

Combining Higher-Order Abstract Syntax with First-Order Abstract Syntax in ATS

Kevin Donnelly Hongwei Xi

Boston University
{kevind, hwxi}@cs.bu.edu

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 α -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 λ -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 α -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 λ -abstractions. This approach enables us to reuse meta-level α -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 α -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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MERLIN’05 September 30, 2005, Tallinn, Estonia.
Copyright © 2005 ACM 1-59593-072-8/05/0009...\$5.00.

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 booleans *B*), *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., \wedge , \vee , $+$, $-$). 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 datasorts with negative occurrences without sacrificing normalization of static terms.

We assume a constraint relation of the form $\Sigma; B_1, \dots, B_n \models B$ that determines the truth of *B* assuming B_1, \dots, B_n , where Σ is a static context containing all free static variables in B_1, \dots, 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 $\lambda a : \sigma.d$ 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 dataproduct definitions. As an example, we can introduce a dataproduct as follows:

```
dataproduct 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 $\{n:int \mid n \geq 0\}$ stands for universal quantification over $n : int$ with guard $n \geq 0$. We may also use the subset sort *nat* which specifies nonnegative elements of *int*, and the syntax $\{n:nat\}$ has the same meaning as $\{n:int \mid n \geq 0\}$. Given an integer *I*, the prop $\mathbf{EVEN}(I, true)$ (resp. $\mathbf{EVEN}(I, false)$)

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

are for proofs that *I* is an even (resp. odd) natural number. We have three proof constructors associated with \mathbf{EVEN} , which are given the following props:

```
zero :  $\mathbf{EVEN}(0, true)$ 
even  :  $\forall n : int. n \geq 0 \supset$ 
          $\mathbf{EVEN}(n, false) \rightarrow \mathbf{EVEN}(n+1, true)$ 
odd   :  $\forall n : int. n \geq 0 \supset$ 
          $\mathbf{EVEN}(n, true) \rightarrow \mathbf{EVEN}(n+1, false)$ 
```

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>.
  (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))

and oddAddEvenIsOdd {n1:nat,n2:nat} .<n1>.
  (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))
```

The syntax *term:type* asserts a type or prop to a dynamic term, or a sort to a static term, the final *:type* in the function signature specifies the return type of the function, which for *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 *oddAddEvenIsOdd* to prove that $n_1 - 1 + n_2$ 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 *evenAddEvenIsEven* is formally written as follows:

$$\forall n_1 : nat. \forall n_2 : nat. \\ (\mathbf{EVEN}(n_1, true), \mathbf{EVEN}(n_2, true)) \rightarrow \\ \mathbf{EVEN}(n_1 + n_2, true)$$

The mutually recursive proof functions *evenAddEvenIsEven* and *oddAddEvenIsOdd* pass totality checking because (1) the metric n_1 (specified by the concrete syntax *.<n1>*.) 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 *true = false* or *false = 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 *evenAddEvenIsEven* (resp. *oddAddEvenIsOdd*) proves the correct prop.

dynamic terms d	$::=$	$x \mid dc(d_1, \dots, d_n) \mid \mathbf{lam} \ x.d \mid \mathbf{fix} \ x.d \mid \mathbf{app}(d_1, d_2) \mid \lambda a : \sigma.d \mid d(s) \dots$
values v	$::=$	$x \mid dcc(v_1, \dots, v_n) \mid \mathbf{lam} \ x.d \mid \dots$
dyn. var. ctx. Δ	$::=$	$\emptyset \mid \Delta, x : T$

Figure 1. Syntax for the dynamics of ATS

3. Representing Object Syntax

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

$$M ::= x \mid \lambda x.M \mid M_1(M_2)$$

A datasort 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 $\ulcorner \cdot \urcorner$ as follows:

$$\begin{aligned} \ulcorner x \urcorner &= x \\ \ulcorner \lambda x.M \urcorner &= lm(\lambda x : tm. \ulcorner M \urcorner) \\ \ulcorner M_1(M_2) \urcorner &= ap(\ulcorner M_1 \urcorner, \ulcorner M_2 \urcorner) \end{aligned}$$

which translates typed λ -expressions into static terms of the sort tm . Because the statics is nothing more than simply typed lambda calculus with products and constants, we know that that $\ulcorner \cdot \urcorner$ is a compositional bijection between canonical forms (i.e., η -long β -normal forms in this case) of the sort tm and pure untyped λ -expressions [18].

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 representation. To do so, we first define the following sort for lists of terms of the sort tm .

`infixr :: // a right associative infix operator`
`datasort tms = nil | :: of (tm, tms)`

We now define a type **IN** for terms showing that a given static term of the sort tm occurs in a static term of the sort tms :

```
datatype IN (tm, tms) =
  | {ts:tms, t:tm} INone (t, t :: ts)
  | {ts:tms, t:tm, t':tm}
    INshi(t, t' :: ts) of IN (t, ts)
```

Formally, the two constructors *INone* and *INshi* associated with **IN** are given the following types:

$$\begin{aligned} \mathbf{INone} &: \forall ts : tms. \forall t : tm. () \rightarrow \mathbf{IN}(t, t :: ts) \\ \mathbf{INshi} &: \forall ts : tms. \forall t : tm. \forall t' : tm. \\ &\quad \mathbf{IN}(t, ts) \rightarrow \mathbf{IN}(t, t' :: ts) \end{aligned}$$

It should soon be clear that such terms correspond to de Bruijn indices: *INone* stands for the first de Bruijn index and *INshi* increases a given de Bruijn index by 1. We can now define a first-order type constructor **TERM** as follows:

```
datatype TERM (tms, tm, int) =
  | {ts:tms, t:tm} VAR (ts, t, 0) of IN(t, ts)
  | {ts:tms, t:tm, t':tm, n:nat, m:nat}
    APP (ts, ap(t, t'), m+n+1) of
      (TERM (ts, t, n), TERM (ts, t', m))
  | {ts:tms, f: tm->tm, n:nat}
    LAM (ts, lm f, n+1) of
      ({x:tm} TERM (x :: ts, f x, n))
```

```
typedef TERM0 (ts:tms, t:tm) = [n:nat] TERM (ts, t, n)
```

Formally, the three constructors *VAR*, *LAM* and *APP* associated with **TERM** are given the types listed in Figure 2.

The intuition behind **TERM** can be readily explained. Assume that $ts = t_1 :: \dots :: t_n :: nil$ and $t_i = \ulcorner M_i \urcorner$ for $i = 1, \dots, n$, that is, t_i represent M_i . Then a term of type **TERM**(ts, t, n) represents a (possibly open) pure untyped λ -expression M of size n such that $t = \ulcorner M[x_1, \dots, x_n \mapsto M_1, \dots, M_n] \urcorner$.

Note that this first-order representation for pure untyped λ -expressions still retains some aspects of higher-order abstract syntax. In particular, the *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 *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 erasable. In the full language we allow branching within props on terms of sort *bool*, and distinguish “parametric sorts” which admit no (non-trivial) predicates for forming terms of the sort *bool*, and *tm* is such a parametric sort.

As an example, the first-order representation for the λ -expression $\lambda x. \lambda y. y(x)$ is:²

$$\mathbf{LAM}(\mathbf{LAM}(\mathbf{APP}(\mathbf{VAR}(\mathbf{INone}), \mathbf{VAR}(\mathbf{INshi}(\mathbf{INone}))))))$$

which corresponds precisely to the de Bruijn notation $\lambda. \lambda. 0(1)$. In general, we can represent a λ -expression with at most n free variables as a term of the following type:

$$\forall x_1 : tm. \dots \forall x_n : tm. \mathbf{TERM}(x_1 :: \dots :: x_n :: nil, t),$$

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

3.1 Substitution

Using this representation for object terms, we must define substitution on the first order **TERM** type. The substitution lemma is defined as follows:

```
fun subst {t:tm, ts:tms, t':tm, n:nat} .<n>.
  (e1: TERM(t :: ts, t', n), e2: TERM0 (ts, t))
  : TERM0 (ts, t') =
  case* e1 of
  | VAR i => (case* i of INone () => e2
    | INshi i' => VAR i')
  | APP (e11, e12) =>
    APP (subst (e11, e2), subst (e12, e2))
  | LAM ef =>
    LAM (lam {x:tm} =>
      subst (exch1 (ef{x}), weaken1 e2))
```

This function needs some explanation. First of all we make use of exchange (*exch1*) and weakening (*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 *LAM* case is particularly

² We have made static abstraction and application implicit here.

<i>VAR</i>	: $\forall ts : tms. \forall t : tm. (\mathbf{IN}(t, ts)) \rightarrow \mathbf{TERM}(ts, t, 0)$
<i>LAM</i>	: $\forall ts : tms. \forall f : tm \rightarrow tm. \forall n : nat. (\forall x : tm. \mathbf{TERM}(x :: ts, f(x), n)) \rightarrow \mathbf{TERM}(ts, \mathbf{lm}(f), n + 1)$
<i>APP</i>	: $\forall ts : tms. \forall t_1 : tm. \forall t_2 : tm. \forall n_1 : nat. \forall n_2 : nat. (\mathbf{TERM}(ts, t_1, n_1), \mathbf{TERM}(ts, t_2, n_2)) \rightarrow \mathbf{TERM}(ts, \mathbf{ap}(t_1, t_2), n_1 + n_2 + 1)$

Figure 2. The types of the constructors associated with **TERM**

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

```
lam {x:tm} => subst (exch1 (ef{x}), weaken1 e2)
```

seems to have the type

$$\forall x : tm. \exists n : nat. \mathbf{TERM}(x :: ts, t', n)$$

but the *LAM* constructor expects an argument of type

$$\exists n : nat. \forall x : tm. \mathbf{TERM}(x :: ts, t', n).$$

Fortunately, since we know that the universal quantification is parametric, 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:

$$\begin{aligned} &\forall ts : tms. \forall t : tm. \forall t' : tm. \forall n : nat. \\ &(\mathbf{TERM}(t :: ts, t', n), \mathbf{TERMO}(ts, t)) \rightarrow \\ &\mathbf{TERMO}(ts, t'). \end{aligned}$$

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 dataprop **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+n3+1) of
    (EVAL(t1, lm f1, n1),
     EVAL(t2, v2, n2),
     EVAL(f1 v2, v, n3))
```

```
propdef 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 λ -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} .<n>.
(pf: EVAL (t, v, n)): ISVAL v =
case* pf of
| EVALlam () => ISVAL ()
| EVALapp (_, _, pf3) => lemma1 pf3
```

```
prfun lemma2 {t:tm} .<>.
(pf: ISVAL t): EVAL (t, t, 0) =
case* pf of ISVAL() => 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:tms, f: tm->tm}
VALclo (lm f) of
  (ENV ts, {t:tm} TERMO (t :: ts, f t))
```

```
and ENV (tms) =
| ENVnil (nil)
| {ts:tms, t:tm}
  ENVcons (t :: ts) of
    (ISVAL t | ENV ts, 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 **ENV** we first see props and types living side by side. The syntax $(P \mid T)$ 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 *eval* is given the following type:

$$\begin{aligned} &\forall ts : tms. \forall t : tm. \forall p : nat. \\ &(\mathbf{TERMO}(ts, t), \mathbf{ENV}(ts, p)) \rightarrow \\ &\exists v : tm. (\mathbf{EVAL0}(t, v) \mid \mathbf{VAL}(v)) \end{aligned}$$

And this type guarantees that if the function returns, it returns the value specified by the **EVAL** prop, so as long as **EVAL** correctly characterizes the evaluation relation, this eval function is partially correct. After type-checking, the proof for the **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 online [28]. Other examples that we have applied this methodology to are simply typed λ -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 example 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 $\mathbf{IN}(tms, tm)$). 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

fun eval {ts:tms, t:tm}
  (e: TERMO (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
      '(lemma2 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

```

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]. Schürmann *et al*'s ∇ -calculus [22] uses a similar idea to support functional programming over higher-order abstract syntax in the Delphin language.

In [21], Schürmann, Despeyroux, Pfenning 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 principal 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-exoticness.

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. Nanevski [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

- [1] AMBLER, S. J., CROLE, R. L., AND MOMIGLIANO, A. Combining Higher Order Abstract Syntax with Tactical Theorem Proving and (Co)Induction. In *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics*, Hampton, VA, USA (2002), vol. 2410 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 13–30.
- [2] AMBLER, S. J., CROLE, R. L., AND MOMIGLIANO, A. A definitional approach to primitive recursion over higher order abstract syntax. In *MERLIN '03: Proceedings of the 2003 workshop on Mechanized reasoning about languages with variable binding* (New York, NY, USA, 2003), ACM Press, pp. 1–11.
- [3] BARENDREGT, H. P. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. Maibaum, Eds., vol. II. Clarendon Press, Oxford, 1992, pp. 117–441.
- [4] BIRD, R. S., AND PATERSON, R. de bruijn notation as a nested datatype. *J. Funct. Program.* 9, 1 (1999), 77–91.
- [5] CHEN, C., AND X1, H. Combining programming with theorem proving. In *Proceedings of the 10th International Conference on Functional Programming (ICFP'05)* (September 2005).
- [6] CHURCH, A. *The Calculi of Lambda-Conversion*, vol. 6 of *Annals of Mathematics Studies*. Princeton University Press, Princeton, 1941.
- [7] CONSTABLE, R. L., AND SMITH, S. F. Partial objects in constructive type theory. In *Proceedings of Symposium on Logic in Computer Science* (Ithaca, New York, June 1987), pp. 183–193.
- [8] DESPEYROUX, J., FELTY, A. P., AND HIRSCHOWITZ, A. Higher-order abstract syntax in coq. In *TLCA* (1995), pp. 124–138.
- [9] DESPEYROUX, J., AND HIRSCHOWITZ, A. Higher-order abstract syntax with induction in coq. In *LPAR '94: Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning* (London, UK, 1994), Springer-Verlag, pp. 159–173.
- [10] GABBAY, M. J., AND PITTS, A. M. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing* 13 (2001), 341–363. Special issue in honor of Rod Burstall. To appear.
- [11] HAYASHI, S., AND NAKANO, H. *PX: A Computational Logic*. The MIT Press, 1988.
- [12] HOFMANN, M. Semantical analysis of higher-order abstract syntax.

- In *LICS '99: Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science* (Washington, DC, USA, 1999), IEEE Computer Society, p. 204.
- [13] HONSELL, F., MASON, I. A., SMITH, S., AND TALCOTT, C. A variable typed logic of effects. *Information and Computation* 119, 1 (15 May 1995), 55–90.
 - [14] McDOWELL, R., AND MILLER, D. A logic for reasoning with higher-order abstract syntax. In *LICS* (1997), pp. 434–445.
 - [15] MENDLER, N. Recursive types and type constraints in second-order lambda calculus. In *Proceedings of Symposium on Logic in Computer Science* (Ithaca, New York, June 1987), The Computer Society of the IEEE, pp. 30–36.
 - [16] MILLER, D., AND TIU, A. A proof theory for generic judgments: An extended abstract. In *Proceedings of LICS 2003* (June 2003), IEEE, pp. 118–127.
 - [17] NANEVSKI, A. Meta-programming with names and necessity. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2002), ACM Press, pp. 206–217.
 - [18] PFENNING, F. *Computation and Deduction*. Cambridge University Press. (to appear).
 - [19] PFENNING, F., AND ELLIOTT, C. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation* (Atlanta, Georgia, June 1988), pp. 199–208.
 - [20] PITTS, A. M., AND GABBAY, M. J. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Program Construction. 5th International Conference, MPC2000, Ponte de Lima, Portugal, July 2000. Proceedings* (2000), R. Backhouse and J. N. Oliveira, Eds., vol. 1837 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, pp. 230–255.
 - [21] SCHÜRMANN, C., DESPEYROUX, J., AND PFENNING, F. Primitive recursion for higher-order abstract syntax. *Theor. Comput. Sci.* 266, 1–2 (2001), 1–57.
 - [22] SCHÜRMANN, C., POSWOLSKY, A., AND SARNAT, J. The [triangle]-calculus. functional programming with higher-order encodings. In *TLCA* (2005), P. Urzyczyn, Ed., vol. 3461 of *Lecture Notes in Computer Science*, Springer, pp. 339–353.
 - [23] XI, H. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. pp. viii+189. Available at <http://www.cs.cmu.edu/~hwxi/DML/thesis.ps>.
 - [24] XI, H. Dependent Types for Program Termination Verification. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science* (Boston, June 2001), pp. 231–242.
 - [25] XI, H. Dependent Types for Program Termination Verification. *Journal of Higher-Order and Symbolic Computation* 15, 1 (March 2002), 91–132.
 - [26] XI, H. Dependently Typed Pattern Matching. *Journal of Universal Computer Science* 9, 8 (2003), 851–872.
 - [27] XI, H. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003* (2004), Springer-Verlag LNCS 3085, pp. 394–408.
 - [28] XI, H. Applied Type System, 2005. Available at: <http://www.cs.bu.edu/~hwxi/ATS>.
 - [29] XI, H., AND PFENNING, F. Dependent Types in Practical Programming. In *Proceedings of 26th ACM SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas, January 1999), ACM press, pp. 214–227.