Typed Transformations of Typed Abstract Syntax

Arthur I. Baars
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Valencia, Spain
abaars@iti.upv.es

S. Doaitse Swierstra
Department of Computer Science
Utrecht University
Utrecht, The Netherlands
doaitse@cs.uu.nl

Marcos Viera
Instituto de Computación
Universidad de la República
Montevideo, Uruguay
mviera@fing.edu.uy

Abstract

Advantages of embedded domain-specific languages (EDSLs) are that one does not have to implement a separate type system nor an abstraction mechanism, since these are directly borrowed from the host language. Straightforward implementations of embedded domain-specific languages map the semantics of the embedded language onto a function in the host language. The semantic mappings are usually compositional, i.e. they directly follow the syntax of the embedded language.

One of the questions which arises is whether conventional compilation techniques, such as global analysis and resulting transformations, can be applied in the context of EDSLs. The approach we take is that, instead of mapping the embedded language directly onto a function, we first build a representation of the abstract syntax tree of the embedded program fragment. This syntax tree is subsequently analyzed and transformed, and finally mapped onto a function representing its denotational semantics. In this way we achieve run-time "compilation" of the embedded language.

Run-time transformations on the embedded language can have a huge effect on performance. In previous work (Viera et al. 2008) we present a case study comparing the *Read* instances generated by Haskells **deriving** construct with instances on which run-time grammar transformations (precedence resolution, left-factorisation and left-corner transformation) have been applied.

In this paper we present the library, which has an arrow like interface, which supports in the construction of analyses and transformations, and we demonstrate its use in implementing a common sub-expression elemination transformation. The library uses *typed abstract syntax* to represent fragments of embedded programs containing variables and binding structures, while preserving the idea that the type system of the host language is used to emulate the type system of the embedded language. The tricky issue is how to keep a collection of mutually recursive structures well-typed while it is being transformed.

We finally discuss the typing rules of Haskell, its extensions and those as implemented by the GHC and show that pure System-F based systems are sufficiently rich to express what we want to express, albeit at the cost of an increased complexity of the code.

Categories and Subject Descriptors F.3.3 [*Logics and meanings of programs*]: Studies of program Constructs; D.3.3 [*Programming languages*]: Language Constructs and Features; D.1.1 [*Programming techniques*]: Applicative (Functional) Programming; D.3.4 [*Programming languages*]: Processors

General Terms Algorithms, Languages, Verification

Keywords GADT, Meta Programming, Type Systems, Typed Transformations, Common Subexpression Elimination

1. Introduction

Modern functional languages such as Haskell and ML are excellent tools for embedding domain specific languages. This is usually done by defining a library of combinators, each combinator representing a grammatical structure from the embedded language. With such a library a programmer can use a domain-specific notation, and at the same time benefit from all the features, such as the type system and abstraction mechanisms, of the general purpose host language. The tight integration of the domain-specific language with the host language has great benefits: there is no need to extend the compiler or to use ad hoc external tools which map the specific notation onto the host language.

One of the essential aspects of a combinator-based domain-specific embedded language is that it shares its type system with the host language.

Although such a combinator-based approach for implementing domain-specific languages looks very convincing at first sight, many problems show up when one starts to use this technique in practice. Just as conventional compilers may analyze programs extensively and may transform them based on the results of such analyses, one also wants to use such techniques for combinator-based embedded languages.

Unfortunately, such analyses and transformations are usually not possible, because the value being constructed by the combinators is a (possibly higher order) function (as in denotational semantics), and hence the internal structure can be neither inspected nor transformed. The approach we take is to have the combinators build a data structure which corresponds to the *typed abstract syntax tree* as described by the grammar of the embedded language. This representation can then be analyzed, transformed, and finally mapped onto its semantics. The main complication we now face is that we have to do this *in a typed setting*.

The problem becomes even harder when the embedded language not only borrows the type system and the abstractions provided by the host language, but also its *declarative* structure. Now we have to deal with collections of abstract syntax trees containing references to each other and the question arises how to representing and observe the binding structures.

In their work on typed meta-programming (Pasalic and Linger 2004), Pasilic and Linger, using the encoding of equality types in Haskell (Baars and Swierstra 2002), show how to represent typed abstract syntax trees containing typed references to values. A group of mutually recursive bindings is represented by a nested cartesian product of terms. The variables occurring in the terms are represented by typed pointers into this enviroment. A similar approach was developed slightly earlier by Xi and Chen, using a dependently typed version of ML, in their work on typeful program transformations (Chen and Xi 2003). The key ingredient for both is the encoding of environments and references pointing into these environments in host language terms. This encoding ensures that references always refer to *existing values* with the right types. In all approaches, references are basically indices into an environment, encoded as Peano numbers.

In earlier work we demonstrated the practical use of run-time typed grammar transformations for combinator parsers. In (Baars and Swierstra 2008) we have described a quite involved grammar transformation, the *Left_Corner transform* which effectively removes left-recursive non-terminals from a grammar and replaces them by a set of non left-recursive ones. As a consequence top-down parsers can directly be generated from the resulting grammar. In (Viera et al. 2008) we eliminate common prefixes from sets of productions in order to improve parsing efficiency. In a small case study we derive an efficient version for the Haskell function *read*, converting the worst case exponential time algorithm used thus far into a linear one.

In this paper we explain the inner-workings of the library underlying these transformations. It has an *Arrow*-style interface and handles introductions of new typed bindings. It is not geared towards grammar transformations and thus can be used to implement a wide variety of other typed program transformations. As a small example of the use of the library we include an implementation of common-subexpression elimination, assuming that the algorithm used thus does not become a object of study by itself and attention can focus on the way the library is used.

Internally, the library makes heavy use of universally and existentially quantified types in combination with Generalized Algebraic Data Types (GADT). This paper therefore can also be seen as a non-trivial exercise using these type system extensions. Throughout the paper, we use GHC (GHC) syntax for these extensions to Haskell. The code in this paper can be found at: `http://www.cs.uu.nl/wiki/Center/TTTAS`, and was produced by running `lhs2TeX` on the source of this paper. The code is accepted by the Glasgow Haskell Compiler (GHC). The library described in this paper is also released as a `HackageDB`(Hackage) package and can be downloaded from: `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/TTTAS`.

This paper is organized as follows. In Section 2, summarizing earlier work (Pasalic and Linger 2004; Baars and Swierstra 2002, 2004), we discuss the encoding of typed abstract syntax trees, references and environments. These are the objects for our transformations. In Section 3, we develop the library that maintains a changing typed environment. It ensures that all typed references (references that "know" the type of the values they refer to) remain consistent whenever a new definition is added to the environment. In Section 4, we show an example of the use of the library: the implementation of common sub-expression elimination. The implementation of the library makes use of lazy-pattern matching on data constructors involving existential types. This combination is unfortunately not supported by GHC. In Section 5 we present several solutions to avoid this problem. Finally, in Section 6, we present our conclusions.

2. Typed References and Environments

We start by shortly repeating the ideas behind typed meta-programming, in which we represent programs explicitly, so they can be manipulated. We want to do this in such a way that we keep the nice properties that typed programming languages have. Specifically, the fact that the representation is type correct can be seen as a proof that the represented expression is type correct.

As an example consider the abstract syntax (formulated as a GADT) of a simple expression language:

```
data Exp a where
  IntVal   :: Int                                → Exp Int
  BoolVal  :: Bool                               → Exp Bool
  Add      :: Exp Int  → Exp Int                 → Exp Int
  Cons1    :: Exp a    → Exp [a]                 → Exp [a]
  Nil1     ::                                      Exp [a]
  LessThan :: Exp Int  → Exp Int                 → Exp Bool
  If       :: Exp Bool → Exp a → Exp a           → Exp a
```

where the expression

```
if 3 + 1 < 4 then 5 else 1 + 2
```

is represented by:

```
expr :: Exp Int
expr = If (LessThan (Add (IntVal 3) (IntVal 1))
                    (IntVal 4))
          (IntVal 5)
          (Add (IntVal 1) (IntVal 2))
```

The value of the represented expression has type *Int*, which is reflected in the type of $expr :: Exp\ Int$. Note that the ill-typed expression $3 < True$ cannot be represented, since it will not pass the type-checker.

The type $Exp\ a$ encodes the typing judgement $\vdash e : \alpha$, that reads "the expression e has type α". Each constructor has the structure of a formal judgment. For example *LessThan* encodes the rule:

$$\frac{\vdash e1 : Int \qquad \vdash e2 : Int}{\vdash e1 < e2 : Bool}$$

Before discussing analyses and transformations we have to decide how to represent variables and binding structures. We extend our simple expression language with a constructor *Var*, where a variable is represented by a reference of type $Ref\ a\ env$, an index pointing to a value of type a in an environment of type env. In the next section, we will delve into the details of the type *Ref*.

We extend the labelling (properties) of the type *Expr* by an extra type parameter *env*, which stands for the type of the environment in which the expression is to be evaluated:

```
data Expr a env where
  Var      :: Ref a env       → Expr a env
  IntVal   :: Int             → Expr Int env
  BoolVal  :: Bool            → Expr Bool env
  Cons     :: Expr a env      → Expr [a] env
           → Expr [a] env
  Nil      :: Expr [a] env
  Add      :: Expr Int env    → Expr Int env
           → Expr Int env
  LessThan :: Expr Int env    → Expr Int env
           → Expr Bool env
  If       :: Expr Bool env → Expr a env
           → Expr a env     → Expr a env
```

Now we have the judgement $\Gamma \vdash e : \alpha$, that reads "the expression e has type α in local context Γ".

An evaluator *eval* for our simple expression language takes as arguments the abstract syntax tree decribing an expression, and an environment which provides values for the variables occurring in the expression, and returns the value of the expression. For the time being only the type of the function *lookup* matters:

```
lookup :: Ref a env → env → a

eval :: Expr a env        → env → a
eval (Var r)            e = lookup r e
eval (IntVal i)         _ = i
eval (BoolVal b)        _ = b
eval (Add x y)          e = eval x e + eval y e
eval (Cons x y)         e = eval x e : eval y e
eval Nil                _ = []
eval (LessThan x y)     e = eval x e < eval y e
eval (If x y z)         e = if   eval x e
                            then eval y e
                            else eval z e
```

2.1 Type equality

Pasalic and Linger (Pasalic and Linger 2004) introduce an encoding of typed references that can be used for meta-programming. This encoding relies on the equality type (Baars and Swierstra 2002; Weirich 2000; Cheney and Hinze 2003). A (non-diverging) value of type *Equal a b* is a witness of the proof that the types a and b are equal. This witness takes the form of a conversion function, which turns out to always be the identity function.

The addition of GADTs (Peyton Jones et al. 2006) to GHC makes programming with the *Equal* data type a lot easier, because all the fiddling with proofs is implicitly done by the compiler. Furthermore, the performance increases, since the construction of the proofs is no longer done at run-time. The compiler "knows" that all proofs of type equality are witnessed by values like *id*, *id id*, *id* (*id id*), *id id id* etc, and can thus omit them safely from the generated code: they have no other observable effect than taking time to execute.

The encoding of type equality becomes trivial when using GADTs:

```
data Equal :: * → * → * where
  Eq :: Equal a a
```

The type *Equal* has just one constructor *Eq* :: *Equal a a*. If a pattern match on a value of type *Equal a b* succeeds (i.e., a non-$\bot$ value *Eq* is available), then the type checker is thus informed that the types a and b were known to be the same at the place the *Eq* was produced.

2.2 Typed References

In their paper on typed meta-programming, Pasalic and Linger introduced the data type *Ref* for representing *typed* indices which are labelled with both the type of value to which they refer and the type of environment (a nested cartesian product, growing to the right) in which this value lives.

```
data Ref a env where
  Zero ::              Ref a (env′, a)
  Suc  :: Ref a env′ → Ref a (env′, b)
```

In the case of a *Suc* we are not interested in the first element; this constructor is polymorphic in the type b. The rules encoded by the type *Ref* are:

$$\frac{}{\Gamma', \alpha \vdash \mathit{Zero} : \alpha} \qquad \frac{\Gamma' \vdash r : \alpha}{\Gamma', \beta \vdash \mathit{Suc}\ r : \alpha}$$

Two references can be compared for equality using the function *match*. If they refer to the same element in the environment this function returns the value *Just Eq*, thus expressing the fact that the types of the referred values are the same too:

```
match :: Ref a env → Ref b env → Maybe (Equal a b)
match Zero    Zero    = Just Eq
match (Suc x) (Suc y) = match x y
match _       _       = Nothing
```

The *lookup* function, the type of which we have seen before, uses its *Ref* parameter as an index in the environment parameter. Whenever we decrease the index, we take the *fst* part of the tuple, until the index reaches *Zero*. The types guarantee that the lookup succeeds:

```
lookup :: Ref a env → env → a
lookup Zero    (_, a) = a
lookup (Suc r) (e, _) = lookup r e
```

The function *update* takes an additional function as argument, which is used to update the value the reference addresses. The other values in the environment are left unchanged:

```
update :: (a → a) → Ref a env → env → env
update f Zero    (e, a) = (e, f a)
update f (Suc r) (e, x) = (update f r e, x)
```

As an example, consider the *example* environment:

```
type ExampleEnv = ((((), Int), Char), String)
example = ((((), 1), 'a'), "b") ::               ExampleEnv
ref_a   =      Suc Zero         :: Ref Char ExampleEnv
ref_one = Suc (Suc Zero)        :: Ref Int  ExampleEnv
```

The expression *lookup* ref_a *example* yields the character `'a'`, while the expression *lookup* ref_{one} *example* yields the integer 1. Notice that the type of the reference determines the type of the result! Application of *update* (+5) ref_{one} to the example environment updates it to ((((), 6), `'a'`), `"b"`). This clearly shows that the ref_a and ref_{one} refer to values of different types in the same environment.

With our extended data type we now also can encode expressions which contain variables of different types, such as **if** b **then** 3 **else** a, using an environment containing an *Int* and a *Bool*:

```
var_a = Var (Suc Zero)
var_b = Var      Zero
e     = If var_b (IntVal 3) var_a
                          :: Expr Int (((), Int), Bool)
env   = (((), 11), False) ::          (((), Int), Bool)
test  = eval e env        ::     Int
```

Some may complain that this Peano representation is extremely cumbersome and error prone. In (Baars and Swierstra 2004), we have shown how, by using some extra combinators, this problem can be overcome. Furthermore the type system also helps us to avoid accidental mistakes. Also, note that building and maintaining the internal representation is the work of the combinator library and is largely invisible to the programmer describing program transformations.

2.3 Declarations

In this section we will focus on the problem how to represent a collection of possibly mutually recursive definitions, each consisting of an identifier being defined and a right-hand side expression containing these identifiers.

The idea is to store the right-hand side expressions in a heterogeneous list, and represent the identifiers by indices in this list. This is very similar to the environments described above, with the main difference that the actual environment now contains abstract syntax terms labelled with a type instead of values having a type.

In this setting an environment consists of terms which in turn are labelled with the type of the environment, because this type is used to label the references contained in these terms. Representing such environments as a nested tuples would lead to an infinite type. For example consider the following two declarations:

```
let x = 1 : y
    y = 2 : x
```

These declarations give rise to an environment containing two terms. Suppose we name the type of this environment $TwoLists$, then both terms have type $Expr\ [Int]\ TwoLists$. This leads to the following type for the environment:

```
type TwoLists = (((), Expr [Int] TwoLists)
                    , Expr [Int] TwoLists)
```

This type definition is cyclic, and is thus not allowed in Haskell.

Our solution is found in splitting the env type parameter into two parameters: one for the environment addressed by the references occurring in the terms and one dscribing the environment which is being constructed by the sequence of terms. The type $Env\ term\ use\ def$ represents a sequence of instantiations of type $\forall a\ .\ term\ a\ use$, where all the instances of a are stored in the type parameter def; thus the type def contains the type parameters a of the terms of type $term\ a\ use$ occurring in the $Env\ term\ use\ def$. The type use on the other hand is a sequence containing the types to which may be referred from within terms of type $term\ a\ use$.

```
data Env term use def where
  Empty :: Env t use ()
  Ext   :: Env t use def' → t a use
        → Env t use (def', a)
```

When the types def and use coincide we can be sure that the references in the terms do not point to values outside the environment and do point to terms representing the right type. hence we can use the nvironment being defined as the environment to be indexed by the refernces contained in the right-hand side terms of the definitions.

Splitting this single type into two type parameters, which we only require in the end to be equal, makes it possible to to use references which refer to terms which still have to be added. Only after we are done with manipulating and extending the environment we require use and def to be the same! The fact that a sequence of terms is closed and well-typed is thus encoded in the type system of the host language. So the mutually recursive declarations:

```
let x = 1 : y
    y = 2 : x
```

is encoded as:

```
type Final = (((), [Int]), [Int])
x     = Var (Suc Zero)   ::  Expr [Int] Final
y     = Var Zero         ::  Expr [Int] Final
decls :: Env Expr Final Final
decls = Empty `Ext` Cons (IntVal 1) y
              `Ext` Cons (IntVal 2) x
```

where we note that the x and y here are Haskell values referring to the right-hand side terms of their definitions in the Env.

The lookup and update operations on the data type Env are defined in a similar way as before:

```
lookupEnv :: Ref a env → Env t s env → t a s
lookupEnv Zero    (Ext _ t)  = t
lookupEnv (Suc r) (Ext ts _) = lookupEnv r ts
```

```
updateEnv :: (t a s → t a s) → Ref a env
          → Env t s env → Env t s env
updateEnv f Zero    (Ext ts t)
          =         Ext ts (f t)
updateEnv f (Suc r) (Ext ts t)
          =         Ext (updateEnv f r ts) t
```

The chosen representation now has an efficiency problem, to be fixed in the next section: whenever we extend the environment with a new Ext all existing references occurring in terms already stored in the environment have to be incremented by applying an extra Suc constructor to them, since the values to which they refer have an index that is one higher in the new environment.

3. Transformation Library

In this section we develop a type $Trafo$ representing typed transformation steps on a heterogeneous collection and an $Arrow$-like (Hughes 2000) library of combinators for composing such transformations. Each $Trafo$ takes input and produces output and can be composed in the same way as $Arrow$s. Internally it maintains an environment containing abstract syntax terms. Additionally, metadata about the transformation process can be maintained as well. Such meta-data could for example be a symbol table, debugging information, reference counts, etc.

In developing the type $Trafo$ we use a Haskell-like type synonym syntax augmented with the symbols ∀ and ∃ to denote universally and existentially quantified types. We first develop the type $Trafo$ to tackle the problem of maintaining a heterogeneous collection of definitions, and subsequently extend it with $Arrow$-style inputs, outputs, and meta-data. Finally we encode the $Trafo$ using data types, as accepted by GHC.

We model a collection of embedded-language definitions as a value of type Env, and make these definitions the subject of transformations that may induce new definitions, as in the case of common-subexpression removal where subexpressions get named. At the end of the transformation process each reference occurring somewhere in a term stored in this Env must be a reference into the final set of definitions. In this case, we call the environment *closed*. One way to ensure that our environment is always closed is to adjust all the references in all the terms whenever a new definition is added to the environment. Unfortunately new definitions are to be added at the head of the sequence (i.e. at position $Zero$), which implies that all existing references have to be updated by the application of an extra Suc constructor. This is cumbersome and inefficient, and is better done once, i.e., when we know how many Sucs to add to each reference so it addresses the right element in the final structure.

We only require an environment to be closed after all transformations have been applied and all new definitions have been added. The final type must be of the form $Env\ t\ s\ s$ (or $FinalEnv\ t\ s$) for some type s.

```
type FinalEnv t usedef = Env t usedef usedef
```

References into this environment s are coined *final references*. If all the transformation steps only add terms of type $(\exists a\ .\ t\ a\ s)$ to the environment, then they contain only final references, and we do not need to adjust the references after each transformation step. However, this seems to be impossible. How can we make the transformation steps half way through the transformation process construct terms of the type $(\exists a\ .\ t\ a\ s)$? After all s is the type of the final environment and is only known after all transformation steps have completed. For creating such final references we need to know how many new definitions will be added to the environment by future transformation steps.

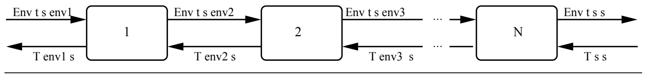

Figure 1.

We solve this problem by using Haskell's lazy evaluation to pass knowledge about the "future" backwards through the computation. In our case information about the number of definitions added by future steps is encoded in a *Ref*-transformer, that prepends as many *Suc*-nodes to a reference as there are new definitions to come. The type of such *Ref*-transformers is[1]:

```
newtype T e s = T{unT :: ∀x . Ref x e → Ref x s}
```

Figure 1 depicts the idea described above. The environment is constructed from left to right. Each step takes as input the environment constructed thus far and yields an updated environment as result. On the top left, the computation starts with an environment of some type *Env t s env1*. Each step extends the environment with some new definitions, making the type of the environment change at every step. The final result of all steps has to be an environment of type *FinalEnv t s*. *Ref*-transformers are passed on and modified from right to left. These transformer effectively inform every intermediate step how deep down the environment constructed thus far is located in the final environment.

The environment yielded by the last step is the place where the *use* and the *def* types have to coincide. Therefore the identity transformer is used as the initial value for the "pass-back" chain. Every step updates the transformer according to the number of definitions it adds to the environment, before passing it on to its preceding transformation step.

3.1 The *Trafo* data type

We now develop the type *Trafo* to implement the idea described above. Every step has two incoming and two outgoing arrows, one of each type at each side. This means our *Trafo*-type is a function taking two arguments and returning two results. We want our *Trafo*-type to be polymorphic in the type of the terms (t) stored in the environment. As a first attempt, we take:

```
type Trafo t =
  T env2 s → Env t s env1
   → (T env1 s, Env t s env2)
```

the role of the different elements is as follows:

Env t s env1 is the environment which has been constructed up to where the current transformation starts, and corresponds to the incoming arrow at the top left. The *env1* parameter describes which elements have thus far been added to the environment.

T env2 s is the incoming arrow at the bottom right. It maps references into an enviroment labelled with *env2* into references into the final environment s.

Env t s env2 is the environment constructed by this step, and *env2* will usually be either *env1* or an extension of *env1*.

T env1 s corresponding to the bottom left arrow coming out of a *Trafo*, is the updated *T env2 s*, which can be constructed by the transformation since it knows how many elements it adds to the environment.

This type definition is incomplete, the type variables *env1*, *env2*, and s are still unbound. We do not want all these variables to appear on the left-hand side of the type definition, as this would expose the internal complexity of the library to the user, so we have to add universal or existential quantifiers.

The type *env1* is the type of the environment constructed thus-far. A step should not make any assumptions about this environment, and hence the type variable *env1* is to be universally quantified. The type *env2* is the type of the result of a transformation step. This type depends on the number of new definitions introduced by the step. As this can be an arbitrary number, the type *env2* is fully determined by the transformation and the incoming *env1*. Because *env2* depends on *env1* by extending it, this quantifier has to be within the scope of *env1*: once *env1* is fixed the transformation fixes *env2*. Finally, the type variable s represents the type of the final result, which we do expose. Its role is similar to the s in the type *ST s a*, which is the type of state threads (Launchbury and Jones 1994) of the Haskell libraries. All this leads to the following definition for the type *Trafo*:

```
type Trafo t s =
  ∀env1 . ∃env2 .       T env2 s → Env t s env1
                      → (T env1 s,    Env t s env2)
```

In the next step, we extend the type *Trafo* with *Arrow*-style input and output. This allows us to pass values from one transformation *Trafo* to the next, in addition to the implicitly passed environment and ref-transformers. So we add two arguments (a and b) to the type *Trafo*, which stand for the types of the input and output:

```
type Trafo t s a b =
  ∀env1 . ∃env2 .       a → T env2 s → Env t s env1
                      → (b,     T env1 s,    Env t s env2)
```

Finally we may want to maintain meta-information about the environment, such as which elements have been added already. This information may be used to determine wheather new elements have to be added to the environment; hence it has to live outside the type which is existentially quantified by *env2*, but it will in general depend on the environment *env1* constructed thus far.

Thus, we introduce an extra argument m that stands for the type of the meta-data. A *Trafo* takes the meta-data on the current environment *env1* as input and yields meta-data for the (possibly extended) environment *env2*.

```
type Trafo m t s a b =
  ∀env1 . m env1
          → ∃env2 .
              ( m env2
              ,        a → T env2 s → Env t s env1
                 → (b,     T env1 s,    Env t s env2)
              )
```

We now have come to a problematic point: the type above is not Haskell, nor is it accepted by the GHC due to the use of existential quantifiers in type definitions. Unfortunately an existential type can only be introduced by using the keyword ∀ on the left side of a constructor in a data-declaration. Thus, we have to resort to an encoding of the above type using two other data types:

```
data Trafo m t s a b =
  Trafo (∀env1 . m env1 → TrafoE m t s a b env1)
data TrafoE m t s a b env1 =
  ∀env2 . TrafoE
              (    m env2)
              (    a → T env2 s → Env t s env1
              → ( b,     T env1 s,    Env t s env2)
              )
```

[1] Note that the keyword **forall** is presented by the logical symbol ∀

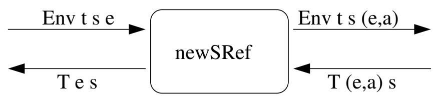

Figure 2.

Now that we have developed the final version of our *Trafo* data type we can define the combinators to construct and compose transformation steps.

3.2 Creating new references

The most important operation is the extension of the enviroment with a new term, returning a reference to this newly added term. This operation is implemented by *newSRef*, which takes a typed term as input, adds it to the environment and yields a reference pointing to this value. The type of *newSRef* is:

```
newSRef :: Trafo Unit t s (t a s) (Ref a s)
data Unit s = Unit
```

No meta-information on the environment is recorded by *newSRef*; therefore we use the type *Unit* for the meta-data. If meta-information is required, one must define an application-specific version of *newSRef*, as we will do in our example application in Section 4. The type variable t stands for the type of the terms. We want the input to *newSRef* to be of type $t\ a\ s$, where s stands for the type of the final environment. The result of a *newSRef* is a reference of type *Ref* $a\ s$, which points to the newly inserted, a labelled, term in the final environment.

```
newSRef
   = Trafo (λ_ → TrafoE Unit extEnv)
extEnv ::    t a s → T (e, a) s → Env t s e
          → (Ref a s, T e       s,    Env t s (e, a))
extEnv ta (T tr) env
          = (tr Zero, T (tr . Suc), Ext env ta)
```

The incoming meta-information is ignored and *Unit* is returned as meta-data. The function *extEnv*, used in *TrafoE*, is more interesting: it takes as arguments the term to be inserted $ta :: t\ a\ s$, the current environment $env :: Env\ t\ s\ e$, and a reference transformer $(T\ tr) :: T\ e\ s$, which transforms references into the current environment into references into the final environment. The result is a tuple containing the new environment, which has type $Env\ t\ s\ (a, e)$, and a reference of type $Ref\ a\ s$. The term (ta) becomes the last element of the new environment, hence the reference pointing to this term is *Zero*. However, more terms may be added in the future, therefore the reference transformer (tr) is applied, which basically prepends to *Zero* as many *Suc*-nodes as there are future additions to the environment under construction. Finally, we record the fact that one new element was added to the environment by adding an extra *Suc*-node to the reference transformer tr, which we pass on to our predecessors. Since all application-specific versions of *newSRef* have to do this work, we include the function *extEnv* as part of the library.

In certain cases an application-specific *newSRef* will not have to add a new term, but will return an existing reference instead. For these cases we added a function that casts a reference in the constructed environment to one in the final environment:

```
castSRef ::    Ref a env
          →     x         → T env s → Env t s env
          →   (Ref a s  ,  T env s  ,  Env t s env)
castSRef r = (λ_ (T t) decls → (t r, T t, decls))
```

3.3 *runTrafo*

Of course we want to "run" our *Trafo*-computations. This is the task of the function *runTrafo*, which has the following type:

```
runTrafo :: (∀s . Trafo m t s a (b s))
            → m () → a → Result m t b

data Result m t b
   = ∀env2 . Result (m env2) (b env2)
                    (FinalEnv t env2)
```

The type of *runTrafo* is inspired by that of $runST :: (\forall\ s\ .\ ST\ s\ a) \rightarrow a$, which is part of the state thread library (ST). The rank-2 type for *runTrafo* ensures that transformation steps cannot make any assumptions about the type of final environment (s).

The function *runTrafo* takes as arguments the *Trafo* we want to run, meta-information for the empty environment, and an input value. The result of *runTrafo* is the final environment ($Env\ t\ env2\ env2$) together with the resulting meta-data ($m\ env2$), and the output value ($b\ env2$). Because $env2$ could be anything we have to hide it using existential quantification, and thus introduce the data definition *Result*.

Note that the type of the output is $(b\ s)$, one might wonder why the output is not just some type c(not labelled with s). The reason is that returning a value of type $b\ s$ is slightly more general. It allows a transformation to return a value labelled with type s, which would otherwise not be allowed.

The implementation of the function *runTrafo* reads:

```
runTrafo :: ∀m t a b . (∀s . Trafo m t s a (b s))
                      → m () → a → Result m t b
runTrafo trafo m a =
  let Trafo trf      = trafo
      TrafoE m2 f = trf m
  in case f a (T id) Empty of
        (b, _, env) → Result m2 b env
```

The function f inside the *Trafo* type is applied to the input value (a), the identity *Ref*-transformer, and the empty environment. The result of f is the output (b), and the transformation result (env), which are wrapped in a *Result* constructor together with the resulting meta-data ($m2$). The function *runTrafo* uses lazy pattern binding for the matches on *Trafo* and *TrafoE* (for strict pattern matching one should use **case** instead of **let**). This is essential as we need to instantiate the universally quantified s with the existential type constructed by the inner *TrafoE* constructor. Unfortunately this is not allowed by the Glasgow Haskell compiler (GHC) as it forbids the use of lazy pattern matching in combination with existential types. Other compilers such as Hugs(Hugs), and the Haskell compiler under construction at Utrecht University(EHC) do allow this combination. In Section 5 we suggest two solutions to circumvent this problem.

3.4 Arrow-style combinators

The *Arrow* library consists of a set of functions for constructing and combining values that are instance of the *Arrow* class. Furthermore there is a convenient notation for programming with *Arrow*s. This notation is inspired by the **do**-notation for *Monad*s. To implement the *Arrow* interface one needs to implement three methods *arr*, >>>, and *first*.

We make the type (*Trafo* $m\ t\ s$) instance of the *Arrow* class:

```
instance Arrow (Trafo m t s) where
```

The method *arr* lifts a function.

```
  -- arr :: (a → b) → Trafo m t s a b
  arr f = Trafo (λm → TrafoE m (λa t e → (f a, t, e)))
```

The >>> operator composes two *Trafo*s. It is actually a straightforward transcription of the composition depicted in Figure 1. In that figure box 1 refers to the incoming environment, box 2 to the intermediate and box 3 to the outgoing.

```
  -- (>>>) :: Trafo m t s a b → Trafo m t s b c
  -- → Trafo m t s a c
Trafo t1 >>> Trafo t2 =
  Trafo
  (λm1 → case t1 m1 of
    TrafoE m2 f1 → case t2 m2 of
      TrafoE m3 f2 →
        TrafoE
        m3
        (λa tt env1 →
            let (b, tt1, env2) = f1 a tt2 env1
                (c, tt2, env3) = f2 b tt env2
            in  (c, tt1, env3)
        )
  )
```

The method *first* applies the first component of the input to the argument *Trafo* and copies the rest unchanged to the output. It is implemented as follows:

```
  -- first :: Trafo m t s a b → Trafo m t s (a, c) (b, c)
first (Trafo tr)
  = Trafo (λm1 → case tr m1 of
    TrafoE m2 f →
      TrafoE
      m2
      (λ~(a, c) tt env1 →
        let (b, tt1, env2) = f a tt env1
        in ((b, c), tt1, env2)))
```

For easy reference, we also show the other functions of the *Arrow*-interface. The code is just the default definition found in the *Arrow*-class.

```
second :: Trafo m t s b c → Trafo m t s (d, b) (d, c)
second f = arr swap >>> first f >>> arr swap
  where   swap~(x, y) = (y, x)
(***) ::   Trafo m t s b c → Trafo m t s b' c'
      →   Trafo m t s (b, b') (c, c')
f *** g = first f >>> second g
(&&&) ::   Trafo m t s b c → Trafo m t s b c'
      →   Trafo m t s b (c, c')
f &&& g = arr (λb → (b, b)) >>> (f *** g)
```

The function *loop* is used to construct feedback loops. It takes a *Trafo* that has an input of type (a, x) and output of type (b, x). The component of type x is fed back resulting in a *Trafo* with input a and output b.

```
instance ArrowLoop (Trafo m t s) where
    -- loop :: Trafo m t s (a, x) (b, x) → Trafo m t s a b
  loop (Trafo st) =
    Trafo
    (λm → case st m of
      TrafoE m1 f1 →
        TrafoE m1
        (λa t e →
          let ((b, x), t1, e1) = f1 ((a, x)) t e
          in (b, t1, e1)
        ))
```

4. Common sub-expression elimination

In this section we show how the library developed in the previous section can be applied to implement common sub-expression elimination (CSE). The object of this transformation is the *Expr*-language from Section 2.

CSE is a compiler optimization, where for each sub-expression e that occurs more than once the CSE transformation introduces a new declaration $v = e$, and furthermore replaces all subsequent occurrences of e with the variable v.

For example the following expressions:

$a = 4;$
$b = (a + 4) + (a + 4);$

are transformed into:

$a = 4;$
$x = a + a;$
$b = x + x;$

The subject of our *CSE* transformation is a sequence of (possibly mutually recursive) declarations. These are reprensented as an *Env* of typed *Expr*s:

```
type Decls env = Env Expr env env
```

In the type *Decls* above the type variable *env* encodes the type of each of the declarations. The result of the transformation is also a sequence of declarations. It is labelled with a different type variable because the *CSE* transformation may introduce new declarations. The amount of newly introduced declarations depends on the number of common sub-expressions in the original set of declarations. As a result the type of the result of the transformation is not statically known. Therefore we introduce the following existential type for the result of the *CSE* transformation:

```
data TDecls env = ∀env' . TDecls (Decls env')
                                  (T env env')
```

In the type *TDecls env*, the type variable *env* stands for the type of the original declarations. The type *TDecls* constains a sequence of declarations (*Decls env'*), in which the type variable *env'* represents the type of the transformed declarations. The transformed declarations are accompanied by a *Ref*-transformer mapping references from the original sequence of declarations to references in the new one.

Summarizing the type implementation of the *CSE* transform, developed in the remainder of this section has the following type:

```
cse :: Decls env → TDecls env
```

Before we delve into the implementation of *cse*, we first show an example.

$a = 4;$
$b = (a + 4) + (a + 4);$

These declarations are encoded as typed abstract syntax as follows:

```
a = Suc Zero
b = Zero
exampledecls :: Decls (((), Int), Int)
exampledecls =
  Empty `Ext` (IntVal 4)
        `Ext` (Add (Add (Var a) (IntVal 4))
                   (Add (Var a) (IntVal 4)))
```

To transform the declarations we apply the *cse* function:

```
resdecls :: TDecls (((), Int), Int)
resdecls = cse exampledecls
```

The transformed declarations (*resdecls*) can be used as follows:

```
evalDecls :: Decls env → env
evalVar   :: Ref a env → TDecls env → a
evalVar var (TDecls ds (T tt))
   = lookup (tt var) (evalDecls ds)

value_a = evalVar a resdecls
value_b = evalVar b resdecls
```

The function *evalVar* takes a reference and the transformed declarations as arguments. It evaluates the declarations(*evalDecls*) and uses the reference in combination with the *Ref*-transformer(*tt*) to select the value from the evaluated declaration that corresponds the reference(*var*). Note that we omitted the definition of *evalDecls* and only show its type.

The transformed declarations (*resdecls*) internally have the following structure:

```
TDecls
   (   Empty ‘Ext‘ (IntVal 4)
             ‘Ext‘ (Add (Var (Suc (Suc Zero)))
                        (Var (Suc (Suc Zero))))
             ‘Ext‘ (Add (Var (Suc Zero))
                        (Var (Suc Zero)))
   )
   (   T (λref → case ref of
          Zero     → Zero
          Suc Zero → Suc (Suc Zero))
       :: T (((), Int), Int) ((((), Int), Int), Int))
```

A new declaration has been inserted in between those for a and b, this fact is reflected in the *Ref*-transformer. The reference *Zero* (for b) remains unchanged because the declaration b is still the last one. The reference for declaration a however gets an extra *Suc* node.

4.1 Implementation

Briefly our implementation of *CSE* performs the following steps: For each sub-expression

- check if we already encountered it
  - if not, add a declaration for this sub-expression to the result
  - if yes, replace it by a reference to the equivalent expression that is already in the result

To determine whether expressions have common sub-expressions we need to compare expressions for equality. Therefore we introduce the function *equals*, which compares two expressions, and, if they are equal returns a witness encoding that the types of the two expressions are the same.

```
equals :: Expr a env → Expr b env
       → Maybe (Equal a b)
equals (Var r1) (Var r2) = match r1 r2
equals (IntVal i1) (IntVal i2)
   | i1 ≡ i2              = Just Eq
equals (LessThan x1 y1) (LessThan x2 y2)
   = do Eq ← equals x1 x2
        Eq ← equals y1 y2
        return Eq

...
equals _ _ = Nothing
```

The implementation of the function *equals* is fairly straightforward. To determine whether two *Var*s are equal the function *match* is applied to determine whether the contained references are the same. Two *IntVal* expressions are equal if their contained values are the same. To determine whether two *LessThan* expressions are equivalent, the function *equals* is recursively applied on their components. We omit the definitions of *equals* for the constructors *BoolVal*, *Add* and *If*, because they are very similar to the ones above.

During the *CSE* transformation we need to determine whether we already encountered an expression before. If an expression has not been encountered before, a declaration for it is added to the result. On the other hand if it was encountered before, it is not added to the result, but is instead replaced by a reference to the equivalent expression that is already present in the result.

For this we introduce the type *Memo*:

```
newtype Memo env env′
  = Memo
    (∀x . Expr x env
        → Maybe (Ref x env′)
    )
```

The *Memo* tells us whether an expression has been encountered before, and if so, returns a witness in the form of a reference to the copy of the expression in the transformation result. Note the use of two distinct type variables: *env* stands for the type of the original sequence of declarations and *env′* for the result of the transformation.

We introduce a "smart" constructor to create an empty *Memo*:

```
emptyMemo :: Memo env ()
emptyMemo = Memo (const Nothing)
```

We proceed by introducing a type synonym for the *CSE* transformation *Arrow*:

```
type TrafoCSE env = Trafo (Memo env) Expr
```

The terms that are to be transformed have type *Expr* and the state (meta-data) maintained is a table of type *Memo*.

During the transformation all sub-expressions are visited. For each sub-expression we check whether it has already been encountered before. If so the table of type *Memo* provides us a reference to the earlier occurrence of the sub-expression, which is used as a replacement for the current sub-expression. On the other hand if the sub-expression was not encountered before, the *Memo* table is extended with an entry for this sub-expression.

This is captured in the function *insertIfNew*, which is our application specific version of *newSRef*. Its argument is the sub-expression that is being visited. Its result is a *TrafoCSE* with as input the transformed version of the sub-expression, which has type (*Expr a s*). The output is a reference to the transformed version of the first occurrence of the sub-expression.

```
insertIfNew ::    ∀s a env . Expr a env
             →    TrafoCSE env s (Expr a s) (Ref a s)
insertIfNew e =
  Trafo
  (λ(Memo m :: Memo env env′) → case m e of
     Nothing → TrafoE (extMemo e (Memo m)) extEnv
     Just r  → TrafoE (Memo m) (castSRef r)
  )
extMemo :: Expr a env → Memo env env′
        → Memo env (env′, a)
extMemo e (Memo m)
        = Memo (λs → case equals e s of
                   Just Eq → Just Zero
                   Nothing → fmap Suc (m s)
               )
```

The first time we encounter a sub-expression it is not found in the *Memo*-table (i.e. the *Nothing*-case above). Firstly the transformed version of the sub-expression is appended to the transformed declarations using *extEnv*. The *Memo* table is extended (using *extMemo*) with an entry mapping the current sub-expression to *Zero*, so for a next occurence of the sub-expression we know where to find the transformed first occurrence. Because we added one declaration ourselves, one extra *Suc* is added to the rest of the references in the *Memo* table.

For every subsequent encounter of the sub-expression, we find it in the *Memo* table (i.e. the *Just* case above). The reference to the first occurrence is simply the one found in the *Memo* table. We apply the function *castSRef* to take into account the declarations that might be added by future transformation steps.

The function *app_cse* [2] applies the *CSE* transformation to a single expression. The resulting *TrafoCSE* has as arrow input a *Ref*-transformer, that maps references from the original sequence of declarations to corresponding ones pointing into the transformation result. As output the *TrafoCSE* yields a reference to the transformed expression.

```
app_cse :: Expr a env
        → TrafoCSE env s (T env s) (Ref a s)
app_cse (Var r) = proc (T tenv_s) →
                     returnA ≺ tenv_s r
```

The reference inside a variable is transformed by applying the supplied *Ref*-transformer. The transformed reference now points to the corresponding value in the transformation result.

```
app_cse e@(IntVal i) = proc _ →
                          insertIfNew e ≺ IntVal i
```

For integer constants the function *insertIfNew* is applied to the original expression (e). As transformed expression *IntVal i* is passed. The function *insertIfNew* only inserts this expression if the integer constant is not already presented in the transformation result.

```
app_cse e@(LessThan x y)
   = proc tt →
     do l ← app_cse x ≺ tt
        r ← app_cse y ≺ tt
        insertIfNew e ≺ LessThan (Var l) (Var r)
   ...
```

For the constructor *LessThan* the function *app_cse* is applied recursively resulting in references to the transformed sub-expressions. The *Ref*-transformer *tt* is passed for both sub-expressions. Again *insertIfNew* is applied to the original expression; as transformed expression we pass a *LessThen* node containing the references to the transformed sub-expressions.

The implementations of *app_cse* for the constructors *BoolVal*, *Add*, and *If* are very similar, and are therefore omitted.

The function *app_cse* defined above applies the *CSE* transform to a single expression only. The final transformation should transform a sequence of declarations, which is encoded as a value of the data type *Env*. Therefore we define *cse_env*, which takes an *Env* as argument and applies *app_cse* to each expression. Analogous to *app_cse*, the function *cse_env* takes a *Ref*-transformer as input. It collects all the references returned by *app_cse* in an *Env*. This collection contains for each reference of the original declarations a corresponding reference in the transformation result.

```
cse_env :: Env Expr env env'
        → TrafoCSE env s
                   (T env s)
                   (Env Ref s env')
cse_env Empty = proc _ → returnA ≺ Empty
cse_env (Ext es e) = proc tt →
   do renv ← cse_env es ≺ tt
      r    ← app_cse e  ≺ tt
      returnA ≺ Ext renv r
```

The collection of *Ref*s returned by the function *cse_env* can be used to compute the *Ref*-transformer that it requires as input:

```
refTransformer :: Env Ref s env → T env s
refTransformer refs = T (λr → lookupEnv r refs)
```

The result of *cse_env* is used to compute its own input. To construct such a feedback-loop, we use the special **mdo**-notation for mutually recursive *Arrow* statements.

```
trafo :: Decls env → TrafoCSE env s () (T env s)
trafo decls = proc _ →
                 mdo let tt = refTransformer refs
                         refs ← cse_env decls ≺ tt
                         returnA ≺ tt
```

Finally we present the function *cse* which simply runs the *trafo* and extracts the result:

```
cse :: ∀env . Decls env → TDecls env
cse decls
  = case runTrafo (trafo decls) emptyMemo () of
        Result _ t env → TDecls env t
```

5. Alternative implementation for *runTrafo*

Recall the data type *Trafo* and the function *runTrafo*:

```
data Trafo m t s a b =
   Trafo (∀env1 . m env1 → TrafoE m t s a b env1)
data TrafoE m t s a b env1 =
  ∀env2 . TrafoE
          (m env2)
          (a → T env2 s → Env t s env1 →
             (b, T env1 s, Env t s env2)
          )

runTrafo :: ∀m t a b . (∀s . Trafo m t s a (b s))
                          → m () → a → Result m t b
runTrafo trafo m a =
  let Trafo trf     = trafo
      TrafoE m2 f = trf m
  in case f a (T id) Empty of
       (b, _, env) → Result m2 b env
```

In the definition of *runTrafo* we want the type of the final environment (s) to be the same as the type of the environment coming out of the transformation (*env2*). To achieve this the universally quantified s must be instantiated as *env2*. For this the use of lazy pattern binding (using **let**) on the existential data type (*TrafoE*) is essential. Unfortunately GHC, the most widely used Haskell Compiler, does not support lazy pattern matching on data constructors with existential types. In such cases it reports the infamous "My brain just exploded" error message. Other compilers such as Hugs(Hugs) and EHC(EHC) do support lazy pattern matching on data constructors with existential types. The reason this is not supported by GHC, is because it cannot be translated into GHCs intermediate language, which is based on System-F. GHCs core language should be extended with some kind of fix-point operator at the type level. However, this has as disadvantage that type level

[2] The following functions use arrow notation (Paterson 2001)

terms may be non-terminating, and therefore type terms can no longer be simply erased.

Using *unsafeCoerce* is a simple solution for this problem:

```
unsafeCoerce :: a → b
runTrafo :: (∀s . Trafo m t s a (b s)) → m () → a
         → Result m t b
runTrafo trafo m a = case trafo of
  Trafo trf → case trf m of
    TrafoE m2 f →
      case f a (T unsafeCoerce) Empty of
        (rb, tt, env2) →
          Result (unsafeCoerce m2)
                 rb
                 (unsafeCoerce env2)
```

The function is named unsafe for a good reason; it effectively switches off the typer checker. We believe this implementation of the function *runTrafo* is safe though. We could not find any examples where the use of use *runTrafo* goes wrong. Furthermore, the implementation is operationally identical to the original implementation of *runTrafo*, which is considered type correct according to other compilers than GHC. However, in a paper on typed transformations the use of *unsafeCoerce* feels a bit like cheating. Therefore we also provide, below, a version that is free of both *unsafeCoerce* and lazy pattern matching on existential types. With this solution, however, the *Trafo* type is no longer a real *Arrow*, and hence the special *Arrow* notation cannot be used.

In the type of *runTrafo* above the universal quantification on s is on the outside of the type *Trafo*, whereas the existential quantification on *env2* is inside. Instantiating s with *env2* would be much easier if this was the other way around. We may move the universal quantification over s inside the quantification over *env2*. This would give us the following type:

```
data Trafo2 m t a b =
  Trafo2 (∀env1 . m env1 → TrafoE2 m t a b env1)
data TrafoE2 m t a b env1 =
  ∀env2 . TrafoE2
           (m env2)
           (∀s . a s → T env2 s → Env t s env1
                 → (b s, T env1 s, Env t s env2)
           )
```

Note that the type variables a and b are now labelled with s, and hence have kind ($* \rightarrow *$). This is essential because we want to manipulate terms and *Ref*s which are labelled with type s. For example the type of *newRef* which used to be:

```
newSRef :: Trafo m t s (t a s) (Ref a s)
```

now becomes:

```
newSRef2 :: Trafo2 m t (t a) (Ref a)
```

The implementation of *runTrafo* on the new *Trafo2* type is fairly straightforward:

```
runTrafo2 :: Trafo2 m t a b → m () → (∀s . a s)
          → Result m t b
runTrafo2 trafo m a =
  case trafo of
    Trafo2 trf → case trf m of
      TrafoE2 m2 f →
        let (rb, tt, env2) = f a (T id) Empty
        in  Result m2 rb env2
```

Unfortunately the new data type *Trafo2* is not really an *Arrow*, because the type variables a and b are of kind $* \rightarrow *$ instead of $*$. We can however provide an *Arrow*-style interface for programming with the type *Trafo2*, by making it instance of the following class:

```
class Arrow2 arr where
  arr2    :: (∀s . a s → b s) → arr a b
  (>>>>)  :: arr a b → arr b c → arr a c
  first2  :: arr a b → arr (Pair a c) (Pair b c)
  second2 :: arr a b → arr (Pair c a) (Pair c b)
  (****)  :: arr a b → arr a' b'
          → arr (Pair a a') (Pair b b')
  (&&&&)  :: arr a b → arr a b' → arr a (Pair b b')

newtype Pair a b s = P (a s, b s)
```

Although the combinators above do not define a real *Arrow*, programming with them is the same as programming with *Arrow*s, except that one cannot use the special *Arrow* syntax (Paterson 2001). This is unfortunate, because the special syntax make programming with *Arrow*s a lot easier.

6. Conclusion

We have shown how to use the Haskell type system and its extensions to perform a fully typed program transformation. Doing so we have used a wide variety of type system concepts: placing existentials precisely at the positions where needed, making things polymorphic where needed, using loop combinators to feed back the result of the computation into the computation inside the scope of an existential, using GADTs to type the environments we construct, scoped type variables, splitting the type labels of the environment into a *use* and a *def* part and thus temporarily decoupling the types of the occurring references and the types associated with the terms in the environment being constructed. We introduced an arrow like style for composing the transformations. Besides this we make use of lazy evaluation in order to get computed information to the right places to be used.

We think that studying the algorithm and its approaches to the various subproblems is indispensable for anyone who wants to program similar transformation-based algorithms in a strongly typed setting. Some might wonder why the approach taken may be necessary at all, and why not resort to off-line techniques, and they have a point. It is often easier to work in an untyped setting, only to check the generated result afterwards for type correctness. On the other hand one can see the added complexity as a partial correctness proof of the transformation, and as we all know proofs of correct lemmas are superfluous.

We believe that the arrow-based library will turn out to be useful in building programs that transform typed abstract syntax, and that the pattern we have followed in this paper will be followed in many more interesting applications to come.

It is unfortunate that GHC does not support lazy pattern matching on data constructors with existential types. We hope this will be supported in the future. Until then, a user of the library is posed the following dillema: either have an *unsafeCoerce* in the implementation of *runTrafo*, or use the alternative *Trafo* type, but loose the convenience of the *Arrow*-notation.

References

Arthur Baars and S. Doaitse Swierstra. Typed transformations of typed abstract syntax. UU-CS 21, Utrecht University, 2008.

Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In S. Peyton Jones, editor, *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 157–166. ACM Press, 2002. ISBN 1-58113-487-8.

Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self-inspecting code. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 69–79, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-850-4.

Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. TTTAS HackageDB package. URL `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/TTTAS`.

Chiyan Chen and Hongwei Xi. Implementing typeful program transformations. In *PEPM'03*, 2003.

James Cheney and Ralf Hinze. First-Class Phantom Types. Technical report TR2003-1901, Cornell University, 2003. `http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.c%is/TR2003-1901`.

EHC. Essential haskell compiler. URL `http://www.cs.uu.nl/wiki/Ehc`.

GHC. Glasgow haskell compiler. URL `http://www.haskell.org/ghc/`.

Hackage. Hackage. URL `http://hackage.haskell.org/`.

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 2000.

Hugs. Hugs. URL `http://www.haskell.org/hugs/`.

John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–35, 1994. URL `citeseer.ist.psu.edu/article/launchbury93lazy.html`.

Emir Pasalic and Nathan Linger. Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering (GPCE'04)*, volume LNCS 3286, pages 136 – 167, October 2004.

Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, 2006. ISSN 0362-1340.

Marcos Viera, S. Doaitse Swierstra, and Eelco Lempsink. Haskell do you read me? constructing and composing efficient top-down parsers at runtime. In A. Gill, editor, *Haskell Symposium*. ACM, 2008.

Stephanie Weirich. Type-safe cast. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 58–67. ACM press, 2000. ISBN 1-58113-202-6.

A. Transformation library

A.1 Data types

```
data Equal :: ∗ → ∗ → ∗ where
  Eq :: Equal a a

data Ref a env where
  Zero ::                 Ref a (env', a)
  Suc  :: Ref a env' → Ref a (env', b)

data Env term use def where
  Empty  :: Env t use ()
  Ext    :: Env t use def' → t a use
         → Env t use (def', a)

type FinalEnv t usedef = Env t usedef usedef

data Result m t b
  = ∀s . Result (m s) (b s) (FinalEnv t s)

newtype T e s = T{unT :: ∀x . Ref x e → Ref x s}

data Unit s = Unit

data Trafo m t s a b =
  Trafo (∀env1 . m env1 → TrafoE m t s a b env1)
data TrafoE m t s a b env1 =
        ∀env2 . TrafoE
          (m env2)
          (a → T env2 s → Env t s env1
            → (b, T env1 s, Env t s env2)
          )
```

A.2 Functions

```
match :: Ref a env → Ref b env → Maybe (Equal a b)

lookupEnv :: Ref a env → Env t s env → t a s

updateEnv :: (t a s → t a s) → Ref a env
          → Env t s env → Env t s env

newSRef :: Trafo Unit t s (t a s) (Ref a s)

extEnv :: t a s → T (e, a) s → Env t s e
     → (Ref a s, T e s, Env t s (e, a))

castSRef :: Ref a env
         → (x       → T env s → Env t s env
         → (Ref a s ,  T env s ,  Env t s env))

runTrafo :: (∀s . Trafo m t s a (b s)) → m () → a
         → Result m t b
```

A.3 *Arrow* interface

```
arr :: (a → b) → Trafo m t s a b

(>>>) :: Trafo m t s a b → Trafo m t s b c
      → Trafo m t s a c

first :: Trafo m t s a b → Trafo m t s (a, c) (b, c)

second :: Trafo m t s b c → Trafo m t s (d, b) (d, c)

(***) :: Trafo m t s b c → Trafo m t s b' c'
      → Trafo m t s (b, b') (c, c')

(&&&) :: Trafo m t s b c → Trafo m t s b c'
      → Trafo m t s b (c, c')

loop :: Trafo m t s (a, x) (b, x) → Trafo m t s a b
```

A.4 *Trafo2*

```
data Trafo2 m t a b =
  Trafo2 (∀env1 . m env1 → TrafoE2 m t a b env1)
data TrafoE2 m t a b env1 =
  ∀env2 . TrafoE2
          (m env2)
          (∀s . a s → T env2 s → Env t s env1
                → (b s, T env1 s, Env t s env2)
          )

newSRef2 :: Trafo2 m t (t a) (Ref a)
```

```
runTrafo2 :: Trafo2 m t a b → m () → (∀s . a s)
          → Result m t b

class Arrow2 arr where
  arr2    :: (∀s . a s → b s) → arr a b
  (>>>>)  :: arr a b → arr b c → arr a c
  first2  :: arr a b → arr (Pair a c) (Pair b c)
  second2 :: arr a b → arr (Pair c a) (Pair c b)
  (****)  :: arr a b → arr a' b'
          → arr (Pair a a') (Pair b b')
  (&&&&)  :: arr a b → arr a b' → arr a (Pair b b')

newtype Pair a b s = P (a s, b s)
```