Abstract

Encodings based on higher-order abstract syntax represent the variables of an object-language as the variables of a meta-language. Such encodings allow for the reuse of $\alpha$-conversion, substitution and hypothetical judgments already defined in the meta-language and thus often lead to simple and natural formalization. However, it is also well-known that there are some inherent difficulties with higher-order abstract syntax in supporting recursive definitions.

We demonstrate a novel approach to explicitly combining higher-order abstract syntax with first-order abstract syntax that makes use of a (restricted) form of dependent types. With this combination, we can readily define recursive functions over first-order abstract syntax while ensuring the correctness of these functions through higher-order abstract syntax. We present an implementation of substitution and a verified evaluator for pure untyped call-by-value $\lambda$-calculus.

Categories and Subject Descriptors D.3 [Software]: Programming Languages

General Terms Languages, Verification

Keywords ATS, ATS/LF, Higher-Order Abstract Syntax, Dependent Types, Theorem Proving

1. Introduction

We have recently seen a great deal of interest in the development of mechanically checked formal systems for specifying and verifying programming languages and their properties. One of the primary technical challenges facing such systems lies in the treatment of variable binding constructs. Formalizations which make use of first-order representations, where variables are explicitly represented as closed syntax, are often burdened by ubiquitous side conditions on variable occurrences, and the need to explicitly define and use $\alpha$-conversion, substitution, and hypothetical and parametric judgments in statements of object-level rules and properties.

The approach of higher-order abstract syntax (HOAS) [19], which goes back to Church [6], makes use of meta-level variables to represent object-level variables. In such representations, all variable-binding constructs of the object-languages are encoded with meta-level $\lambda$-abstractions. This approach enables us to reuse meta-level $\alpha$-conversion, substitution, and hypothetical and parametric judgments and thus eliminates the need for complex side-conditions in rules. However, there seems to be an inherent problem with HOAS that we encounter immediately when attempting to support recursion over HOAS.

There have been primarily two responses to this problem. One is to give up HOAS and instead develop enhanced first-order representations, and the other is to employ some mechanism to restrict free variable occurrences in HOAS so as to support recursion. In the first category we have encodings based on nominal logic [10], which allows us to readily capture $\alpha$-equivalence in terms but still requires that substitution and hypothetical judgments be defined for each object-language. Work in the second category is more diverse. For instance, we see the use of modality to restrict free variables [21] as well as proposals that stratify the meta-language into separate representation and computation levels [22, 14]. Also in this category is the so-called “weak” HOAS in which variables are drawn from a different type than object-level terms [8].

We are to demonstrate a practical solution which falls into the second category. We make use of a (restricted) form of dependent types originally developed in Dependent ML [29, 23] to stratify the meta-language into separate representation and computation layers. The higher-order representation is done in the statics, where the function space is restricted (e.g., no recursion is allowed). This representation is linked to the computation layer through the use of dependent types (the higher-order representations become type indices for a first-order representation in the dynamics). With this strategy, we are required to define substitution for the first-order representation, but we can assure the correctness of our implementation by making use of the higher-order representation in type indices.

2. The meta-language: ATS

A primary motivation for developing ATS [27] stems from an earlier attempt to support a (restricted) form of dependent types in practical programming. While there is already a framework Pure Type System (PTS) [3] that offers a simple and general approach to designing and formalizing type systems, it is well understood that there often exist some acute problems (in the presence of dependent types) making it difficult for PTS to accommodate many common realistic programming features. In particular, we have learned that some great efforts are required in order to maintain a style of pure reasoning as is advocated in PTS when programming features such as general recursion [7], recursive types [15], effects [13], exceptions [11] and input/output are present.
To address such limitations of PTS, ATS is developed to allow for designing and formalizing (advanced) type systems that can readily support common realistic programming features. The key salient feature of ATS lies in a complete separation between statics, in which types are formed and reasoned about, from dynamics, in which programs are constructed and evaluated. This separation, with its origin in a previous study on a restricted form of dependent types developed in Dependent ML (DML) [29, 23], makes it feasible to support dependent types in the presence of effects such as references and exceptions.

ATS is a programming language with an expressive type system rooted in ATS, and ATS/LF is a component in ATS primarily developed to support a paradigm that combines programming with theorem proving [5]. The statics of ATS itself is a simply typed language and the types in it are referred to as sorts. For instance, we have the base sorts bool (for boolean values), int (for integers I), prop (for props P) and type (for types T). There are also certain built-in static constructors (e.g., various type constructors) and functions (e.g., ∧, ∨, +, ·). A prop is just like a type, but it can only be assigned to total dynamic terms (containing no effects such as nontermination). In particular, there are only props in ATS/LF (as ATS/LF contains only proof terms). We may use the name proof term to refer to a dynamic term that can be assigned a prop.

In ATS, it is also allowed to introduce new sorts. For instance, the following concrete syntax introduces a sort $tm$:

```
datasort tm = lm of (tm -> tm) | ap of (tm, tm)
```

and there two static constructors $ap$ and $lm$ associated with $tm$ which are given the following sorts:¹

```
ap : (tm, tm) -> tm
lm : (tm -> tm) -> tm
```

However, we do not allow definition by recursion over static terms. Because of this restriction, we may freely define datatypes without negative occurrences of type constructors.

We assume a constraint relation of the form $\Sigma; B_1, \ldots, B_n \models B$ that determines the truth of $B$ assuming $B_1, \ldots, B_n$, where $\Sigma$ is a static context containing all free static variables in $B_1, \ldots, B_n, B$. In practice, we impose some restrictions (e.g., only allowing terms of the sort bool to be linear inequalities on integers) so as to make the constraint relation (effectively) decidable.

A tiny portion of the syntax of the dynamics is given in Figure 1. We refer to $\lambda : \sigma.d$ and $d(s)$ as static abstraction and static application, respectively.

The design of the concrete syntax of ATS is inspired by that of SML. As in SML, datatypes can be declared in ATS. Furthermore, datatypes in ATS can be indexed by static terms (aka. type indices). Also, datatypes are available in ATS/LF, which are just the prop version of datatypes. One major addition in the concrete syntax is universal quantification over static terms, which is used extensively in function signatures and datatype and datadrop definitions. As an example, we can introduce a datadrop as follows:

```
datadrop EVEN (int, bool) =
  | zero (0, true)
  | {n:int | n > 0} even (n+1, true) of EVEN (n, false)
  | {n:int | n > 0} odd (n+1, false) of EVEN (n, true)
```

where the syntax $\{n:int | n > 0\}$ stands for universal quantification over $n : \text{int}$ with guard $n \geq 0$. We may also use the subset sort $nat$ which specifies nonnegative elements of $\text{int}$, and the syntax $\{n:nat | n > 0\}$ has the same meaning as $\{n:int | n > 0\}$.

¹ In ATS, we support constructors and functions of multiple arguments, so the sort $(tm, tm) \to tm$ indicates that $ap$ forms a static term of the sort $tm$ when applied to two static terms of the sort $tm$.

In order to guarantee that recursively defined proof functions are terminating, we require that termination metrics be explicitly provided (by the programmer). In the current implementation of ATS, a termination metric consists of a tuple of static natural numbers that are lexicographically decreasing in each recursive call. Also, case coverage is guaranteed by making sure any cases not listed are redundant. Please see [25, 24] for details on termination metrics and [26] for details on case coverage checking.

We now present an example to explain how totality checking works in ATS/LF. The following function is a proof that the sum of two even natural numbers is even.

```
prfun evenAddEvenIsEven:n1:nat,n2:nat. <n1,n2>.
  (pf1: EVEN(n1,true), pf2: EVEN(n2,true))

  : EVEN(n1+n2,true) =
    // [case*] mandates exhaustive pattern matching
    case* pf1 of
        | zero () => pf2
        | even pf1 => even (oddAddEvenIsOdd(pf1,pf2))
    end

  oddAddEvenIsOdd:n1:nat,n2:nat. <n1,n2>.
  (pf1: EVEN(n1,false), pf2: EVEN(n2,true))

  : EVEN(n1+n2,false) =
    // [case*] mandates exhaustive pattern matching
    case* pf1 of
        | odd pf1 => odd (evenAddEvenIsEven(pf1,pf2))
    end
```

The syntax $\text{term} : \text{type}$ asserts a type or prop to a dynamic term, or a sort to a static term, the final $\text{type}$ in the function signature specifies the return type of the function, which for $\text{evenAddEvenIsEven}$ is a proof that $n_1 + n_2$ is even if both $n_1$ and $n_2$ are even. The proof works by case analysis on the proof that $n_1$ is even. If $n_1$ is 0, then $n_1 + n_2 = n_2$ so the proof that $n_2$ is even suffices for the conclusion. If $n_1$ is an odd number plus one, we use $\text{oddAddEvenIsOdd}$ to prove that $n_1 - 1 + n_2 = n_1 + n_2$ is odd, and use the even constructor to prove $n_1 - 1 + n_2 + 1 = n_1 + n_2$ is even. Note that the prop assigned to $\text{evenAddEvenIsEven}$ is formally written as follows:

```
\forall n_1 : \text{nat}. \forall n_2 : \text{nat}. \ E\VEN(n_1,\true) \& \ E\VEN(n_2,\true) \rightarrow \ E\VEN(n_1+n_2,\true)
```

The mutually recursive proof functions $\text{evenAddEvenIsEven}$ and $\text{oddAddEvenIsOdd}$ pass totality checking because (1) the metric $n_1$ (specified by the concrete syntax $.\text{even}$. ) is decreasing in the recursive call (it is $n_1 - 1$ in the recursive call if $n_1 > 0$ holds) and (2) all cases are determined to have been covered: those that are not considered are all redundant because taking one of these branches would introduce the contradictory $\text{true} = false$ or $\text{false} = \text{true}$ assumption. Also note that the integer inequality decision procedure is used to solve the constraint $((n_1 - 1) + n_2) + 1 = n_1 + n_2$ to check that the result of the second (resp. the first) branch in the definition of $\text{evenAddEvenIsEven}$ (resp. $\text{oddAddEvenIsOdd}$) proves the correct prop.
3. Representing Object Syntax

We now present an encoding of the pure untyped \(\lambda\)-calculus in ATS. The syntax of the object language is given as follows:

\[
M := x | \lambda x. M | M_1(M_2)
\]

A dataset \(tm\) for HOAS representation of this language can be declared in ATS with which the following static constructors are associated:

\[
ap : (tm, tm) \rightarrow tm \quad lm : (tm \rightarrow tm) \rightarrow tm
\]

Let us define a function \(\Gamma \cdot \cdot\) as follows:

\[
\Gamma x = x \quad \Gamma x. M = lm(\lambda x : tm. \Gamma M) \quad \Gamma M_1(M_2) = ap(\Gamma M_1, \Gamma M_2)
\]

which translates typed \(\lambda\)-expressions into static terms of the sort \(tm\).

In order to push this higher-order representation down to a first-order representation type, we will represent terms as terms-in-contexts [9, 2]. The reason for doing this is that it will allow us to capture meta-variables as de Bruijn indices for a first-order in-contexts [9, 2].

3.1 Substitution

Using this representation for object terms, we must define substitution on the first order

\[
\text{fun subst } \{t:tm, ts:tm, t':tm, n:nat\} .<n>.
\]

where

\[
\begin{align*}
\text{TERM} & := \text{VAR} (t) \mid \text{lam} (t :: ts) \\
\text{TERM} & := \text{app} (t', e) \mid \text{lam} (t :: ts) \mid \text{fix} (x.d) \mid \text{app} (d_1, d_2) \mid \lambda a : \sigma. d \mid d(s) \ldots
\end{align*}
\]

Formally, the three constructors \(\text{VAR}\), \(\text{LAM}\) and \(\text{APP}\) associated with \(\text{TERM}\) are given the types listed in Figure 2.

The intuition behind \(\text{TERM}\) can be readily explained. Assume that \(ts = t_1 :: \ldots :: t_n:: \text{nil}\) and \(t_i = \Gamma M_i\) for \(i = 1, \ldots, n\), that is, \(t_i\) represent \(M_i\). Then a term of type \(\text{TERM} (ts, t, n)\) represents a (possibly open) pure untyped \(\lambda\)-expression \(M\) of size \(n\) such that \(t = \Gamma M[x_1, \ldots, x_n\rightarrow M_1, \ldots, M_n]\).

Note that this first-order representation for pure untyped \(\lambda\)-expressions still retains some aspects of higher-order abstract syntax. In particular, the \(\text{LAM}\) constructor takes a term of some universally quantified type. In order for this representation to be adequate, we need to know that the argument of \(\text{LAM}\), which is a dependent function, is parametric in \(x\) (i.e. there is no case-distinction on \(x\)). In the fragment of ATS presented, all quantification is parametric and statics are fully reasailable. In the full language we allow branching within props on terms of sort \(\text{bool}\), and distinguish “parametric sorts” which admit no (non-trivial) predicates for terms of the sort \(\text{bool}\), and \(tm\) is such a parametric sort.

As an example, the first-order representation for the \(\lambda x. \lambda y. g(x)\) is:

\[
\text{LAM} (\text{LAM} (\text{APP} (\text{VAR} (\text{INone}), \text{VAR} (\text{INshi} (\text{INone}))))))
\]

which corresponds precisely to the de Bruijn notation \(\lambda \lambda. 0(1)\).

In general, we can represent a \(\lambda\)-expression with at most \(n\) free variables as a term of the following type:

\[
\forall x_1 : tm \ldots \forall x_n : tm. \text{TERM} (x_1 :: \ldots :: x_n :: \text{nil}, t),
\]

where \(t\) is a static term of the sort \(tm\) that may contain free occurrences of \(x_1, \ldots, x_n\).

3.1 Substitution

Using this representation for object terms, we must define substitution on the first order \(\text{TERM}\) type. The substitution lemma is defined as follows:

\[
\begin{align*}
\text{fun subst } \{t:tm, ts:tm, t':tm, n:nat\} .<n>. \\
\text{(e1: TERMINAL(t :: ts, t' :: n), e2: TERMINAL (ts, t))} \\
\text{TERMNO (ts, t')} = \\
\text{case* e1 of} \\
\text{VAR i => (case* i of INone () => e2} \\
\text{INshi i' => VAR i')} \\
\text{APP (e11, e12) =>} \\
\text{APP (subst (e11, e2), subst (e12, e2))} \\
\text{LAM ef =>} \\
\text{LAM (lam {x:tm} =>} \\
\text{subst (exch1 (ef{x}), weaken1 e2))}
\end{align*}
\]

This function needs some explanation. First of all we make use of exchange (\(\text{exch1}\)) and weakening (\(\text{weaken1}\)) as lemmas. These lemmas perform exchange or weakening at the first position in the context. Additionally, we supply a termination metric even though this is a program term and not a proof term. Since we do not make use of any effectful constructs within this function, the metric guarantees termination as expected. The \(\text{LAM}\) case is particularly

\footnote{We have made static abstraction and application implicit here.}
Fortunately, since we know that the universal quantification is para-

consistent, and since terms of sort

application as environment extension rather than actual substitu-
in particular, we will use environments and closures and implement

in order to demonstrate the utility of this style of programm-
ing with a combination of higher-order and first-order representa-
tion, we

seems to have the type

and the size of the evaluation.

We will now define a prop to identify values, which in our case is

propdef EVAL0(t1:tm,t2:tm) = \[n:nat\] EVAL(t1,t2,n)

but the

legal. The substitution function is given the following typ-

e.

The types of the constructors associated with TER-

Figure 2. The types of the constructors associated with TERM

interesting because in the result of that case, the argument to LAM

1am \{x:tm\} => subst (exch1 (ef(x)), weaken1 e2)

seems to have the type

\[\forall x: tm. \exists n: nat. \text{TERM}(x :: ts, t', n)\]

but the LAM constructor expects an argument of type

\[\exists n: nat. \forall x: tm. \text{TERM}(x :: ts, t', n)\].

Fortunately, since we know that the universal quantification is para-

metric, and since terms of sort int cannot contain terms of sort tm,

we know \(n\) cannot depend on \(x\), so this quantifier alternation is

legal. The substitution function is given the following type:

\[\forall ts: tms. \forall t: tm'. \forall n: nat. \text{TERM}(t :: ts, t', n)\]

(\text{TERM}(t :: ts, t', n), \text{TERM0}(ts, t)) \rightarrow

\text{TERM0}(ts, t').

And this type shows that the substitution function returns the proper

substituted term.

3.2 Evaluation

In this section we define the call-by-value big-step evaluation rules

for the language and give an implementation of an evaluator which

is verified correct with respect to this definition. We define the

semantics as a datatype \text{EVAL} indexed by a term, its evaluation

and the size of the evaluation.

dataprop EVAL(tm, tm, int) =

| \{f:tm->tm\} EVALlam (lm f, lm f, 0)

| \{t1:tm, t2:tm, f1:tm->tm, v2:tm, v:tm,

| n1:nat, n2:nat, n3:nat\} EVALapp (ap (t1, t2), v, n1+n2+\n
| 1) of

(\text{EVAL}(t1, lm f1, n1),

\text{EVAL}(t2, v2, n2),

\text{EVAL}(f1 v2 v, n3))

prodef EVAL0(t1:tm,t2:tm) = \[n:nat\] EVAL(t1,t2,n)

We will now define a prop to identify values, which in our case is

only \(\lambda\)-expression. Additionally we prove value soundness and its

converse, which will be used as lemmas in the evaluation function.

dataprop ISVAL (tm) = \{f:tm->tm\} ISVAL (lm f)

prfun lemma1 \{t:tm, v:tm, n:nat\} .\langle n\rangle.

(\text{pf}: \text{EVAL} (t, v, n)): \text{ISVAL} v =

case* pf of

| \text{EVALlam} () => ISVAL ()

| \text{EVALapp} (\_, \_, pf3) => lemma1 pf3

prfun lemma2 \{t:tm\} .\langle \rangle.

(\text{pf}: \text{ISVAL} t): \text{EVAL} (t, t, 0) =

case* pf of \text{ISVAL}() => \text{EVALlam} ()

In order to demonstrate the utility of this style of programming with

a combination of higher-order and first-order representation, we

will implement evaluation in a different way than it was specified.

In particular we will use environments and closures and implement

application as environment extension rather than actual substitution

(as it is specified). In order to do this, we create datatypes for values

and environments as follows:

datatype VAL (tm) =

| \{ts:tm, f: tm->tm\} VALclo (lm f) of

(\text{ENV} ts, \{t:tm\} \text{TERMO} (t :: ts, f t))

| \text{ENVnil} (nil1)

| \{ts:tm, t:tm\} ENVcons (t :: ts) of

(\text{ISVAL} t | \text{ENV} ts, \text{VAL} t)

The only values are closures which are a pair of an environment

and a function term for that environment. An environment is a list

of values along with proofs that they are values. In the definition

of \text{ENV} we first see props and types living side by side. The syntax

(P | T) forms a type which is a tuple of a prop, P, and a type, T.

After type-checking, the proof term corresponding to the prop will

be erased.

We can now use these definitions to implement an evaluation

function in Figure 3, where the type of the function proves its partial

correctness. Note that the function \text{eval} is given the following type:

\[\forall ts: tms. \forall t: tm. \forall p: nat. \text{TERM0}(ts, t, p) \rightarrow

\forall v: tm. \text{EVAL}(t, v) \rightarrow \text{VAL}(v)\]

And this type guarantees that if the function returns, it returns the

value specified by the \text{EVAL} prop, so as long as \text{EVAL} correctly

characterizes the evaluation relation, this eval function is partially

correct. After type-checking, the proof for the \text{EVAL} prop is

erased and we are left with a verified evaluator.

3.3 Other Examples

We have completed a suite of examples, which can be found on-

line [28]. Other examples that we have applied this methodology to

are simply typed \(\lambda\)-calculus with products and fix-point as well as

mini-ML with references. The example of mini-ML with references

is particularly interesting because we specify evaluation using a list

of values to represent references, leading to a linear lookup time

for references, but we implement evaluation using an array to store

references, allowing for constant time lookup. Unfortunately this

element is too complicated to adequately present here.

4. Related Work and Conclusion

There is a good deal of work on supporting induction on higher-

order abstract syntax. Hofmann [12] gives a survey of several

variants of HOAS and shows how to derive induction principles for

them. We consider our approach to be most closely related to the

stratified meta-logic described by Miller and McDowell in [14]. In

their system, they start with a simple intuitionistic meta-meta-logic

with natural numbers induction, then encode an object-meta-logic

of sequents used for judgments (using a similar representation for

variable lookups as \text{IN}(tms, tm)). They then encode languages

within the object-meta-logic and do inductive proofs on judgments

within the meta-meta-logic. Our method basically collapses the

object-meta-logic and object-language.
fun evalVar {ts:tms, t:tm}
  (i: IN (t, ts), env: ENV ts)
  : '(ISVAL t | VAL t) =
  case* i of
    | INone () =>
      let val ENVcons(pf | _, v) = env in
        'pf | v
      end
    | INshi i =>
      let val ENVcons (_, env, _) = env in
        evalVar (i, env)
      end
  end

fun eval {ts:tms, t:tm}
  (e: TERO0 (ts,t), env: ENV ts)
  : [v:tm] '(EVAL0(t,v) | VAL v) =
  case* e of
    | VAR i =>
      let
        val '(pf | v) = evalVar (i, env)
        in
        'lemma0 pf | v
      end
    | LAM ef => 'EVALlam () | VALclo (env, ef)
    | APP (e1, e2) =>
      let
        val '(pf1 | v1) = eval (e1, env)
        val '(pf2 | v2) = eval (e2, env)
        val VALclo (env0, ef1) = v1
        val '(pf3 | v) =
          eval (ef1..., ENVcons (lemma1 pf2 | env0, v2))
        in
        'EVALapp (pf1, pf2, pf3) | v
      end
  end

Figure 3. A verified evaluation function for λ-calculus

More recently, Miller and Tiu introduced the ∇ quantifier to
allow for local scoping of parameters in order to eliminate the need for the object-meta-logic [16]. Schürrmann et al’s ∇-calculus [22] uses a similar idea to support functional programming over higher-order abstract syntax in the Delphin language. In [21], Schürrmann, Despeyroux, Pfennig use a modal operator distinguish closed HOAS terms, over which induction is allowed. This lets them allow primitive recursive functions over HOAS without losing adequacy. While this formulation is simpler than our mixed representation, it is also less flexible in that it does not allow recursion over open terms.

Our representation also bears a close resemblance to the terms-in-context representation introduced by Despeyroux and Hirschowitz in [9], and more recently by Ambler, Crole and Momigliano in [2]. The main difference in our approach is that we use separate languages for the higher-order and first order representations. This separation allows us to use full HOAS rather than weak HOAS. Additionally we use finite-lists for contexts rather than functions-as-infinite-lists for contexts.

Ambler, Crole and Momigliano’s Hybrid [1] uses de Bruijn representation under the hood to validate an induction principle over HOAS. This strategy is very similar to our own, however the motivation and applications are different. In addition, because they are working within Isabelle which has only one function space, they are forced to define predicates for function parametricity and expression non-exotiness.

In [4], Bird and Paterson give a representation of de Bruijn terms as a nested datatype. This representation is somewhat similar to ours in that the free variables of a term are constrained by the type and this can be used to guarantee well-scopedness. Our representation, however, allows for verification of many more complex properties as the type can specify a fixed substitution of terms for variables in the de Bruijn term. This enables specification and verification of evaluation, among other things.

Along a different line, Pitts and Gabbay develop the theory of FM-sets into a logic which supports fresh-variable quantification, nominal logic, and a programming language based on this logic [20, 10]. Using such nominal encodings gives one α-conversion for free and eliminates the need for complex side conditions on rules, but one must still define substitution. Nanaveksi [17] combines fresh-name quantification and a modality to allow for construction of more efficient meta-programs.

We have investigated an encoding in ATS that combines higher-order abstract syntax with first-order syntax. A significant advantage of this encoding is that we may define recursive functions over first-order abstract syntax while ensuring the correctness of these functions through higher-order abstract syntax. Also, we have completed a variety of running examples in ATS in support of the viability of this approach, some of which are readily available online [28].

References


