Type-directed Syntax Transformations for ATS-style Programs with Proofs

Andrei Lapets

August 14, 2007

1 Introduction

In their work on ATS [CX05] and ATS^{LF} [CX04], the authors present an ML-like language which distinguishes between term-level "dynamic" expressions which may contain side effects and pure "static" expressions which are functions on proof objects and might occur inside types. This distinction allows programmers to combine code (potentially containing side effects) with proofs, and with the help of an erasure property [CX05], these proofs can be discarded when programs are compiled.

Unfortunately, some practical inconveniences arise when programming in a language which uses this approach. Particularly, it is cumbersome to use many of the operations and relations we commonly associate with data types (for example, addition, multiplication, and ordering of naturals, concatenation of lists, etc.) in both proofs and dynamic terms without encoding these properties separately and redundantly at both the static and dynamic levels. We attempt to address this problem by proposing a small set of type-directed syntax transformations which turn definitions of operations and relations from their functional form to their propositional form, and vice versa. Finally, we mention how these more specific techniques might be generalized to make programming in ATS easier, and discuss what the necessity of these transformations says about languages based on ATS.

2 Redundant Definitions

Our work addresses a particular kind of redundancy which arises when programming in languages based on ATS. If a programmer wants to reason about relatively simple, pure functions (such as operations on natural numbers, boolean values, or even lists) using proofs, she typically must define those functions at two separate levels. More generally, we can characterize this redundancy as the necessity to provide definitions at both the type and

term levels, e.g. if `nat` is both a kind and a type where `nat(x)` denotes the singleton type for some type-level `nat` expression `x`, we might have (in no particular language):

```
(+) : nat -> nat -> nat
plus : forall n:nat, forall m:nat, nat(n) -> nat(m) -> nat(m + n)
```

Note that in order to indicate that `plus` returns a singleton type equivalent to the result, we must separately define addition functions at both the type and term levels.

In the case of languages based on ATS, this manifests itself in a particular way. Here, the definitions can occur both as relations (or propositions) which can be used to construct proofs, and as dynamic functions which compute the result (and, if need be, simultaneously generate a proof about some property of that result). This means that the return type of `plus` from the above example cannot be `nat(m + n)` unless `plus` inductively constructs a proof of this fact, which means the actual return type would look something like $\exists$`s.(s = m + n | nat(s))`, where `s = m + n` can be viewed as a ternary predicate on three naturals. However, this is not so dissimilar – just as in the example before it, we must define this predicate (which is itself closely related and can arguably be derived from the pure addition function) at the type level. Note that we ignore the use of a constraint solver by ATS to reason about types indexed by simple arithmetic operations on integers, and instead assume that all such reasoning must be done using proofs supplied by the programmer. This allows us to focus on simpler examples while at the same time addressing the more general problem of allowing users to reason about programs using proof objects which they define themselves.

3 Type-directed Syntax Transformations

We experiment with type-directed syntactic transformations as a way to reduce the amount of redundant work a programmer must do when using a language such as ATS. Our strategy is to first define a simple, pure language based on the language found in DML at the type level [XP99]:

$$\begin{array}{rrcl} \text{kinds} & \kappa & ::= & \text{Nat} \mid \text{Bool} \mid \kappa_1 \to \kappa_2 \ldots \\ \text{pure expressions} & \tau & ::= & \text{Zero} \mid \text{Succ} \mid \text{NatInd} \mid \text{True} \mid \text{False} \mid \text{BoolInd} \\ & & & \mid x \mid \lambda x : \kappa.\tau \mid \tau_1\tau_2 \mid \ldots \end{array}$$

By "..." we indicate that the above can be viewed as a subset of a language which allows us to define pure functions at the type level. The types of the constructors are as expected: True : Bool, False : Bool, Zero : Nat, Succ : Nat $\to$ Nat, BoolInd : $\kappa \to \kappa \to$ Bool $\to \kappa$, and NatInd : $(\kappa \to \kappa) \to \kappa \to$ Nat $\to \kappa$.

We can now view this as the "source" language for describing pure functions. It is easy to embed this small language of pure terms inside Coq, and we take advantage of this by

using Coq to perform type checking on these terms before performing transformations, and sometimes (when we generate Coq terms) afterwards. Transformations which translate the pure functions into ATS propositions and corresponding ATS dynamic terms can then be applied to the pure code written in this language.

The focus of this work was primarily on performing these transformations to ATS, as this process is not straightforward. In the future, it should be possible to define a full language which allows reflection of pure programs at the type level to the term level, and then to utilize what we learned from encoding these transformations to translate the full language into ATS.

3.1 Propositions from Pure Functions

Pure functions, the bodies of which are expressions from a subset of the static expression language described above, can be transformed automatically into propositions of the form found in ATS. We can also transform such expressions into Coq syntax for easier type checking of the transformation process. For a very simple example, consider the pure function `eqZero` on naturals:

```
Definition eqZero (n:Nat) : Bool
  := NatInd Bool (fun (r:Bool) => False) True n.
```

We generate the following inductive definition of the corresponding proposition:

```
Inductive PeqZero : Nat -> Bool -> Prop :=
  | eqZeroBas : PeqZero (Zero) (True)
  | eqZeroInd : forall n:Nat,
                forall r:Bool,
                  PeqZero (n) (r) -> PeqZero (Succ n) (False).
Implicit Arguments eqZeroInd [n r].
```

For any pure function whose body consists of simple expressions which do not contain any lambda abstractions or instances or NatInd and BoolInd, only a single proposition constructor is generated, for example:

```
Definition true (b:Bool) : Bool := True.

Inductive Ptrue : Bool -> Bool -> Prop :=
  | trueBas : forall b:Bool, Ptrue (b) (True).
```

A function on n arguments generates a proposition on $n + 1$ arguments. If the body of the pure function consists of a NatInd or BoolInd expression, two cases are generated. For example, consider `plus` on naturals:

```
Definition plus (n:Nat) (m:Nat): Nat
  := NatInd Nat (fun (r:Nat) => Succ r) m n.
```

Here, for the variable `n` over which induction occurs, we have two cases, `Zero` and `Succ`:

```
NatInd Nat (fun (r:Nat) => Succ r) m Zero      ~~~> (Zero, m, m)
NatInd Nat (fun (r:Nat) => Succ r) m (Succ n)  ~~~> (n, m, r) -> (Succ n, m, Succ r)
```

This leads to two constructors for the generated proposition `Pplus`:

```
Inductive Pplus : Nat -> Nat -> Nat -> Prop :=
  | plusBas : forall m:Nat, Pplus (Zero) (m) (m)
  | plusInd : forall n:Nat,
              forall m:Nat,
              forall r:Nat,
                Pplus (n) (m) (r) -> Pplus (Succ n) (m) (Succ r).
Implicit Arguments plusInd [n m r].
```

Once the constructors for the propositions are constructed, any free variables (that is, arguments) which occur in the type of each constructer are universally quantified. Any variables which are universally quantified and occur in the types of the arguments to the second (inductive) constructor are marked implicit for convenience when testing.

Currently, it is not possible to transform expressions with arbitrary lambda abstractions, and we expect the function arguments to the NatInd and BoolInd primitives to be of the form `fun (r:t) => f r` where `f` is some previously defined pure function for which a proposition has already been generated, and `t` is the explicit type for the argument. Thus, chains of dependent pure functions can be generated in order. For example, consider `mult`, when defined after `plus`:

```
Definition mult (n:Nat) (m:Nat) : Nat
  := NatInd Nat (fun (r:Nat) => plus m r) Zero n.
```

Effectively, we first generate the proposition

```
Inductive Pmult : Nat -> Nat -> Nat -> Prop :=
  | multBas : forall m:Nat, Pmult (Zero) (m) (Zero)
  | multInd : forall n:Nat,
              forall m:Nat,
              forall r:Nat,
                Pmult (n) (m) (r) -> Pmult (Succ n) (m) (plus m r).
```

We then replace this occurrence of a function call (in this case, `plus`) inside the proposition type with a result variable, and introduce new argument propositions (in this case, `Pplus`):

```
Inductive Pmult : Nat -> Nat -> Nat -> Prop :=
  | multBas : forall m:Nat, Pmult (Zero) (m) (Zero)
  | multInd : forall n:Nat,
              forall m:Nat,
              forall r:Nat,
              forall r':Nat,
                Pmult (n) (m) (r) -> Pplus (m) (r) (r') -> Pmult (Succ n) (m) (r').
Implicit Arguments multInd [n m r r'].
```

Notice that the order in which we insert new arguments is important if we eventually hope to generate in a straightforward way dynamic terms using these propositions. For example, a function would need to first return `Pmult (n) (m) (r)` before we could call another function (here, `Tplus`, which we encounter further below) which could then return `Pplus (m) (r) (r')`.

3.2 ATS Dynamic Terms from Pure Functions and Propositions

Any collection of pure function which can be used to generate proposition definitions can also be used to generate dynamic ATS terms which build results along with a proposition exactly describing that result. When we apply the transformation to the above examples, we once again obtain the propositions, this time in ATS syntax:

```
dataprop Pplus (Nat, Nat, Nat) =
  | {m:Nat} plusBas (Zero, m, m)
  | {n:Nat, m:Nat, r:Nat}
        plusInd (Succ n, m, Succ r) of Pplus (n, m, r)

dataprop Pmult (Nat, Nat, Nat) =
  | {m:Nat} multBas (Zero, m, Zero)
  | {n:Nat, m:Nat, r:Nat, r':Nat}
        multInd (Succ n, m, r') of (Pmult (n, m, r), Pplus (m, r, r'))
```

These `dataprop` definitions are exactly like the Coq definitions seen above. Next, for functions defined using constants, only a single case is generated, but for ones using induction principles, case expressions are generated. For each case, the corresponding proof term is constructed – in the inductive case, we must introduce an intermediate value to apply a constructor to the proofs which are generated by the recursive call:

```
fun Tplus {n:Nat, m:Nat} (n': Nat(n), m': Nat(m)) : `(Pplus (n, m, r) | Nat(r))
```

```
 = case n' of
     | Zero => `(plusBas m | m')
     | Succ n'' =>
         let val `(pf | r) = Tplus n'' m
         in
            `(plusInd pf | Succ r)
         end

fun Tmult {n:Nat, m:Nat} (n': Nat(n), m': Nat(m)) : `(Pmult (n, m, r) | Nat(r))
 = case n' of
     | Zero => `(multBas m | Zero)
     | Succ n'' =>
         let val `(pfmult | r) = Tmult n'' m'
             val `(pfplus | r') = Tplus m' r
         in
            `(multInd (pfmult, pfplus) | r')
         end
```

This transformation is straightforward to perform so long as every pure function at the type level is named and a corresponding proposition has been generated for it.

3.3 ATS Boolean Functions from Propositions

For the simplified propositions generated in Section 3.1, it is easy to generate dynamic terms which are boolean functions. We take advantage of the fact that all arguments to the propositions being constructed are constructors. We also assume there exists a boolean function `eq`*TypeName* for every possible type which may arise as an argument to a proposition constructor, and use it when necessary. For example, suppose we are given:

```
dataprop Pplus (Nat, Nat, Nat) =
  | {m:Nat} plusBas (Zero, m, m)
  | {n:Nat, m:Nat, r:Nat} plusInd (Succ n, m, Succ r) of Pplus (n, m, r).
```

It is possible to generate two cases, plus a third default case used to reject inputs which do not satisfy the predicate:

```
fun TPplus (n, m, r)
 = case (n, m, r) of
     | (Zero, m, m) => True andalso (eqNat m m)
     | (Succ n, m, Succ r) => TPplus (n, m, r)
     | (_, _, _) => False
```

The disadvantage of obtaining boolean functions in this way is that the boolean value returned is not paired with a proof. It often makes more sense to define the boolean function as a pure function at the type level and perform the usual transformations described in the two previous sections to obtain a term-level function which returns the actual proof along with `True` or `False`.

Furthermore, this transformation cannot be performed easily when propositions depend on other propositions. We might easily look again at the `Pmult` case:

```
dataprop Pmult (Nat, Nat, Nat) =
  | {m:Nat} multBas (Zero, m, Zero)
  | {n:Nat, m:Nat, r:Nat, r':Nat}
        multInd (Succ n, m, r') of (Pmult (n, m, r), Pplus (m, r, r'))
```

We cannot simply pattern match on the arguments because there is no way to perform "subtraction" on `r'` to recover `r`, and thus no way to make the inductive call to `TPmult` with the third argument as `r`.

4 Further Work

As mentioned earlier, defining a language in which programs at the type level can be reflected at the term level and then using these transformations to translate programs from this language into ATS propositions and terms is one obvious next step. Introducing user-defined data sorts into this language would make things interesting, as for any user-defined data sort, it is possible to automatically generate an induction principle corresponding to NatInd and BoolInd. By defining a more general transformation on arbitrary induction principles, it would then be possible to generate propositions and dynamic terms which deal with all user-defined data sorts. To make the transformations more general, it should be possible to transform code with anonymous lambda abstractions into a series of let bindings. However, the propositions formed in this way would be almost meaningless to a programmer, so this does not seem to be a practical extension.

5 Conclusions

Resorting to transformations can in many cases be a serious limitation. For example, we first defined both `plus` and `mult` as pure functions in our examples in Section 3.1, then applied the transformation all at once to the chain of functions. However, a programmer might want to first define `plus` as a pure function, apply the transformation to obtain a function `Tplus` which produces a proof, and then define a pure `mult` function at the term level which uses

`Tplus`. Currently, this would be impossible due to the phase distinction. This is, at best, an inconvenience.

Another concern is that for more complex pure functions which depend on many other pure functions, the propositions constructed will be impractically large and difficult to read. However, if programmers are expected to apply syntactic transformations to definitions of pure functions to generate definitions of corresponding propositions, the understandability of these definitions is essential, as the programmers are then expected to use these proposition constructors within their dynamic function definitions. Furthermore, these proposition are unfortunately, as far as the language is concerned, completely separate from the pure functions, and thus it is arguable that this will already be somewhat unnatural for programmers. We must also consider the relative difficulty of generating propositions in full generality (as opposed to the special cases covered above), especially for arbitrary user-defined data sorts. When all this is taken into account, it raises serious concerns about the feasibility and scalability of programming with proofs under the ATS paradigm, and whether having a more flexible language at the type level coupled with a proof-writing system similar to that of Coq is not a better option.

References

[CX04] Chiyan Chen and Hongwei Xi. Ats/lf: a type system for constructing proofs as total functional programs. November 2004.

[CX05] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *Proceedings of the 10th International Conference on Functional Programming (ICFP05)*, September 2005.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.