

# Eliminating Specificational Data from Untyped Areas in OPTT

Todd Schiller

May 6, 2008

## Abstract

Operational Type Theory (OPTT) permits the combination of external and internal verification by maintaining a dichotomy between programs, formulas, and types. Specificational data is data that is necessary for dependent types and their reasoning, but not for program evaluation and determining operational joinability. In this paper, a procedure is presented for eliminating the presence of specificational data in untyped positions in the OPTT, thereby lifting the burden of reasoning about the data in these positions. We then discuss the advantages of removing specificational data from untyped areas using Guru, a machine-checked implementation of OPTT.

## 1 Introduction

Dependent data types are useful for program verification. However, the use of dependent types typically requires the presence of the extraneous data in computation as well. As such, there is a tradeoff between the ease of verifying certain properties about a program and the inefficiency of dealing with the extra data during computation. Moreover, the presence of the data in an evaluative context may actually increase the burden of proof during verification. Data that is necessary for dependent types and type checking, but is not used in an evaluative context, is called specificational data.

In this paper, a procedure is proposed for eliminating specificational data from untyped areas, while still maintaining the data for type checking, under Operational Type Theory (OPTT). By doing so, the computational inefficiencies that generally follow from the use of dependent types are removed.

In Section 2, a brief introduction to Operational Type Theory is offered. In Section 3, a procedure is described for removing type annotations from untyped areas. Sections 4 and 5 present the procedure for removing specificational data from non-evaluative contexts. In Section 6, an example result of implementing the procedure is given using Guru, a machine-check implementation of OPTT. Finally, in Section 7, the paper concludes with a general discussion of the results and future work.

$$\begin{aligned}
t & ::= x \mid c \mid \text{fun } x(\bar{x} : \bar{A}) : T. t \mid (t X) \mid \\
& \quad \text{cast } t \text{ by } P \mid \text{abort } T \mid \\
& \quad \text{let } x = t \text{ by } y \text{ in } t' \mid \\
& \quad \text{match } t \text{ by } x y \text{ with} \\
& \quad \quad c_1 \bar{x}_1 \Rightarrow t_1 \mid \dots \mid c_n \bar{x}_n \Rightarrow t_n \text{ end} \\
X & ::= t \mid T \mid P \\
A & ::= T \mid \text{type} \mid F \\
T & ::= x \mid d \mid ! \mid \text{Fun}(x : A). T \mid \langle T Y \rangle \\
Y & ::= t \mid T
\end{aligned}$$

Figure 1: Terms ( $t$ ) and Types ( $T$ )

## 2 Operational Type Theory

OPTT is founded on the observation that program evaluation, proof normalization evaluation, and definitional equality evaluation can be kept separate. As such, OPTT treats programs, proofs, and their classifiers separately.

OPTT permits internal and external verification under the aforementioned dichotomy through the use of computationally irrelevant casts. These casts are not considered during evaluation and formal reasoning. Instead, two terms are considered definitionally equal under the operational semantics if they converge to a common term.

### 2.1 Notation

An identical notation for the syntax of OPTT as in [1] is adopted. The context is denoted as  $\Gamma$ . Proofs are denoted as  $P$ , formulas as  $F$ , variables as  $x$ , term constructors as  $c$ , type constructors as  $d$ , terms as  $t$ , and types as  $T$ . An array of variables is denoted using  $\bar{x}$ . Similarly,  $\bar{T}$  denotes an array of types.

$E$  and  $I$  are used to denote evaluative contexts and inactive terms, respectively. Inactive expressions may include variables and `cast`-terms to accommodate proof checking, but these constructs cannot be present during evaluation.

The grammar for terms and types is shown in Figure 1.

### 2.2 Definitional Equality

In OPTT, proofs and type annotations are dropped during evaluation. Therefore, joinability under the operational semantics does not (and cannot) depend on proofs and type annotations. Instead, two expressions are joinable if they both converge to a common term. The call-by-value semantics of the language allow for the safe renaming of the bound variables within the expressions. Figure 2 shows the rules for small-step evaluation.

$$\begin{aligned}
E[(F \bar{I})] & \rightsquigarrow E[[\bar{I}/\bar{x}, F/x]t] \\
F \equiv \text{fun } x(\bar{x} : \bar{I}) : !.t & \\
\\
E[\text{match } (c_i \bar{I}) & \\
\text{by } x y \text{ with} & \\
c_1 \bar{x}_1 \Rightarrow s_1 \mid \dots \mid c_n \bar{x}_n \Rightarrow s_n & \\
\text{end}] & \rightsquigarrow E[[\bar{I}/\bar{x}_i]s_i] \\
\\
E[\text{let } x = I \text{ by } y \text{ in } t] & \rightsquigarrow E[[I/x]t] \\
\\
E[\text{abort } !] & \rightsquigarrow \text{abort } !
\end{aligned}$$

where:

$$\begin{aligned}
E & ::= * \parallel (E X) \parallel (I E) \parallel \\
& \quad \text{let } x = E \text{ by } y \text{ in } t \parallel \\
& \quad \text{match } E \text{ by } x y \text{ with} \\
& \quad \quad c_1 \bar{x}_1 \Rightarrow t_1 \mid \dots \mid c_n \bar{x}_n \Rightarrow t_n \text{ end} \\
\\
I & ::= x \parallel c \parallel ! \parallel T \parallel P \parallel (c I_1 \cdots I_n) \parallel \\
& \quad \text{cast } I \text{ by } P \parallel \\
& \quad \text{fun } x(\bar{x}_1 : A_1) \cdots (\bar{x}_n : A_n) : T. t
\end{aligned}$$

Figure 2: Small-step evaluation

While this formulation of definitional equality is essential for Guru to maintain the separation between types, formulas, and programs, it presents some challenges. Consider, for example, the arguments of term applications with their associated type information. When determining joinability, the associated type information cannot be considered, disallowing the conditional dropping of the data based on the information.

### 3 Removing type annotations from untyped areas

As with specificational data, type annotations are not necessary for evaluation and definitional equality. As such, type annotations are dropped during evaluation (according to the rules in Figure 3). The dropped annotations are represented using the special constant, !.

In untyped areas, specific types do not appear, only ! – as the annotations have been dropped. These dropped annotations are not necessary for computation under the call-by value semantics of OPTT. Therefore, type annotations can be eliminated (and any resulting !s) from untyped areas. The elimination occurs by changing the drop annotation rules for function terms to eliminate parameters that are of type `type`. Thus the dropped types never need to be instantiated (substituted into) in the body of the function term. Term applications which have the function term as their head no

$$\begin{array}{lcl}
\text{fun } x(\bar{x} : \bar{A}^-) : T . t & \Rightarrow & \text{fun } x(\bar{x} : \bar{!}) : T . t \\
\text{fun } x(\bar{x} : \bar{A}) : T^- . t & \Rightarrow & \text{fun } x(\bar{x} : \bar{A}) : ! . t \\
P^- & \Rightarrow & ! \\
(t T^-) & \Rightarrow & (t !) \\
\text{cast } t \text{ by } P & \Rightarrow & t \\
\text{abort } T^- & \Rightarrow & \text{abort } !
\end{array}$$

Figure 3: Dropping Annotations

longer use the dropped types as arguments.

It should be noted that removing the type annotations can result in some term applications being evaluated that otherwise would not have been. Consider the Guru function which takes a single type as an argument:

```
Define genZ := fun(A:type) : nat.
  Z.
```

Without extending the language to remove type annotations from untyped areas,  $(\text{genZ}) \ Z$  would not be definitionally equal to  $Z$  since  $(\text{genZ})$  is a function taking in a type as an argument. However,  $(\text{genZ } !)$  and  $Z$  would be definitionally equal. When type annotations are removed from the untyped areas,  $(\text{genZ})$  is equal to  $Z$  because  $(\text{genZ})$  evaluates to  $Z$ .

## 4 Marking data as specificational

There are only two areas in which an expression can be explicitly marked as specificational: in the parameter list of a function term and the parameter list of a function type (for constructor definitions). As such, only variables can be explicitly marked as specificational.

### 4.1 Function term parameters

When a function term is created, the parameters that have been explicitly marked as specificational are recorded. The record is stored in the function term construct so that it can be referred to when the function term is used as the head of a term application. Those variables are then marked as specificational in the context so that the specificational status of the variables in the body of the function can be determined by polling the context.

### 4.2 Function type parameters

Parameters may also be explicitly marked as specificational in the function types that appear as constructors in inductive definitions. As with function term parameters, whichever parameters were marked as specificational in the function type construct are recorded. Similarly, the variables are marked as specificational in the context.

### 4.3 Term application arguments

A term application can be used in an untyped area. Therefore, the specificational status of the arguments must be known so that they may be dropped when checking for definitional equality. While the specificationality of a given variable in the current context is known (by polling the context), there is no way to implicitly deduce the specificationality of an argument expression when in an untyped area. Recall, from Section 2.2, that the definitional equality used by OPTT prohibits the use of typing when dropping annotations. Instead, the specificational status of the parameters for which the arguments are being passed is used.

The specificational status of the parameters can be determined by classifying the head of the term application. This information is then stored in the term application construct.

## 5 Checking specificational data usage

In order to properly drop specificational data, it must be established that the data is not required for computation – that is does not appear in an evaluative context. In OPTT, evaluative contexts can contain only four constructs: term applications, `match`-terms, `abort`-terms, and `let`-terms. As per the grammar, `abort`-terms are specified only with a type, and therefore cannot contain any specificational data. Therefore the usage of specificational data must only be verified in term applications, `match`-terms, and `let`-terms.

A specificational data usage check is performed for each function term (`fun`-term) that is defined after classification, since instances of their bodies will be evaluated when the functions are called. The check fails if the body of the function term depends on specificational data. Since classification occurs before the check, the specificational data has been properly marked.

Consider an attempt in Guru at a specificational identity function for natural numbers:

```
Define bad := fun(spec x:nat) : nat .  
  x.
```

The specificational usage check fails because the variable `x` was marked as specificational during classification but is in the return position of the function term.

### 5.1 Scrutinee of match terms

`Match`-terms direct the flow of evaluation according to the pattern of a term (the scrutinee). As the scrutinee is required for evaluation of the `match`-term, the scrutinee cannot depend on specificational data. For example, the following Guru code is an invalid use of specificational data:

```
Define bad := fun(spec x :nat)(y:nat) : nat .  
  match x by u v return nat with  
    Z => Z
```

```
| S x' => y
end.
```

## 5.2 Non-specificational arguments of term applications

If a term application is in an evaluative context, it must be verified that the head and the arguments properly use specificational data. During classification, the specificational status of the parameters for which the arguments are being supplied was set. For each of the arguments, a check that the argument properly utilizes specificational data must occur. If the argument can be specificational (it is being supplied for a specification parameter), no analysis of the expression's specificational data usage is required. However, if the argument cannot be specificational, the specificational data usage in the expression is checked to ensure the expression does not depend on specificational data.

The following Guru code is invalid because the specificational variable  $x$  is being used as an argument for the first parameter of `plus`, which is non-specificational:

```
Define bad := fun(spec x:nat) (y:nat): nat.
            (plus x y).
```

## 5.3 Let-terms

Let-terms are evaluative in that they assign the result of a computation to a variable. The elements of the assign statement must be checked for proper specificational data usage (that the result is not dependent upon specificational data). Then, it must be verified that the body of the let-term, the terms which depend on the let-term, also use specificational data properly. In general, for a let-term of the form

```
let x = I by y in t
```

$x$ ,  $I$ , and  $t$  cannot contain specificational data ( $y$  is a proof that  $x = I$ ).

# 6 Specificational data and polymorphic lists indexed by length

In this section, an example is presented of the use of specificational data in an extension to Guru, a machine-checked implementation of OPTT. The benefits of specificational data are demonstrated in the context of proving properties of functions which operate on polymorphic lists indexed by length.

## 6.1 Inductive definition

In Guru, an inductive data type for a polymorphic list indexed by length is created as follows:

```

Inductive llist : Fun(A:type) (n:nat).type :=
llistn : Fun(A:type).<llist A Z>
| llistc :
  Fun(A:type) (a:A) (n:nat) (l:<llist A n>).
  <llist A (S n)>.

```

As shown, the type `llist` has two constructors: a constructor `llistn` to create an empty list and a constructor `llistc` to construct a list by adding an element to the head of another list. For example, a list, `l`, of length `n` containing elements of type `A` has the type `<llist A n>`. The list constructed by adding another element, `x`, of type `A` to list `l` will result in a list of type `<llist A (S n)>`, where `S` is the successor function. Using the two constructors, arbitrary lists can be constructed (by starting with an empty list).

While the size of the list is necessary to maintain the dependent type of the list, the size of the list is not needed for many computations with lists (such as appending two lists or reversing a list). Therefore, it would be useful to mark the length of the list as specification. With the presented extension of Guru, this is achieved by annotating the parameter in term constructor with `spec`:

```

Inductive llist : Fun(A:type) (n:nat).type :=
llistn : Fun(A:type).<llist A Z>
| llistc :
  Fun(A:type) (a:A) (spec n:nat) (l:<llist A n>).
  <llist A (S n)>.

```

## 6.2 Appending polymorphic lists indexed by length

A function to append two `llists` together can be defined as follows:

```

Define append :=
fun append(A:type) (n m:nat)
  (l1 : <llist A n>) (l2 : <llist A m>):
  <llist A (plus n m)>.
  match l1 by u v
  return <llist A (plus n m)> with
  llistn B => cast l2 by
    cong <llist A *>
    symm trans
    cong (plus * m)
    inj <llist ** *> v
    join (plus Z m) m
| llistc A' x n' l1' =>
  cast
  (llistc A' x (plus n' m)
  (append A' n' m l1'
  cast l2 by cong <llist * m>
  inj <llist * **> v))
  by trans

```

```

    cong <llist * (S (plus n' m))>
      symm inj <llist * **> v
  cong <llist A *>
    trans
      symm join (plus (S n') m)
        (S (plus n' m))
      cong (plus * m)
        symm inj <llist ** *> v
end.

```

The above `append` function will work with both the non-specificational and specificational definition of `llist` as the function takes as arguments two non-specificational `nats` for the lists length. However, if the specificational version of `llist` is used, it would be favorable for `append` to only require non-specificational data for the list lengths, since they are not needed for computation. The advantages of using specificational data for the `append` function are realized by annotating its parameter list:

```

Define append :=
fun append(A:type) (spec n m:nat)
  (l1 : <llist A n>) (l2 : <llist A m>):
  <llist A (plus n m)>

```

The new function only differs by the inclusion of the `spec` annotation. However, an attempt to define this function while using the non-specificational definition of `llist` would fail specification checking, as the `(plus n' m)` argument being passed to the `llistc` constructor for the new lists length is constructed using specificational data (and the length parameter in the constructor is non-specificational).

Denoting the list lengths as specificational indicates that the lengths are not needed to evaluate `append`. As such, performing computations with them during evaluation (including recursively adding the two list lengths together) can be avoided resulting in greater efficiency. Moreover, the specificational data does not need to be passed in as arguments during evaluation, since it cannot be used computationally.

### 6.3 Proving the associativity of `append`

When using non-specificational data, the proof of the associativity of the `append` function is constructed as follows in `Guru`:

```

Define append_assoc : Forall(A:type)
  (n1 : nat) (l1 : <llist A n1>)
  (n2 n3 : nat) (l2 : <llist A n2>) (l3 : <llist A n3>).
{ (append ! (plus n1 n2) n3
  (append ! n1 n2 l1 l2) l3) =
  (append ! n1 (plus n2 n3)
    l1 (append ! n2 n3 l2 l3)) } :=
forallli(A:type) .
induction(n1:nat) (l1:<llist A n1>)
  by x1 x2 IH return Forall(n2 n3 : nat)
    (l2 : <llist A n2>) (l3 : <llist A n3>).

```

```

      { (append ! (plus n1 n2) n3
        (append ! n1 n2 l1 l2) l3) =
        (append ! n1 (plus n2 n3)
          l1 (append ! n2 n3 l2 l3)) } with
l1listn A' => foralli(n2 n3 : nat)
  (l2 : <l1list A n2>)(l3 : <l1list A n3>).
  trans cong (append ! (plus n1 n2) n3
    (append ! n1 n2 * l2) l3)
    x1
  trans cong (append ! (plus * n2) n3
    (append ! * n2 (l1listn !) l2) l3)
    inj <l1list ** *> x2
  trans join (append ! (plus Z n2) n3
    (append ! Z n2 (l1listn !) l2) l3)
    (append ! n1 (plus n2 n3)
      (l1listn !) (append !
        n2 n3 l2 l3))
  symm cong (append ! n1 (plus n2 n3)
    * (append ! n2 n3 l2 l3)) x1
| l1listc A' x' n1' l1' =>
  foralli(n2 n3 : nat)
    (l2 : <l1list A n2>)(l3 : <l1list A n3>).
    trans cong (append ! (plus n1 n2) n3
      (append ! n1 n2 * l2) l3) x1
    trans cong (append ! (plus n1 n2) n3 * l3)
      join (append ! n1 n2 (l1listc ! x' n1' l1') l2)
        (l1listc ! x' (plus n1' n2)
          (append ! n1' n2 l1' l2))
    trans join (append ! (plus n1 n2) n3
      (l1listc ! x' (plus n1' n2)
        (append ! n1' n2 l1' l2)) l3)
      (l1listc ! x' (plus (plus n1' n2) n3)
        (append ! (plus n1' n2) n3
          (append ! n1' n2 l1' l2) l3))
    trans cong (l1listc ! x' (plus (plus n1' n2) n3) *)
      [IH n1' cast l1' by cong <l1list * n1'>
        inj <l1list * **> symm x2
          n2 n3 l2 l3]
    symm trans cong (append ! n1 (plus n2 n3)
      * (append ! n2 n3 l2 l3)) x1
    trans join (append ! n1 (plus n2 n3)
      (l1listc ! x' n1' l1') (append ! n2 n3 l2 l3))
      (l1listc ! x' (plus n1' (plus n2 n3))
        (append ! n1' (plus n2 n3) l1'
          (append ! n2 n3 l2 l3)))
    cong (l1listc ! x' *
      (append ! n1' (plus n2 n3) l1'
        (append ! n2 n3 l2 l3)))
      symm [plus_assoc n1' n2 n3]
end.

```

A proof that `plus` is associative is required as the `append` function tracks the size of the lengths during evaluation (the `plus_assoc` proof is called in last line of the inductive case in the above proof). However, if tracking the size of the list during evaluation is not needed, reasoning about the size of the lists (and thus the associativity of `plus`) is not needed either. Therefore, when using the specificational definitions of `l1ist` and `append`, the proof is significantly simpler to reflect the reduced burden of proof:

```

Define append_assoc : Forall(A:type)
  (n1 : nat) (l1 : <l1ist A n1>)
  (n2 n3 : nat) (l2 : <l1ist A n2>) (l3 : <l1ist A n3>).
  { (append (append l1 l2) l3) =
    (append l1 (append l2 l3)) } :=
foralli(A:type) .
  induction(n1:nat) (l1:<l1ist A n1>)
    by x1 x2 IH return Forall(n2 n3 : nat)
      (l2 : <l1ist A n2>) (l3 : <l1ist A n3>).
      { (append (append l1 l2) l3) =
        (append l1 (append l2 l3)) } with
  l1listn A' => foralli(n2 n3 : nat)
    (l2 : <l1ist A n2>) (l3 : <l1ist A n3>).
    trans cong (append (append * l2) l3) x1
    trans join (append (append (l1listn) l2) l3)
      (append (l1listn) (append l2 l3))
    symm cong (append * (append l2 l3)) x1
| l1listc A' x' n1' l1' => foralli(n2 n3 : nat)
  (l2 : <l1ist A n2>) (l3 : <l1ist A n3>).
  trans cong (append (append * l2) l3) x1
  trans cong (append * l3)
    join (append (l1listc x' l1') l2)
      (l1listc x' (append l1' l2))
  trans join (append (l1listc x' (append l1' l2)) l3)
    (l1listc x' (append (append l1' l2) l3))
  trans cong (l1listc x' *)
    [IH n1' cast l1' by cong <l1ist * n1'>
      inj <l1ist * **> symm x2
      n2 n3 l2 l3]
  trans join (l1listc x' (append l1' (append l2 l3)))
    (append (l1listc x' l1') (append l2 l3))
  symm cong (append * (append l2 l3)) x1
end.

```

## 7 Conclusion

A process for removing dropped type annotations from untyped areas in OPTT has been presented. While the removal changes the operational behavior for a subset of programs, it eliminates the need for `!`s in Guru.

Furthermore, a procedure was described for marking specificational data and subsequently dropping the specificational data in untyped areas. By extending the method for removing dropped type annotations from untyped areas, the dropped specificational data is removed from the untyped areas as well.

By eliminating the specificational data from the untyped areas, the burden of proof is reduced for the programmer and extraneous evaluation during interpretation is eliminated.

## 7.1 Future Work

Since the Guru interpreter has been extended to drop specificational data, the next step would be to extend the compiler as well. By adding specification data support to the compiler, the passing of specificational data and the corresponding computations can be avoided. This extension will give Guru code with dependent types performance similar to comparable compiled programs without dependent types.

The compiler can be extended by using the specificationality of terms to determine whether code should be emitted. Arguments to specificational parameters in term applications ought not be emitted. Similarly, computations involving specificational expressions in the bodies of functions should not have their corresponding code emitted.

## References

- [1] A. Stump, E. Westbrook, and T. Simpson. Operational type theory. 2008. under review.