ART Memo 26 (Version 0)
Thoughts on Mixing Traced and Untraced Computation

Olaf Chitil

November 1, 2001

1 Introduction

Even when a computation of a program is traced, it shall be possible to name some parts (modules) of the program as trusted. The parts of a computation performed by trusted program parts shall not be traced. Motivation:

- efficiency: shorter run-time and less trace data;
- comprehensibility of the trace: avoid irrelevant trace data.

Furthermore, untraced program parts should not need to be transformed at all. Instead, an interface that wraps untraced code for use by traced code should be produced for an untraced program part. Motivation:

- efficiency: shorter compile-time and less object code;
- portability: do not need source of prelude or compiler-specific libraries to transform them.

In the traced computation each traced (sub)expression needs to carry additional tracing information. Hence the tracing transformation also transforms all type definitions. So traced and normal code inhabit different types between which the wrapping interface needs to translate.

2 Problems with Transforming Untraced Code

Hat 1.10 transforms untraced code as well (in a different way than traced code). A problem with it besides efficiency and portability is that even untraced computation produces some trace fragments. All untraced values are recorded in the trace, just in case that they become part of the result of an untraced function.

Possibly, the combinators of untraced program parts could be made non-strict in the trace information, so that the trace of the result of an untraced function is only written to file when the result reaches the traced world. Deferred trace writing may increase heap space usage linearly. Alternatively, wrappers could add the trace information for expressions constructed by an untraced function (marked with special trace info) and keep the trace information of threaded expressions.

3 The Problem with Wrapping Classes

For every class in untraced code there must be a respective traced class, because instances may be defined in traced code. That is straightforward. There is also no problem with classes defined in traced code, because they can only be used by traced code.

The problem is defining wrappers for untraced overloaded variables. A variable is not parametrically polymorphic in the type variables that appear in its type context. Hence arguments/results of these type variable types have to be unwrapped/wrapped. But the concrete types are not known at the call sites of the wrappers.

So how define wrappers for example for:

```
elem :: Eq a => a -> [a] -> Bool
fmap :: Functor f => (a -> b) -> (f a -> f b)  -- is a method
fromIntegral :: (Integral a, Num b) => a -> b
```

3.1 Solution 1: A class for every overloaded variable

Wrappers for untraced *methods* are still easy to define: For every instance in untraced code there must a respective traced instance of the traced class and traced data type. The methods are defined as wrappers to the untraced instance.

```
tfmap :: SR -> Trace -> R (Fun (Fun a b) (Fun (f a) (f b)))
tfmap s t = fun2 "fmap" wfmap s t

class TFunctor f where
  wfmap :: Trace -> R (Fun a b) -> R (f a) -> R (f b)

instance TFunctor TList where
  wfmap t f x = wrapList t (fmap (unwrapFun id id f) (unwrapList x)
```

We can define a wrapper for an overloaded non-method variable such as `elem` by defining a class for it and defining an instance for all types for which instances for its context exist. For example

```
telem :: TElem a => SR -> Trace -> R (Fun a (Fun (TList a) TBool))
telem s t = fun2 "elem" welem s t

class TEq a => TElem a where
  welem :: Trace -> R a -> R (TList a) -> R TBool

instance TElem TInt where
  welem t x xs = wrapBool t (elem (unwrapInt x) (unwrapList unwrapInt xs)
```

However, for overloaded variables with more than one type variable in the context, for example `fromIntegral`, we would need multi-parameter classes.

Also the transformation would introduce a large number of new classes and instances and transformation of type declarations of variables requires knowledge about the definitions of these variables.

3.2 Solution 2: Classes with Functional Dependencies

We introduce a two-parameter class with functional dependencies:

```
class Wrapping a b | a->b, b->a where
  wrap :: Trace -> a -> R b
  unwrap :: R b -> a
```

Then wrappers of overloaded variables and methods can be defined without further classes and instances, for example:

```
telem :: (Wrapping a b, Eq b)
      => SR -> Trace -> R (Fun a (Fun (TList a) TBool))
telem s t = fun2 "elem" welem s t

welem :: (Wrapping a b, Eq b)
      => Trace -> R a -> R (TList a) -> R TBool
telem s t x xs = wrap t (elem (unwrap x) (unwrap xs))
```

We notice that `unwrap` needs a trace argument for unwrapping a function (applications of the traced function will be recorded in the trace and hence so will be the arguments).

```
class Wrapping a b | a->b, b->a where
  wrap :: Trace -> a -> R b
  unwrap :: Trace -> R b -> a

newtype Fun a b = Fun (Trace -> R a -> R b)

instance (Wrapping a c, Wrapping b d) => Wrapping (a->b) (Fun c d) where
  wrap t f = R (Fun \x -> wrap t (f (unwrap t x))) t
  unwrap t (R (Fun f) _) = \x -> unwrap t (f (wrap t x))
```

It becomes clear that the whole wrapping can be done by a single function call.

```
telem :: (Wrapping a b, Eq b)
      => SR -> Trace -> R (Fun a (Fun (TList a) TBool))
telem s t = wrap (mkFun "elem" s t) elem
```

For constructor classes we obtain non-simple contexts. For example:

```
tfmap :: (Wrapping (f a) (g c), Wrapping (f d) (g b), Functor g)
      => SR -> Trace -> R (Fun (Fun a b) (Fun (f a) (f b)))
tfmap s t = wrap (mkFun "fmap" s t) fmap
```

However, this `tfmap` also unwraps/wraps the components `b` of the type constructor. To avoid this (un)wrapping and to preserve parametricity we need a wrapping class for each kind. Two functions are passed for each type argument, because the type constructor can be co-, contra-variant or both in each argument. For example:

```
class Wrap(*->*->*) f g | f->g, g->f where
  wrap(*->*->*) :: Trace -> (a->c) -> (c->a) -> (b->d) -> (d->b)
                -> f a b -> R (g c d)
```

```
  unwrap(*->*->*) :: (a->c) -> (c->a) -> (b->d) -> (d->b)
                  -> R (g c d) -> f a b

instance Wrap(*->*->*) (->) Fun where
  wrap(*->*->*) t w1 u1 w2 u2 f = R (Fun (\t' x -> w2 (f (u1 x)))) t
  unwrap(*->*->*) w1 u1 w2 u2 (R (Fun g) _) = \x -> u2 (f Root (w1 x)))

class Wrap(*->*) f g | f->g, g->f where
  wrap(*->*) :: Trace -> (a->b) -> (b->a) -> f a -> R (g b)
  unwrap(*->*) :: (a->b) -> (b->a) -> R (g b) -> f a

instance Wrap(*->*) [] TList where
  wrap(*->*) t w u [] = R TNil t
  wrap(*->*) t w u (x:xs) = R (TCons (w x) (wrapW t w xs)) t
  unwrap(*->*) w u (R TNil) = []
  unwrap(*->*) w u (R (TCons x xs)) = u x : unwrapW u xs

tfmap :: (Wrap(*->*) f g, Functor g)
      => SR -> Trace -> R (Fun (Fun a b) (Fun f a) (f b)))
tfmap s t = let t' = mkFun "fmap" s t in
              wrap(*->*->*) t'
                undefined
                (unwrap(*->*->*) id undefined undefined id)
                (wrap(*->*->*) t' undefined (unwrap(*->*) undefined id)
                  (wrap(*->*) t' id undefined) undefined)
                undefined
                fmap
```

The many arguments and calls of wrappers might be expensive, but optimising them (inlining and reduction) is the task of the compiler. By hand we can obtain

```
tfmap s t = fun2 "fmap" wfmap s t

wfmap :: (Wrap(*->*) f g, Functor g)
      => Trace -> R (Fun a b) -> R (f a) -> R (f b)
wfmap t (R (Fun f) _) (R x _) =
  wrap(*->*) t id undefined
    (fmap (\z -> unR (f undefined (R z undefined)))
      (unwrap(*->*) undefined id))
```

Wrappers for type arguments are either `id` or standard wrappers. So instead of passing wrappers around, we might want to pass around a data structure that encodes which parts have to be wrapped/unwrapped. However, it is unclear how to type such wrappers:

```
class Wrapp(*->*->*) f g | f->g, g->f where
  wrap :: Trace -> ? c d ? -> f a b -> R (g c d)
  unwrap :: Trace -> ? c d ? -> R (g c d) -> f a b
```

Alternatively, overlapping instances might enable avoiding (un)wrapping and preservation of parametricity using only a single wrapping class.

3.3 Solution 2': Two-Parameter Class with Non-Simple Instances

```
class Trans a b where
  trans :: Trace -> a -> b

instance (Trans c a, Trans b d) => Trans (a->b) (R (Fun c d)) where
  trans t f = R (Fun \t' x -> trans t' (f (trans t x))) t

instance (Trans a c, Trans d b) => Trans (R (Fun c d)) (a->b) where
  trans t (R (Fun f) _) = \x -> trans t (f (trans t x))

instance Trans (a->b) (a->b) where
  trans t f = f
```

I don't know if resulting ambiguity problems can be resolved by type annotations.

4 Problems with Wrapping Data Structures

Arguments of normal functions need to be of normal types. Hence a wrapper strips off all added tracing information. However, the normal function may construct part of its result from parts of its arguments. In the tracing world such a threaded expression should carry the same tracing transformation as part of the result as as part of an argument. As Colin suggested, the function that wraps the result could (impurely) test if its argument is an unevaluated application of unwrap to a traced expression. If that is the case, then the traced expression would be returned, preserving the original tracing information of the threaded expression.

However, a normal function such as "smallestTree" could traverse arguments completely and then return one of the arguments. The argument traversal removes all tracing information so that the (impure) wrapper could not re-attach it.

5 Problems with Wrapping Based on Interface Files

Interface files are not standardised at all. In practise there seems to be some information interface files of all compilers cover: variables with their type signature; type definitions, possibly without (some) data constructors if not exported; class definitions, without default definitions; heads of instance definitions; fixity information for variables and data constructors.

Interface files seem to list all exported entities. However, wrappers need to be constructed only for the exported entities that are defined in the module. For imported and reexported entities the wrapper module needs to import the wrappers and reexport them.

6 Calling Traced Code from Untraced Code

We are assuming that traced code may call untraced code, but not vice versa. The intuition of "trusting" suggests this restriction as natural. However, "trusting" makes only sense in the context of debugging. We may well want to trace a module of a large program without tracing all other modules which call the traced module (e.g. to trace a single compiler phase) . The resulting trace will in general consists of several, probably unconnected parts. Also, just to test a module we may need other “driver” modules that produce test data and compare results. We are not interested in tracing these “driver” modules.

Traced functions may already be passed to untraced higher-order functions which call the traced functions. So dealing with static calls of traced functions from untraced functions should not be hard.

Note also that derived class instances should not be traced (automatically generated code is assumed to be correct), but the methods may call traced code (hand written instances).