y)^x &= l;y \\
(l;[l_1 = M_=,\ldots,l = x,\ldots] :: L)^x &= (l;L)^x
\end{aligned}$$

This judgment removes x from any list in ϕ that mentions it as a free variable. It is possible that a list specifier ends with a variable. Since purging is only used at a λ-abstraction binding x, it is safe to leave such variables as they are, since their scope is greater than that of x and therefore cannot have x free. It is also possible that x is free in per-variable data for some other variable y. In this case, y is still free, but any per-variable data with x free is no longer valid. This is the purpose of $R\backslash x$: it removes all fields of R with x free. If there are none, then R remains intact, and permissiveness can put all such R that are separated by the third to last clause above back into the same list after purging has finished.

The rule for ν uses a simpler method of ensuring that x is not in the modality ϕ. It requires that the modality be well-formed in the original context Γ before x is added. To ensure x does not end up free in the type of a ν expression, this type must also be well-kinded without x.

The rule for typing a projection uses the function **RTSel**, defined as follows:

$$\begin{array}{rcl} \mathbf{RTSel}_l(M, \{l : T, RT\}) & = & T \\ \mathbf{RTSel}_l(M, \{l' : T, RT\}) & = & \mathbf{RTSel}_l(M, [M.l'/l']RT) \text{ for } l \neq l' \end{array}$$

When projecting a field l of a record, the projections of all earlier fields of that record are substituted for the fields in the type associated with l in the record type. For more on dependently-typed records, see [9], and see [13] for more on **RTSel**.

Typing computational applications requires the argument to satisfy the input modality of the function. This modality is then replaced with the combination of the output modality and the modality of the whole function in the modality of the application. This is because a modality on a function type is the same as a modality on the output of the function type.

Computational functions have a built-in fixpoint, so typing them requires assuming the name of the function as well as the input variables have the appropriate types and showing the body yields the appropriate type. The modality satisfied by the body of the function is then used as the modality of the whole function.

The rule for typing $\hat{}(l; M; RP)$ makes three requirements. First, RP must have a computational variable in field l to match the variable matched by the whole form. Second, RP must have some valid record type RT. Third, M must be a well-typed list whose elements have the same record type RT, i.e. it must have type $\mathsf{List}\, RT$.

The most complicated rule in the Figure is that for **case** expressions. The first two premises require that the patterns all have the same type as the argument. The modalities on these are not important. The third premise ensures that the bodies of the pattern match all have the appropriate return type. All bodies must also satisfy the output modality, as they will be returned for the value of the **case** expression. The fourth clause tests for strictness of the patterns, a syntactic condition that ensures higher-order pattern matching is decidable. (For more on strict patterns, see Schürmann and Pfenning [10].) Finally, the last clause ensures that the case expression passes coverage checking, as described in Section 4. This judgment ensures that splitting of a computational variable is only done over a type matched by one of its modalities.

This type system has the following useful properties. These can be proved by straightforward induction on the size of the judgments.

Theorem 1 ***Free Variables***
If $\Gamma \vdash M : \phi T$ and M is a normal form, then the free representational variables of M are all listed in modalities in ϕ.

Theorem 2 ***Representational Substitution***
If $\Gamma \vdash M : \phi T$ and $\Gamma \vdash N : (\phi)^x T'$, then $\Gamma \vdash [N/x]M : (\phi)^x T$.

6 RELATED WORK

Dependently-typed programming languages in general are the subject of active research. (See e.g. [13, 1, 4]) Although some older languages (e.g. [7]) support HOAS, there has recently been a push to overcome some of the limitations on recursion over HOAS representations outlined in the introduction. One example of this direction of research is the ∇-calculus of Schürmann et. al. [12] and its refinement to the language $\lambda^{\ominus\Pi}$ [11]. This is work upon which the current calculus is based, as it introduced the ν operator present here. These languages were not as powerful as the one given here, however, since showing there that a ν-bound variable does not escape its scope generally involves a runtime check.

Other work on programming with HOAS representations includes the following. In [2], HOAS is combined with a first-order representation in such a way to allow some level of recursion over HOAS representations. In [5], the type system allows specification of the free variables of a term, and [6] allows this indirectly by reasoning about contexts. Yet another approach has been to use nominal logic [8] to reason about variable binding constructs.

7 CONCLUSION AND FUTURE WORK

We have seen a language with a built-in capability for reasoning about free variables. It allows reasoning not only about the free variables of HOAS representations, but more importantly about how computational functions manipulate the free variables of these representations. It also allows reasoning about the free variables in specific arguments, yielding a fine-grained specification of what variables end up free in what results.

We have also seen how reasoning about free variables allows us to express more functions over HOAS representations of data. This is accomplished by using the ν construct to create fresh variables to pass into λ-abstractions. The type system can then ensure that a fresh variable does not escape the scope of the ν that created it.

For future work, a proof of type safety should be completed for the λ^{FV}-calculus. As mentioned in the introduction, the author does not anticipate this should be a serious challenge, but it has not yet been completed. The current system also must be "stress-tested" with more programs written in it. This would test its expressiveness; it is unclear exactly exactly how expressive the λ^{FV}-calculus is compared with other dependently-typed languages supporting HOAS.

The system given here is also complicated. It requires lists and dependently-typed records to express properties about free variables. This suggests an interesting avenue for future work in trying to simplify it in some manner.

Acknowledgments: The author would like to thank his advisor, Aaron Stump, along with Ian Wehrman and Charles Comstock for discussions and errata on the paper. The author would also like to thank Adam Poswolsky for pointing out major flaws in earlier versions of this work.

REFERENCES

[1] C. Chen and H. Xi. Combining Programming with Theorem Proving. In *Proceedings of the 10th International Conference on Functional Programming (ICFP05)*, Tallinn, Estonia, September 2005.

[2] K. Donnelly and H. Xi. Combining higher-order abstract syntax with first-order abstract syntax in ats. In *Workshop on Mechanized Reasoning about Languages with Variable Binding (MERλIN'05)*, Tallinn, Estonia, September 2005.

[3] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.

[4] C. McBride and J. McKinna. The View from the Left. *Journal of Functional Programming*, 14(1), 2004.

[5] A. Nanevski and F. Pfenning. Meta-programming with names and necessity. *Journal of Functional Programming*, 2005. To appear.

[6] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. submitted, 2005.

[7] F. Pfenning and C. Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.

[8] A. M. Pitts. Nominal logic: A first order theory of names and binding. In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001. Proceedings*, volume 2215 of *Lecture Notes in Computer Science*, pages 219–242. Springer-Verlag, Berlin, 2001.

[9] R. Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002.

[10] C. Schürmann and F. Pfenning. A Coverage Checking Algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *LNCS*, pages 120–135. Springer-Verlag, 2003.

[11] C. Schürmann and A. Poswolsky. A temporal-logic approach to programming with dependent types and higher-order encodings. unpublished manuscript, 2005.

[12] C. Schürmann, A. Poswolsky, and J. Sarnat. The ∇-Calculus. Functional Programming with Higher-Order Encodings. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications*, volume 3461 of *Lecture Notes in Computer Science*, pages 339–353, Long Beach, CA, January 2005. Springer-Verlag.

[13] E. Westbrook, A. Stump, and I. Wehrman. A Language-based Approach to Functionally Correct Imperative Programming. In *Proceedings of the 10th International Conference on Functional Programming (ICFP05)*, 2005.