A Lazy Language Needs a Lazy Type System

Introducing Polymorphic Contexts

S. Doaitse Swierstra
Utrecht University
Utrecht, The Netherlands
doaitse@swierstra.net

Marcos Viera
Universidad de la Republica
Montevideo, Uruguay
mviera@fing.edu.uy

Atze Dijkstra
Standard Chartered
London, UK
atze@atzedijkstra.net

ABSTRACT

Most type systems that support polymorphic functions are based on a version of System-F. We argue that this limits useful programming paradigms for languages with lazy evaluation. We motivate an extension of System-F alleviating this limitation.

First, using a sequence of examples, we show that for lazily evaluated languages current type systems may force one to write a program in an unnatural way; in particular, we argue that in such languages the relationship between polymorphic and existential types can be made more systematic by allowing to pass back (part of) an existential result of a function call as an argument to the function call that produced that value.

After presenting our extension to System-F we show how we can implement the strict-state thread monad *ST* by using a returned existential type to instantiating a polymorphic function that returns that type. Currently this monad is built-in into the runtime system of GHC and as such has become part of the language.

Our proposed language extension, i.e. the introduction of *polymorphic contexts*, reverses the relationship between the context of a function call and the called function with respect to where it is decided with which type to instantiate a type variable.

CCS CONCEPTS

• **Theory of computation → Functional constructs**; **Type structures**;

KEYWORDS

type systems, lazy evaluation, existentials, System-F, state monad

ACM Reference format:
S. Doaitse Swierstra, Marcos Viera, and Atze Dijkstra. 2017. A Lazy Language Needs a Lazy Type System. In *Proceedings of IFL Conference, Leuven, Belgium, August 31-September 2, 2016 (IFL 2016),* 12 pages.
DOI: http://dx.doi.org/10.1145/3064899.3064906

1 INTRODUCTION

In a strict language a value of type

$\exists\, x\ .\ x \rightarrow Int \rightarrow (x, Int)$

does not make much sense; the only way to call such a function is by passing the x returned as part of the result, but this will lead to non-termination. In languages with lazy evaluation and **letrec** bindings we can however provide such an useful argument by passing part of the computed result[1]:

$$\begin{array}{l}\textbf{let}\ f :: \exists\, x\ .\ x \rightarrow Int \rightarrow (x, Int) = \ldots \\ \textbf{in}\ \ \textbf{let}\ (x, v) = f\ x\ 3 \\ \quad\ \ \textbf{in}\ \ v\end{array}$$

We claim that conventional System F based way of introducing existential types [10] excludes some useful programming paradigms and thus we propose a small extension to System F. The larger part of this paper consists of two examples in which we show how to put our extension to good use. We show in Section 2 how the current way of dealing with existential types in Haskell (GHC) forces undesirable strictness on our programs and can make our programs unnecessarily complicated. Next we discuss our type system in Section 3. We then proceed by showing in Section 4 how we can encode the *ST*-monad, so there is no longer the need to have it built into the language. We finish with some possible extensions, discussion and conclusions.

2 BEING LESS STRICT

In order to explain what kind of programs we should like to write, we start out with the *repmin* problem [5]. We start with a lazy version and convert that into a strict version that performs the same number of pattern matches. Next we write a similar version of an identity function. When we convert that function back to a lazy version, similar to the first version of *repmin* we run into a typing problem.

2.1 *repmin*

The challenge is to write a function $repmin :: Tree \rightarrow Tree$ which returns a binary tree with the same shape as the argument tree, but with the leaf values replaced by the minimum of the original leaf values. A straightforward solution, which also can be seen as a specification of the problem, is given in Figure 1. Note that in this solution no use of lazy evaluation is made and that each node of the tree is inspected twice in a pattern match during the computation.

The reason that this problem has drawn a lot of attention is that, provided the programming language supports lazy evaluation (call by need), the result can be computed by inspecting each constructor of the argument tree only once (Figure 2): in the function *repmin′* we tuple the computation of the minimal leaf value with the construction of the resulting tree. The latter uses the computed

[1]As in Haskell our **let**'s are to be interpreted as **letrec**'s

IFL 2016, Leuven, Belgium
© 2016 ACM. 978-1-4503-4767-9/16/08…$15.00
DOI: http://dx.doi.org/10.1145/3064899.3064906

```
data Tree = Leaf Int
          | Bin Tree Tree
repmin t = let m = minval t in replace t m
minval :: Tree → Int
minval (Leaf v)    = v
minval (Bin l r)   = minval l `min` minval r
replace :: Tree → Int → Tree
replace (Leaf _) m = Leaf m
replace (Bin l r) m = replace l m `Bin` replace r m
```

Figure 1: ***repmin*** **in a strict language**

```
data Tree = Leaf Int | Bin Tree Tree
repmin :: Tree → Tree
repmin t = let (m, r) = repmin′ t m in r
repmin′ :: Tree → Int → (Int, Tree)
repmin′ (Leaf v) m = (v, Leaf m)
repmin′ (Bin l r) m = (ml `min` mr, tl `Bin` tr)
                      where (ml, tl) = repmin′ l m
                            (mr, tr) = repmin′ r m
```

Figure 2: ***repmin*** **in a lazy language**

```
repmin2 :: Tree → Tree
repmin2 t = let (m, reconstruct) = repmin2′ t
            in reconstruct m
repmin2′ :: Tree → (Int, Int → Tree)
repmin2′ (Leaf v) = (v, Leaf)
repmin2′ (Bin l r) = (ml `min` mr
                     , λm → tfl m `Bin` tfr m)
                     where (ml, tfl) = repmin2′ l
                           (mr, tfr) = repmin2′ r
```

Figure 3: strict version of ***repmin2***

minimal value which is passed as the parameter *m*. In the top function *repmin* the minimal value computed by *repmin′* is *passed back* as argument to the same call of *repmin′* as the value to be bound to argument *m*. Such programs are referred to as *circular programs*.

If we perform a global flow analysis of the program, inspired by analyses from the attribute-grammar world [4, 6, 8], we discover that no information flows from the *Int* parameter to the *Int* part of the result. This implies that we can replace the type *Tree* → *Int* → (*Int*, *Tree*) with the type *Tree* → (*Int*, *Int* → *Tree*), provided the program is adapted accordingly. Figure 3 shows the result of this transformation. The *Int* part of the result again contains the computed minimal value and the *Int* → *Tree* part is a function that constructs the sought tree from the passed minimal value; we have "remembered" the shape of the argument tree in that function.

This code maintains the characterizing property of our lazy *repmin* solution: each constructor of the tree is only inspected once in a pattern match; the order in which values are to be evaluated however has been made more explicit (although lazy evaluation still evaluates them in the same order!). Note that the first version of *repmin* depends essentially on lazy evaluation (the **let** actually is a **letrec** in Haskell), whereas *repmin2*, despite being here written in Haskell, could straightforwardly be transcribed into a strict language like ML.

2.2 *idTree*

The next step in introducing our problem is that instead of writing a *repmin* function we want to write an identity function of type *Tree* → *Tree*. In later sections we will provide variations on this example to make it more interesting, but for the time being it serves its purpose. Where *repmin2* computed an intermediate representation holding the minimal value of the leaves tupled with a function which remembered the *shape* of the tree, our first identity function *idTree2* (Figure 4) uses a similar intermediate structure which contains all the values stored in the leaves of the original tree in a nested Cartesian product tupled with a tree-reconstruction function. The latter function, as before, has remembered the shape of the argument tree and, once provided with the leaf values that were harvested from that tree, reconstructs that very tree. Since, in contrast to the *repmin2* function, *the type of the intermediate result depends on the shape of the tree*, we have introduced an existential type *vs* in this intermediate representation.

To make explicit what is going on we introduce some notation to make the places where existential values are constructed and deconstructed explicit using the *pack/unpack* paradigm [10]. In a *packing* expression $\lessdot t, e \gtrdot$ the *t* denotes the existential type and the *e* a value of a type in which this type may occur. The *unpack* function is implicitly called by using pattern matching (following Pierce [12]); binding to a $\lessdot t_v, v \gtrdot$ pattern makes that the a freshly new type constant is bound to the type variable t_v and the value part to the variable *v*.

When pairing the two parts of the result in the branches of *idTree2′* we thus hide whether we combine an *Int* value with a function of type *Int* → *Tree* as in the first alternative of *idTree2′* or a pair of existentially typed values returned by the recursive function calls with a function taking such a pair as in the second alternative. When we unpack the packed value in *idTree2* using pattern matching we can however be sure that it is safe to apply the function to its accompanying value, because this was the case when we packed them together.

Now suppose we have a language with lazy evaluation and that we prefer the lazy version of *repmin* over *repmin2*, and thus we set out to define a similar *idTree* (Figure 5), in which we do not use an intermediate representation.

We now run into a problem, since this program is type-incorrect and we cannot provide a type for *idTree′*. In the first alternative of *idTree′* the argument *w* is of type *Int*, whereas in the second alternative the type is a pair of values, and these two types do not unify. So, why is the first version of *repmin* permitted, and is our corresponding version of *idTree* is rejected? When checking the types at runtime we do not run into problems.

```
idTree2 :: Tree → Tree
idTree2 t =  let ≺ t_vs, (vs, reconstruct)≻ = idTree2′ t
             in reconstruct vs
idTree2′ :: Tree → ∃ vs . (vs, (vs → Tree))
idTree2′ (Leaf v) =  ≺ Int, (v, Leaf) ≻
idTree2′ (Bin l r)
  =  ≺ (t_vsl, t_vsr),
        ((vsl, vsr), (λ(vsl, vsr) → tfl vsl ‘Bin‘ tfr vsr)) ≻
     where ≺ t_vsl, (vsl,  tfl) ≻  = idTree2′ l
           ≺ t_vsr, (vsr, tfr) ≻  = idTree2′ r
```

Figure 4: *idTree2*

```
idTree :: Tree → Tree
idTree t = let (vs, r) = idTree′ t vs in r
idTree′ (Leaf v) w
        = (v,           Leaf w)
idTree′ (Bin l r) ~(vsl′, vsr′)
        = ((vsl, vsr), tl ‘Bin‘ tr)
              where (vsl, tl) = idTree′ l vsl′
                    (vsr, tr) = idTree′ r vsr′
```

Figure 5: A type incorrect *idTree*

```
idTree :: Tree → Tree
idTree t = let ≺ t_vs, f ≻ = idTree′ t
               (vs, r) = f vs
           in r
idTree′ :: Tree → ∃ vs . vs → (vs, Tree)
idTree′ (Leaf v) =  ≺ Int, (λw → (v, Leaf w)) ≻
idTree′ (Bin l r) =
   let ≺ t_vsl, fl ≻    = idTree′ l
       ≺ t_vsr, fr ≻    = idTree′ r
   in  ≺ (t_vsl, t_vsr),
         (λ~(vsl′, vsr′) → let (vsl, tl)  = fl vsl′
                               (vsr, tr)  = fr vsr′
                           in  ((vsl, vsr), tl ‘Bin‘ tr)) ≻
```

Figure 6: *idTree***, computing the types first**

It appears that we actually are in need of a dependent type; the type of the returned structure containing the leaf values, and thus the type of the second argument which holds the leaf values from which the tree is to be reconstructed, entirely depends on the shape of the argument tree. By rewriting the code, and making use of some explicit lambda's we reach the solution given in Figure 6.

Unfortunately this version cannot be transcribed into GHC while keeping its semantics. One of GHC's design decisions has been to forbid irrefutable patterns, and thus all pattern matching for existential data types has to be strict using a **case** construct, which

```
top (Bin _ _) = "Bin"
top (Leaf  _) = "Leaf"
main = let infTree = Bin infTree infTree
           in print . top . idTree $ inftree
```

Figure 7: GHC version is too strict

excludes the use of **let**-based bindings with existential types as their right hand side. In requiring strict pattern matching the "computation" of the complete type is enforced before being able to pack it with the existential construct, and we have thus silently changed the semantics of our original *idTree*. This is demonstrated by the program in Figure 7. The GHC solution will not terminate since it enforecs a complete traversal of an infinite tree trying to construct an infinite type representing the shape of the infinite tree argument. In this sense the GHC solution is not an honest realization of the identity function.

A more serious, methodological shortcoming, of this last version of *idTree* is that we had to separate the computation of the type from the computation of the values: first we inspect the tree, this gives us the types and constructs the computations to be performed, and then we perform the computations. So can we make Figure 5 type check?

2.3 $\bar{\exists}$

One way of looking at the type-incorrect *idTree* is not to see the type of the gathered leaves as something which is computed by inspecting the parameter tree, but as something which is computed by the function too, and returned as part of its result; the function then expects a value of this returned type as a lazy evaluated argument. Intuitively the type of *idTree′* is something like:

$$idTree' :: Tree \rightarrow t_{vs} \rightarrow \exists\ t_{vs}\ .\ (t_{vs}, Tree)$$

This notation reflects the operational idea that the function *idTree′* not only returns a value of type t_{vs}, but also the type t_{vs} itself. The notation is however a bit unconventional since the t_{vs} in the argument position is also supposed to be introduced by this $\exists$ quantifier, which is unfortunately not reflected in the notation at all. To overcome this problem we introduce a new quantifier $\bar{\exists}$, which is to be interpreted as the specification above; hence it introduces a type variable which is bound to a type computed as part of the result of the function, but which scopes over (part of) the signature too:

$$idTree' :: Tree \rightarrow \bar{\exists}\ t_{vs}\ .\ t_{vs} \rightarrow (t_{vs}, Tree)$$

By extending the scope to an earlier position in the list of arguments, we express that both the t_{vs} argument and the t_{vs} part of the result are to be the same for each call to *idTree′*, but furthermore opaque at the calling position.

Before paying attention to the precise type rules relating to $\bar{\exists}$ we will give yet another example of its usefulness.

2.4 *sortTree*

As a next step in showing that it may be undesirable to separate the computation of the type from the computation of part of the result

```
idTree t = let ⋖ t_vs, (vs, r)⋗ = idTree′ vs
           in r

idTree′ :: Tree → ∃̄ vs . (vs → (vs, Tree))
idTree′ (Leaf v)
   = ⋖Int, (λw → (v, Leaf w)) ⋗
idTree′ (Bin l r)
   = λ~(vsl′, vsr′) →
        let ⋖ t_vsl, (vsl, tl) ⋗  = idTree′ l vsl′
            ⋖ t_vsr, (vsr, tr) ⋗  = idTree′ r vsr′
        in  ⋖ (t_vsl, t_vsr), ((vsl, vsr), tl ‘Bin‘ tr) ⋗
```

Figure 8: *idTree* using ∃̄

```
sortTree t = let (vs, [ ], res) = sortTree′ t [ ] vs
             in  res

insert v [ ] = [v]
insert v (w : ws) = if v < w then v : w : ws
                              else  w : insert v ws

sortTree′ :: Tree → [Int] → [Int] → ([Int], [Int], Tree)
sortTree′ (Leaf v) rest ~(x : xs) = (insert v rest, xs, Leaf x)
sortTree′ (Bin l r) rest xs
  = let (vl, xsl,  tl) = sortTree′ l vr   xs
        (vr, xsr,  tr) = sortTree′ r rest xsl
    in  (vl, xsr,  Bin tl tr)
```

Figure 9: Sorting using lists

we modify our *idTree* example by requiring that the leaf values from the original tree are to be reordered in such a way that a prefix traversal of the resulting tree finds and increasing list of leaf values; the resulting tree however should have the same shape again.

In order to explain our algorithm we first give a version (Figure 9) in which we use conventional lists to represent collected leaf values. In order to avoid expensive concatenations we thread two list values through the tree: one in a backwards direction in which we collect the leaf values, and one in a forwards direction from which we take leaf values. At the top level the constructed first value is used to initialise the second one.

The helper function *sortTree′* takes as arguments:

- the *Tree* to be sorted
- the *sorted list rest* containing the leaf values following the node in a prefix traversal
- the list *xs* of values still not used in building the result tree.

It returns:

- a sorted list of leaf values containing the leaf values of this node and the leave values following the node at hand in a prefix traversal (i.e. it adds its contained leaves to its second argument),
- a tail of the parameter *xs* containing the values to be used in constructing the rest of the tree
- the final tree constructed from the prefix of its *xs* argument

```
data OrdList cl where
  OrdList :: cl → (Int → cl → (Int, cl)) → Ordlist

sortTree t = let ⋖ t_vs, (OrdList vs, _, res) ⋗
                   = sortTree″ t
                     (OrdList () (λx () → (x, ()))) vs
             in res
sortTree″ :: Tree → ∀ rest . OrdList rest
                   → ∃̄ xs . xs → (OrdList xs, rest, Tree)
sortTree″ (Leaf v) (OrdList (rest :: t_rest) insert)~(x, xs)
    =      ⋖ (Int, t_rest),
             (OrdList
                (ins v rest)
                (λw (x, xs) → if w < x then (w, (x, xs))
                                       else (x, insert w xs))
           , xs, Leaf x) ⋗
sortTree″ (Bin l r) (rest :: t_rest) xs
    = let ⋖ t_l, (vl, xsl, tl) ⋗
             = sortTree″ l (vr :: t_r) xs
          ⋖ t_r, (vr, xsr, tr) ⋗
             = sortTree″ r (rest :: t_rest) xsl
      in  ⋖ t_l, (vl, xsr, Bin tl tr) ⋗
```

Figure 10: Sorting using Cartesian products

Our next step (Figure 10) is to replace the intermediate lists with nested Cartesian products. This guarantees that the top-level *sortTree* function cannot cheat, e.g. by secretly replacing elements in the list of leaf values, and each node adds exactly one element to the list and removes exactly one element

The data type *Ordlist* represents a sorted list to which elements can be added using the function part it carries. This makes more explicit what is going on. We have used scoped type variables to get hold of the type of the parameter *rest* and used explicit type annotations in $vr :: t_r$ to indicate how the polymorphic type *rest* in the calls is to be instantiated. Note that the returned type does not simply depend on the shape of the argument tree anymore, but also on the polymorphic type *rest*!

3 POLYMORPHIC CONTEXTS

3.1 The $\bar{\exists}$ quantifier

By inspecting at the uses of $\bar{\exists}$ in both examples we see that the types constructed do not play a role at all at the place where they are fed back into the computation; they only serve to describe the type of part of the result of the function which is used to pass back an argument of that type. This suggests that it is more natural to compute this type as part of the result and use a pattern-matching **letrec** construct to get access to this type; thus making it possible to enforce that the value passed back is described by the computed type. It is here that we deviate from the standard way of dealing with existential values. We observe that once all arguments to the function call have been given the computed type in principle is fully determined, and could be computed by the callee. This might however imply that we have to split the computation: one part in which we just do enough work to compute the type, and once the type has been determined the rest of the computation to compute its associated value, as in Figure 6. We thus distinguish between a type being *computable*, meaning that all information needed to compute it is available, and *computed*, meaning that it has been computed from this available information.

This brings us to the most important message of this paper; we are dealing with a situation in which *the rôles of the context in which a function is called and the function itself are reversed with respect to which side of the function call decides how types are to be instantiated*:

- when calling a *polymorphic function* it is the *context* which decides on the type of the polymorphic argument and it is the duty of the function to return a value consistent with that type.
- a *function* requiring a *polymorphic context* decides on the type returned by the call and it is the duty of the context to pass back an argument consistent with that type.

For the callee it looks like calling into a polymorphic context. In our *idTree* example the context behaves as a function of type $\forall\ a\ .\ a \rightarrow a$.

We may wonder what is the correct place to insert the $\bar{\exists}$ quantifier. Just as the type rules of System-F can be used to show that there is no essential difference between the types $\forall\ a\ .\ Int \rightarrow a \rightarrow a$ and $Int \rightarrow \forall\ a\ .\ a \rightarrow a$, we do not distinguish between $\bar{\exists}\ t_{vs}\ .\ Tree \rightarrow t_{vs} \rightarrow (t_{vs}, Tree)$ and $Tree \rightarrow \bar{\exists}\ t_{vs}\ .\ t_{vs} \rightarrow (t_{vs}, tree)$.

3.2 Type rules

$e\ = v$	-- variable
$\mid\ e\ e$	-- application
$\mid\ \lambda(v : \sigma) \rightarrow e$	-- abstraction
$\mid\ \bar{\exists}\ \alpha\ .\ e$	-- introduction
$\mid\ e\ [v]$	-- elimination
$\mid\ \lessdot (\alpha = \sigma), e \gtrdot$	-- pack
$\mid\ e_1\ [\!]\ \dots\ [\!]\ e_n$	-- function alternatives
$\mid\ \mathbf{let}\ p = e_1\ \mathbf{in}\ e_2$	-- local binding
$\mid\ \dots$	-- other terms
$p\ = v$	-- variable
$\mid\ \lessdot v, p \gtrdot$	-- unpack
$\mid\ \dots$	-- other patterns
$\sigma = \alpha$	-- type variable
$\mid\ v$	-- type name from the program
$\mid\ \sigma \rightarrow \sigma$	-- abstraction
$\mid\ \forall\ \alpha\ .\ \sigma$	-- universal quantification
$\mid\ \bar{\exists}\ \alpha\ .\ \sigma$	-- our existential
$\mid\ \lessdot (\alpha = \sigma), \sigma \gtrdot$	-- packed existential
$\mid\ \dots$	-- other types

Figure 11: Structures

In Figure 11 we have given the underlying syntax of our extension, as far as it deviates from standard System-F. We have made

$$\begin{array}{l}
idTree :: Tree \rightarrow Tree \\
idTree\ t = \mathbf{let}\ \lessdot t_{vs}, (vs, r) \gtrdot\ = idTree'\ [t_{vs}]\ t\ vs\ \mathbf{in}\ r \\
idTree' :: \bar{\exists}\ t_{vs}\ .\ Tree \rightarrow t_{vs} \rightarrow (t_{vs}, Tree) \\
idTree' = idTree'_{Leaf}\ \ [\!]\ \ idTree'_{Bin} \\
idTree'_{Leaf}\ = \bar{\exists}\ t_{vs}\ . \\
\qquad\qquad \lambda(Leaf\ \ v)\ w \rightarrow \lessdot (t_{vs} = Int), (v, Leaf\ \ w) \gtrdot \\
idTree'_{Bin}\ \ = \bar{\exists}\ t_{vs}\ . \\
\qquad\qquad \lambda(Bin\ l\ r) \sim\!(vsl', vsr') \rightarrow \\
\qquad\qquad\quad \mathbf{let}\ \lessdot t_{vsl}, (vsl, tl\,) \gtrdot\ = idTree'\ [t_{vsl}]\ l\ \ vsl' \\
\qquad\qquad\qquad\ \ \lessdot t_{vsr}, (vsr, tr) \gtrdot\ = idTree'\ [t_{vsr}]\ r\ vsr' \\
\qquad\qquad\quad \mathbf{in}\ \ \lessdot (t_{vs} = (t_{vsl}, t_{vsr})), \\
\qquad\qquad\qquad\quad ((vsl, vsr), tl\ \text{`}Bin\text{`}\ tr) \gtrdot
\end{array}$$

Figure 12: annotated *idTree* using $\bar{\exists}$

the underlying typing a lot more explicit than in the example code thus far, which was made to resemble Haskell as much as possible. In practice a lot of this information can be inferred, as we are assuming in a lot of our example code.

In Figure 12 we have rephrased our *idTree* function once more, but now with more explicit typing directives, and making clear how function alternatives can be represented. Our function *idTree′* is composed of alternatives $idTree'_{Leaf}$ and $idTree'_{Bin}$, each defined as a $\bar{\exists}$-quantified λ-expression. We require that all parameters with a type introduced by the $\bar{\exists}$ match lazily, so they cannot be inspected at runtime when matching the pattern. The alternatives are tried in order until a match occurs. The result of the function is constructed by explicitly packing a value with the corresponding type.

If we look at the definition of *idTree* we see that the expression returns an existentially typed value; the type part is given the name t_{vs}, and this is also the type which is to be used in typing the expressions. Since this very much resembles the way polymorphic functions are instantiated we have again borrowed notation from System-F in $idTree'\ [t_{vs}]$. To paraphrase Henry Ford[2] we say that the calling environment can pick any type as long as it is the returned t_{vs}, because that is the only way in which the returned result can match the left-hand side $\lessdot t_{vs}, (vs, r) \gtrdot$.

$$\boxed{\Lambda \vdash^{p} p : \sigma}$$

$$\frac{}{\Lambda,\ v \rightarrow \sigma \vdash^{p} v : \sigma}\ (\textsc{varpat})$$

$$\frac{\begin{array}{c} v\ \notin\ \Lambda \\ \Lambda,\ v \vdash^{p} p : [\alpha \mapsto v]\ \sigma \end{array}}{\Lambda,\ v \vdash^{p} \lessdot v, p \gtrdot : \lessdot (\alpha = v),\ \sigma \gtrdot}\ (\textsc{unpackpat})$$

Figure 13: Pattern

We start out with a new form of judgement for dealing with patterns in the environment. We extend the rules given by Harper

[2] "A customer can have a car painted any color he wants as long as it is black"

[7] with the form needed for our specific purpose, i.e. dealing with values packed with an existential type. To introduce notation we show the rule VARPAT which states that a variable is a proper pattern and adds a binding $v \mapsto \sigma$ to the environment. The rule UNPACKPAT states that if a pattern p introduces names in an environment Λ then so does $\lessdot v, p \gtrdot$. The existential type is given a name v, which is stored in the environment, in which it has to be unique.

$$\boxed{\Gamma \vdash^{e} e : \sigma}$$

$$\frac{\Gamma \vdash^{e} e : [\alpha \mapsto \sigma'] \; \sigma}{\Gamma \vdash^{e} \lessdot(\alpha = \sigma'), e \gtrdot : \lessdot (\alpha = \sigma'), \sigma \gtrdot} \; (\text{PACK})$$

$$\frac{\Gamma \vdash^{e} e : [\alpha \mapsto \sigma] \; \sigma' \rightarrow \ldots \rightarrow \lessdot(\alpha = \sigma), \sigma'' \gtrdot}{\Gamma \vdash^{e} \bar{\exists} \; \alpha \; . \; e : \bar{\exists} \; \alpha \; . \; \sigma' \rightarrow \ldots \rightarrow \sigma''} \; (\text{E.I})$$

$$\frac{\Gamma \vdash^{e} e : \bar{\exists} \; \alpha \; . \; \sigma' \rightarrow \ldots \rightarrow \sigma''}{\Gamma, v \vdash^{e} e \; [v] : [\alpha \mapsto v] \; \sigma' \rightarrow \ldots \rightarrow \lessdot(\alpha = v), \sigma'' \gtrdot} \; (\text{E.E})$$

$$\frac{\Gamma \vdash^{e} e_i : \sigma}{\Gamma \vdash^{e} (e_1 \; [\!] \ldots [\!] \; e_n) : \sigma} \; (\text{CHOICE}) \qquad \frac{\Lambda \vdash^{p} p : \sigma \quad \Gamma \, \Lambda \vdash^{e} e_1 : \sigma \quad \Gamma \, \Lambda \vdash^{e} e_2 : \sigma'}{\Gamma \vdash^{e} \textbf{let} \; p = e_1 \; \textbf{in} \; e_2 : \sigma'} \; (\text{LETREC})$$

Figure 14: Expression

In Figure 14 we have given the rules for our form of existential types. We are intentionally incomplete as we only wish to clarify the non-standard construct $\bar{\exists} \; \alpha \; . \; \sigma$ for our existential types in the context of the given examples. Our conjecture is that System-F cannot deal with the computational order in which types are computed (as a result) and passed back (into an earlier parameter), and hence we do not provide a translation to System-F. This is due to the System-F approach to unpacking, which uses a continuation style of formulation in e.g. Harper's book [7] **open** e **as** t **with** $x : \sigma$ **in** e', which does not allow x to be referred to in e.

Both the language for expressions e and types σ are open ended, describing the subset of what Haskell offers required for our examples. We furthermore require that all environments and the types therein are well-formed.

The rule PACK describes how we can forget (part of) a type and replace it by an existential type.

Rule E.I forgets about type σ' in σ by replacing it by a type variable α; rule E.E does the inverse, by reintroducing the forgotten type. Since we have no longer access to the original type we assume this to be the name of some anonymous type with name v.

To show how these rules can be used in typing our desired form of *idTree* we consider the various function alternatives. Due to the CHOICE rule, both alternatives must have type $\bar{\exists} \; t_{vs} \; . \; Tree \rightarrow t_{vs} \rightarrow (t_{vs}, Tree)$. We sketch the derivation of the first alternative:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, v \mapsto Int, w \mapsto Int \vdash^{e} (v, Leaf \; w) : (Int, Tree)} \; \text{VAR}(^{*}2)}{\begin{array}{c}\Gamma, v \mapsto Int, w \mapsto Int \vdash^{e} \lessdot(t_{vs} = Int), (v, Leaf \; w)\gtrdot \\ : \lessdot (t_{vs} = Int), (t_{vs}, Tree)\gtrdot\end{array}} \; \text{PACK}}{\begin{array}{c}\Gamma \vdash^{e} \lambda(Leaf \; v) \; w \rightarrow \lessdot(t_{vs} = Int), (v, Leaf \; w)\gtrdot \\ : Tree \rightarrow Int \rightarrow \lessdot(t_{vs} = Int), (t_{vs}, Tree)\gtrdot\end{array}} \; \text{ABS}(^{*}2)}{\begin{array}{c}\Gamma \vdash^{e} \bar{\exists} \; t_{vs} \; . \; \lambda(Leaf \; v) \; w \rightarrow \lessdot(t_{vs} = Int), (v, Leaf \; w)\gtrdot \\ : \bar{\exists} \; t_{vs} \; . \; Tree \rightarrow t_{vs} \rightarrow (t_{vs}, Tree)\end{array}} \; \text{E.I}$$

If we read the derivation bottom-up, by applying the introduction rule E.I we derive the type of the lambda abstraction to be $Tree \rightarrow Int \rightarrow \lessdot(t_{vs} = Int), (t_{vs}, Tree)\gtrdot$. Then, with the application of the usual rule for abstraction (ABS) twice we determine that the type of $\lessdot(t_{vs} = Int), (v, Leaf \; w)\gtrdot$ is $\lessdot(t_{vs} = Int), (t_{vs}, Tree)\gtrdot$ given that v and w are bound to *Int* in the environment. Applying PACK we conclude that this type is correct given that the pair $(v, Leaf \; w)$ has type $(Int, Tree)$.

In the second branch, the type of the lambda abstraction is $Tree \rightarrow (t_{vsl}, t_{vsr}) \rightarrow \lessdot(t_{vs} = (t_{vsl}, t_{vsr})), (t_{vs}, Tree)\gtrdot$, with t_{vsl} and t_{vsr} coming from the recursive calls of *idTree′*. We show how the rules are applied in the derivations corresponding to the first binding of the **let** expression.

The derivation for the pattern:

$$\dfrac{\dfrac{}{vsl \mapsto t_{vsl}, tl \mapsto Tree, t_{vsl} \vdash^{p} (vsl, tl) : (t_{vsl}, Tree)} \; \text{VARPAT}(^{*}2)}{\begin{array}{c}vsl \mapsto t_{vsl}, tl \mapsto Tree, t_{vsl} \vdash^{p} \lessdot t_{vsl}, (vsl, tl) \gtrdot : \\ \lessdot(t_{vs} = t_{vsl}), (t_{vs}, Tree)\gtrdot\end{array}} \; \text{UNPACKPAT}$$

If we read the pattern judgment $\Lambda \vdash^{p} p : \sigma$ as an output Λ produced out of the inputs p and σ, we can see how the existential type is unpacked by giving the name t_{vsl}, and the pair pattern introduces the bindings $vsl \mapsto t_{vsl}$ and $tl \mapsto Tree$.

The derivation for the right hand side, assume Γ includes $idTree' \mapsto \bar{\exists} \; t_{vs} \; . \; Tree \rightarrow t_{vs} \rightarrow (t_{vs}, Tree), l \mapsto Tree, vsl' \mapsto t_{vsl}, vst \mapsto t_{vsl}, lt \mapsto Tree, t_{vsl}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash^{e} idTree' : \bar{\exists} \; t_{vs} \; . \; Tree \rightarrow t_{vs} \rightarrow (t_{vs}, Tree)} \; \text{VAR}}{\begin{array}{c}\Gamma \vdash^{e} idTree' \; [t_{vsl}]: \\ Tree \rightarrow t_{vsl} \rightarrow \lessdot(t_{vs} = t_{vsl}), (t_{vs}, Tree)\gtrdot\end{array}} \; \text{E.E}}{\Gamma \vdash^{e} idTree' \; [t_{vsl}] \; l : t_{vsl} \rightarrow \lessdot(t_{vs} = t_{vsl}), (t_{vs}, Tree)\gtrdot} \; \text{APP-VAR}}{\Gamma \vdash^{e} idTree' \; [t_{vsl}] \; l \; vsl' : \lessdot(t_{vs} = t_{vsl}), (t_{vs}, Tree)\gtrdot} \; \text{APP-VAR}$$

In this case we apply the usual rules for APP and VAR (which we applied together under the name APP-VAR) twice. Then, by aplying the elimination E.E, we conclude that *idTree′* has type $\bar{\exists} \; t_{vs} \; . \; Tree \rightarrow t_{vs} \rightarrow (t_{vs}, Tree)$, which can be unified to the type we already have in the context.

3.3 Safety

We want to stress the differences between our $\bar{\exists}$ and the conventional $\exists$. Suppose we want to define the following function:

$$\begin{array}{l} f :: \exists \; x \; . \; x \rightarrow Bool \rightarrow (x, Int) \\ f = \lambda v \; True \rightarrow (v + 1, v) \\ \quad [\!] \; \lambda c \; False \rightarrow (chr \; (ord \; c + 1), ord \; c) \end{array}$$

This function does not compile using $\exists$, while it is valid if we use $\bar{\exists}$. In case such a function was accepted using the conventional existentials, then the following code would be wrongly accepted too:

```
newSTRef   ::                 a → ST s (STRef s a)
writeSTRef :: STRef s a → a → ST s ()
readSTRef  :: STRef s a     → ST s a
runST :: (∀ s . ST s a) → a
```

Figure 15: Types of the ST operations

```
let ⋖ t_c, g⋗ = f    -- unpacking f
    (i, v1) = g c True
    (c, v2) = g i False
in ...
```

Here we instantiate the existential type of f with a type constant and then bind the resulting value to a g. Both calls of g are now assuming that the same type constant is used for the existential value in the type of f, causing type checking to succeed where its should not. With our $\bar{\exists}$ this cannot happen since we cannot unpack f but only a result. Suppose f has type $\bar{\exists}\ x\ .\ x \to Bool \to (x, Int)$, if we want to unpack it we have to start by eliminating the $\bar{\exists}$ by doing $f\ [t_c]$. This expression has type $t_c \to Bool \to \lessdot(\alpha = t_c), (\alpha, Int)\gtrdot$ and thus it cannot be used as right hand side of a let binding.

4 THE *ST* MONAD

In our last example we will show how we can put the fact that we have made it possible to have lazy unpacking to further good use. We will use the type stemming from the unpacking match to parameterise a polymorphic function, which returns that existentially typed value!

The state monad *ST* [9] is an important Haskell data type, and a de-facto required part of any Haskell infrastructure. Although there are good reasons for supporting this data type at a very low level, and for providing it with extensive runtime support, the question arises whether we can implement the data type in Haskell itself.

The *ST*-monad represents a *stateful computation*; i.e. a computation that takes a state and transforms it into another state. Such transformations include extending the state by introducing a new variable, and writing and reading already introduced variables. The code in Figure 15 introduces the types of the corresponding basic operations *newSTRef*, *writeSTRef* and *readSTRef*.

The function *runST* creates a new empty state, runs the computation starting with this new state, discards the final state when done and *return*s the value resulting from the computation. Its type $(\forall\ a\ .\ ST\ s\ a) \to a$ is interesting by itself because of its higher order type: it takes a monad *ST s a* that is polymorphic in *s*. One might think of this as giving the function *runST* the right to choose a unique label for a new ‘named heap’ by instantiating the type *s*, the choice of which is kept hidden from the rest of the program. This label is then used to label the handed out references of type *STRef s a*. Since there is no way to get hold of this unique label *s* in the rest of the program, this guarantees that all *STRef*’s created by this call of *runST* indeed point into the ‘heap’ for which they were handed out.

To show how to construct a state and how to read from it and write to it we give a small *example* in Figure 16; two variables *r1* and *r2* are introduced and initialized. In the *True* branch of the

```
import Control.Monad.ST
import Data.STRef
example = do r1 ← newSTRef 2
             v1 ← readSTRef r1
             r2 ← newSTRef (show v1)
             modifySTRef r1 (+3)
             v2 ← readSTRef r2
             if v2 ≡ "2" then do v1 ← readSTRef r1
                                 r3 ← newSTRef v1
                                 v3 ← readSTRef r3
                                 return (v2, v1)
                         else return ("false", 7)
demo = runST example
```

Figure 16: An example of the *ST* monad

conditional expression we introduce another new variable. The value of *demo* evaluates to ("2", 5). This demonstrates that in general we will not be able to easily determine from the program text how many variables will be created when running the code; we actually have to mimic the execution. Note that the type of value held by each variable is fixed upon creating the variable, but not all variables hold a value of the same type.

We present our implementation of the functions introduced above in two steps. In Section 4.1 we show how a state may be extended with new variables and how it is made accessible. We will however not be able yet to access the variables. In Section 4.2 we extend the code and show how we can create references pointing into the state, and how to use these to read from and write to variables.

4.1 Constructing a State

4.1.1 The type ST. As we have seen in our example our state can accomodate many values of different types, so one of the first types which comes into mind to use for storing all these values is a nested Cartesian product. Indeed it is easy to do so if we know beforehand precisely which kind of values have to be stored and how many. If however the state evolves as a result of running the computation, and may even depend on values of variables introduced earlier this approach fails. Furthermore one of the distinguishing features of the *ST* monad is that the handed out references are labeled with a type, which uniquely identifies the state they point into; this is a property we definitely want to maintain since it is an important safety guarantee and we do not want to have dangling references into a state which has been garbage collected.

In order to understand our solution it is helpful to try to forget about a specific evaluation order (as a reader might be naturally inclined to do), and to move to a *data-flow view* of our computations.

The box in Figure 17 represents a piece of stateful computation, i.e. a value of type *ST s a*, which is –for the time being– a function type which:

- takes two arguments of type *s* and *rest*, represented by incoming arrows

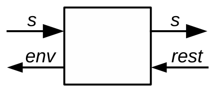

Figure 17: *ST* **type**

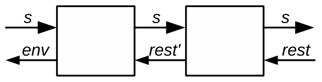

Figure 18: Bind

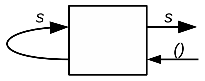

Figure 19: Run

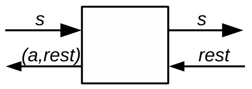

Figure 20: *insert a*

- returns a result of type s and a result of type env, represented by outgoing arrows. These two results will be represented by a tuple in our final code.

The Haskell type[3] that corresponds to such a box is:

$$\textbf{data}\ ST\ s\ a = ST\ (\forall\ t_{rest}\ .\ t_{rest} \rightarrow s \rightarrow (a, \exists\ t_{env}\ .\ t_{env}, s))$$

The rôle of the various arrows is as follows:

- The input arrow *rest* represents the right-nested Cartesian product of all the values that are possibly introduced by *succeeding computations*. Since our computation is indifferent to this value its type is polymorphically quantified over this *rest* type. Keep in mind that we deal with a language which has lazy evaluation, so by the time this 'value' is passed it will not have been evaluated yet.
- The result of type t_{env} again is a right-nested Cartesian product, which has a value of the incoming type *rest* as its tail. This prefix to this tail corresponds to fields holding to the new variables introduced by this *ST* value; hence it is a (possibly) extended version of *rest*. Since only the internals of our computation know how many variables and of which types are added by this computation, we use an existential result type. Although some tail of the nested product type t_{env} will be of the type t_{rest} this fact is not explicitly represented in the *ST* type.
- The remaining input and output arrows both carry a value of some type s, which is the type of the final state of the overall computation as ran by *runST*, eventually containing all created variables. This value is threaded though the computation, while the variables contained in it are being read and written. Since this value has the shape of the final state this shape (and thus its type) remains unchanged. Whereas in the *ST* monad as built-in into GHC the type of the state merely serves as a label indicating which 'heap' we are dealing with, here it is the actual heap in the form of the nested Cartesian product that is being passed on.
- In our pictures we have not shown the type a, since it does not play a role in our explanation of how the state is constructed and represented.

Stateful computations can be composed into larger computations by connecting their arrows as shown in 18; boxes to the left represent earlier computations and boxes to the right succeeding computations.

4.1.2 runST. Before looking at the code of *runST* we take a look at Figure 19. When running the computation with the function *runST*, the initial empty state () is passed as t_{rest} argument at the very end of the computation. It emerges as the lazily constructed value of some type t_{env} at the far left of our composed boxes, containing all the variables (to be) introduced. Now we decide upon the type s and choose it to be the same as the returned type t_{env}, and feed the value of type t_{env} back into the computation as the s parameter.

In the code below we have given two definitions of runST: the first one unannotated, and the second one containing explicit type annotations:

```
runST :: (∀ s . ST s a) → a
runST (ST st) = let (a, env, _) = st () s
                    s = env
                in  a

runST :: (∀ s . ST s a) → a
runST st = let (ST st′) = st [t_env]
               (a, <t_env, env>, _) = st′ [()] () env
           in  a
```

Focusing on the second definition we see that we instantiate the polymorphic type of the value *st* with two types: the first type parameter is the type t_{env} returned as the existential type by this computation describing the lazily constructed complete state, and the second is the type of the empty state (), which we feed in from the far right end of our composed sequence of boxes. We have omitted the annotations for the type a, since they play no special rôle here. Hence we do not only have a **letrec** at the value-level, but also at the type level: the type t_{env} which is 'returned' as the type of the existential *env*-part of the result is used as a type parameter in a right hand side expression of the binding group!

Since the type of the final state s is universally quantified when the computation is run, the user of the *ST* monad cannot assume anything about it. Although we can easily add operators like *get* and *put* to read the *ST* state and to put it back we canot make any other use of it; in particular we do not have access to the individual elements making up the state.

```
get :: ST s s
get = ST (λenv s → (s, env, s))

put :: s → ST s ()
put s = ST (λenv _ → ((), env, s′))
```

4.1.3 Monad. Computations are monads and can thus be composed with the monadic bind (≫=) operator. In Figure 18 we show how the state is constructed from right to left and how this final state is modified in a left-to-right traversal of the individual steps of the computation.

```
instance Monad (ST s) where
  return = pure
  (ST st_a) ≫= f
```

[3]This is the type as accepted by UHC. For GHC is is necessary to introduce an extra constructor when introducing the type *env*.

```
    = ST (λrest s → let (a, env, s') = st_a rest' s)
                        (ST st_b)     = f   a
                        (b,  rest', s'') = st_b rest s'
                     in (b,  env,  s'')
```

Again this binder cannot be implemented with System-F existential types, because we pass *rest′* returned by the second computation (st_b) to st_a, whereas the state of type *s* is passed in the other direction.

4.1.4 Functor and Applicative. In the last version of the Haskell libraries it is required that *Monad* instances are also *Functor* and *Applicative* instances, amongst others to support the applicative **do**-notation. So we define:

```
instance Functor (ST s) where
fmap f (ST st)
    = ST (λrest s → let  (a, rest', s') = st rest s
                     in  (f a, rest', s')
         )
instance Applicative (ST s) where
  pure a = ST (λrest s → (a, rest, s))
  (ST st_b2a) <*> ~(ST st_b)
    = ST (λrest s → let  (b2a,   rest'', s')  = st_b2a rest' s
                         (b,     rest',  s'') = st_b   rest  s'
                     in  (b2a b, rest'', s'')
         )
```

Note that we have used an irrefutable pattern for the right hand side parameter of <*>; the evaluation of the right hand side should not be pushed any further than strictly necessary. It is important to note how the final state *s* is passed in a forward direction, whereas the future additions in *rest* are passed backwards through the computation; again we are tying a knot here.

4.2 *newSTRef*

4.2.1 insert. As a first step in showing how a state is constructed, we define the function *insert*, which takes a value of some type *a* and extends the state with this value like a *newSTRef*, but it returns the newly stored value instead of a reference to it. Again we have given the code with and without type annotations.

```
insert :: a → ST s a
insert a = ST (λrest → s → (a, (a, rest), s))

insert = Λ t_a → a → ST (Λ t_s t_rest
         → λ(rest :: t_rest) (s :: t_s) → ( a :: t_a
                                          , ≺ (t_a, t_rest), (a, rest) ≻
                                          , s :: t_s
                                          ))
```

Figure 20 shows its effect on the state being constructed. Notice that the *rest* of the state needs no inspection in order to be able to extend it. Thus, due to lazy evaluation, even infinite states can be constructed. Note furthermore that the function *insert* should definitely not be made strict. The *rest* argument is likely to depend on values read from a state of which the constructed (*a*, *rest*) pair is a trailing component, and which is passed to it as the *s* parameter!

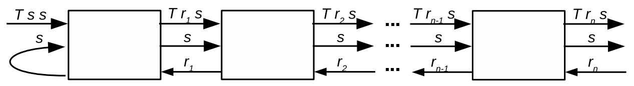

Figure 21: *ST* type with references

In the rest of this section we are going to extend our definitions such that we can create and use references into the constructed state.

4.2.2 STRef s a. Typed references over a nested Cartesian product are represented by the *STRef s a* GADT [2, 13], indexed by the type *s* of the Cartesian product representing our complete state and *a* being the type of the value referred to. The constructor *RZ* refers to the first element of the nested product, whereas *RS* constructs the successor of an index *STRef r a* in a product of type *r*. The type *b* in this case is thus the first element which has to be skipped when indexing.

```
data STRef s a where
  RZ :: STRef (a, b) a
  RS :: STRef r a → STRef (b, r) a
```

With such references, look-ups and modifications can be performed safely. The type system makes sure that no pointers can point outside of the structure and that the type pointed to is the type we expect:

```
rlookup :: STRef s a → s → a
rlookup  RZ      (a, _) = a
rlookup  (RS r) (_, b) = rlookup r b
rmodify :: (a → a) → STRef s a → s → s
rmodify f RZ      (a, r) = (f a, r)
rmodify f (RS r) (a, b) = (a, rmodify f r b)
```

4.2.3 Transforming STRef's. When adding a new location to the state, with the function *newSTRef*, the reference to its position in the final state is to be returned.

```
newSTRef :: a → ST s (STRef s a)
```

If we look at the *ST* type defined in the previous subsection (and represented in Figure 17), there is no way to relate the state *env* constructed by this computation and the final state. Thus, we have to extend the *ST* type so this relation becomes available.

For this purpose we define a type $T\ s_1\ s_2$ which represents a transformation of a reference into a structure s_1 to one in s_2.

```
newtype T s1 s2
   = T { unT :: ∀ a . STRef s1 a → STRef s2 a}
```

4.2.4 Our extended version of ST. We now change the type *ST* such that transformations from references in the *rest* state to references in the final state are passed on from left to right, together with the state of type *s*. One may think of it as a counter keeping track of how many elements have thus far been added to the state. The counter is indexed with the type which remains of the final state provided the counted number of elements have been removed. Note that because this type is indexed by the type of *env* it is actually determined by the passed value *rest* together with the number and types of the new variables added by this computation.

```
data ST s a
  = ST (     ∀ t_rest . t_rest → s
         → ∃̄ t_env . T t_env s → (a, t_env, s, T t_rest s))
```

Notice that we are using our new quantifier $\bar{\exists}$, which is to be interpreted as follows: as before we think of the type *env* as being determined by the call to the function and made available as part of the result. Since we want to make use of this type in specifying the type of one of the parameters we have somehow to extended the scope 'forward'.

The instances given have now to be extended to pass these transformations through the constituting computations. We do so for the *Monad* instance. The others follow trivially.

```
instance Monad (ST s) where
  return a = ST (λenv s tr_env↦s → (a, env, s, tr_env↦s))
  (ST st_a) ≫= f
    = ST $ λrest s tr_env↦s
      → let (a,  env,  s′,  tr_rest′↦s) = st_a rest′ s  tr_env↦s
            (ST st_b)                   = f a
            (b,  rest′, s′′, tr_rest↦s) = st_b rest  s′ tr_rest′↦s
        in  (b,  env,  s′′, tr_rest↦s)
```

In Figure 21 we show the types involved when running a composition of computations. For the leftmost computation, the transformation is just the identity function, since this is the type of the final state (no more locations will be added). Note that because we have again chosen the returned *env* to be the value to pass on as the initial state *s* these have the same type, and thus $T\ id :: T\ t_{env}\ tenv$ has the correct type.

```
runST :: (∀ s . ST s a) → a
runST (ST st) =
  let (a, ≺t_env, env≻, _, _) = st () env (T id) in a
```

When adding a new location, the reference to this new location in the final state is obtained by applying the transformation $tr_{env \mapsto s}$ (with type $T\ env\ s$) to a reference to the first position in the just extended state. The transformation of references for the succeeding computations is obtained by composing current transformation with *RS*; i.e. references in the rest of the state point to locations in the second component of the pair (*a*, *rest*).

```
newSTRef :: a → ST s (STRef s a)
newSTRef a = ST $ λrest s tr_env↦s
                  → ((unT tr_env↦s) RZ
                    , (a, rest)
                    , s
                    , T (unT tr_env↦s . RS)
                    )
```

Having a reference to an element in the final state, to obtain the referred value is to perform an *rlookup* in the state *s* (traveling from left-to-right).

```
readSTRef :: STRef s a → ST s a
readSTRef r
  = ST $ λenv s tr_env↦s → (rlookup r s, env, s, tr_env↦s)
```

Similarly, we can overwrite or modify stored values.

```
writeSTRef :: STRef s a → a → ST s ()
writeSTRef r a
  = ST $ (λenv s tr_env↦s → (()
                            , env
                            , rmodify (const a) r s
                            , tr_env↦s
                            ))
modifySTRef :: STRef s a → (a → a) → ST s ()
modifySTRef r f
  = ST $ λenv s tr_env↦s → ((), env, rmodify f r s, tr_env↦s)
```

A nice property of our representation of pointers is that they may be compared, and if two pointers are found to be equal then the GADT-based type system returns a proof that the type of the value pointed at is the same.

5 FINE TUNING

5.1 Do we need $\exists$ and $\bar{\exists}$?

When we compare the type rules for $\exists$ and $\bar{\exists}$ we see that they only differ for function types. One may argue that the case for having a normal $\exists$ does not make much sense for a function type; so when we decide to treat the $\exists$ for a function as an $\bar{\exists}$ there is no need for an extra symbol.

We do not loose expressivity, since if we really want to have the classical existentially quantified function, of which the unpacked version can be called at multiple places we can easily tuple it with a dummy value into an existential pair:

```
d_f :: ∃ x . (x, x → Bool → (x, Int) = (3, λv _ → (v + 1, v))
let ≺ t, (_, g) ≻ = d_f    -- unpacking
    (i1, v1) = g i2 True
    (i2, v2) = g i1 False
in v1 ...
```

Note that in d_f the choice for type x can no longer depend on the passed *Bool* argument.

5.2 Extending the class system

An indispensable component of the Haskell type system is its class system, which makes it possible to pass extra information about a polymorphic value to a function. In a similar way we may want to provide extra information to the calling context. Currently Haskell only allows constructors to be constrained by classes if these classes refer to existential types. We propose to generalise this: a constraint on a constructor just packs an extra dictionary in the record, and pattern matching on such a constructor brings the class instance in scope. With this extension we can rewrite our solution for the sorting tree to the code given in Figure 22.

Note how, just as in the case with polymorphic functions, the class instances will be automatically constructed, passed around and accessed.

```
class Insertable cl where
  insert :: Int → cl → (Int, cl)
instance Insertable cl ⇒ Insertable (Int, cl) where
  insert w (x, xs) = if  w < x then (w, (x, xs))
                                else  (x, insert w xs))
instance Insertable () where
  insert w ()      = (w, ())
data OrdList cl where
  OrdList :: Insertable cl ⇒ cl → OrdList cl
sortTree″ (Leaf v) (OrdList rest)~(x, xs)
   = (OrdList (insert v rest), xs, Leaf x)
sortTree″ (Bin l r) rest xs
   = let (vl, xsl, tl)      = sortTree″ l vr xs
         (vr, xsr, tr)      = sortTree″ r rest xsl
     in (vl, xsr, Bin tl tr)
```

Figure 22: Using classes

As a final extension we show how the new extension comes in handy in the case of the use of guards. Suppose we want to change our tree sorting algorithm such that we sort the sublists containing the even and the odd leaf values separately. This can be done by duplicating the parameters:

```
sortTree″ (Leaf v) ((OrdList e, o) ~((x, xs), ys)
  | even v = ((OrdList (insert v e), o), (xs, ys), Leaf x)
sortTree″ (Leaf v) ((e, Ordlist o) ~((xs, (y, ys))
  | odd  v = ((e, OrdList (insert v o)), (xs, ys), Leaf x)
sortTree″ (Bin l r) rest xs =
  let (vl, xsl, tl)  = sortTree″ l vr xs
      (vr, xsr, tr) = sortTree″ r rest xsl
  in (vl, xsr, Bin tl tr)
```

Note that again we can provide all parameters at once, and use guards. We could have written this code in GHC style, computing the types of the pair of Cartesian products by first inspecting not only the shape of the tree but also the values stored in the tree. We think however that our approach, in which we see the computed existential type as part of the result, is a more natural one given that we are dealing with a language which has lazy evaluation.

6 FUTURE WORK

With respect to the implementation of the *ST* monad, one may want to remark that the presented implementation is very inefficient, since access to components of the state is done in linear time. It is however possible to lazily convert the Cartesian product into a tree-like structure, which gives us logarithmic lookup time. From this tree we may compute a list of indices in the tree, which can then be passed on from left-to-right through the computation. Whenever we add an element to the state, we take the first index from this list, since it is guaranteed to point to the position in the tree-shaped state where the element currently being added will end up.

There is still work to be done to mechanically verify the soundness of our type rules, in the sense that "no well-formed program can go wrong". It is clear from our description that the type rules will depend on the user providing sufficient type annotations, since the standard HM inference system is not able to infer neither the ∃ nor the $\bar{\exists}$ quantifiers.

7 DISCUSSION

We have completed our description. We think however that it is desirable to spend some attention to why we managed to implement our code, whereas it is not accepted by GHC. There are several reasons.

In the first place the Utrecht Haskell Compiler allows to specify an existential type without the introduction of an extra intervening data type. This makes the code more concise, but is not essential. In GHC we could have defined our *ST* type by introducing an extra type *ST′*:

```
data ST s a
   =            ST ∀ . t_rest → s → ST′ s t_rest a
data ST′ s t_rest a
   = ∀ t_env . ST′ (T t_env s) → (a, t_env, T t_rest s)
```

A more serious problem is that pattern matching for existential types in GHC is strict, and we thus cannot unpack [10] such a value in the right hand side of a **let**. This restriction (probably) finds its roots in the fact that existential types naturally come with GADT's and that in the current GHC implementation non-strict pattern matching for GADT's may lead to unsafe code [16]. There are two solutions for this. If the GADT does not introduce equality constraints, as is the case in our code, the restriction could be relieved. Another solution is to represent the equality constraints implicitly in the generated code. This corresponds to the approach taken in the pre-cursor of GADTs by Baars and Swierstra [1], where equality constraints are represented as coercions which are called when the proof that two types are equal is needed; failing to return such a proof leads to non-termination. The current GHC approach is thus more strict in requiring that it can be statically determined that a proof exists, and thus does not have to be checked for dynamically. This restriction makes it impossible to pass the value back into the computation as demonstrated by a 'rewrite' of *runST* where the passed parameter *s* has become unbound. Pairing *s* with the result *a* in attempt to get hold of it, so we can feed it back, does not work either since the existential type in that case 'escapes' [16].

```
runST (ST st) = case st () s (T id) of
                   (a, s, _, _) → a
```

Note that the latter problem could be solved by using a more fine-grained description of the use of existentials [11].

But in all these cases the problem remains that conventional System-F does not allow for a **letrec** construct at the type level: we cannot use a type which we have gotten access to by unpacking it, as a type parameter to a polymorphic function the call of which produced that very type.

8 CONCLUSION

We have shown how to widen the use of existential types, such that besides polymorphic functions we can also have polymorphic contexts. As the main example of the usefulness of this approach we have given an alternative implementation of the *ST* monad. A similar version was developed in [2]. In that implementation a state was constructed in a left-to-right fashion, with the constructed state coming out at the right. This state had to be fed back in order to be able to run the state. Unfortunately there the last added elements end up at the beginning of the Cartesian product, so handed out references had to be updated. This makes a monadic interface impossible, and instead an arrow-based interface was given. We believe the implementation given here is the one to be preferred, because of its more expressive interface. In the original implementation of the TTTAS (Typed Transformations of Typed Abstract Syntax)[2, 3] library we had to make provisions for maintaining so-called meta information during the transformation process. With the monadic interface this extra provision is no longer needed, and thus the library can be simplified considerably.

Although the *idTree* example may seem to be quite artificial we have encountered the pattern used there quite often. When programming in an attribute-grammar based style (writing so-called *circular* functions) one gets very accustomed to constructing values from trees, which are later fed back into the computation. Although many of these applications could be rewritten into multi-visit functions, this implies the explicit construction of intermediate representations and makes resulting programs much more difficult to develop and maintain.

Another example where a constructed value is passed back can be found in the implementation of a pretty printer which has a bounded look-ahead [15]. In this algorithm two processes walk over a tree-like structure which we want to layout in a nice way. These processes communicate with each other through streams, which we represented as lists. One process produces a list of questions to be answered by the second process, which communicates back these answers through another list. The latter process produces a list which is threaded backwards through the tree and which is, when it emerges at the top, passed back into the tree and then passed on in a left-to-right tree traversal. In the attribute grammar based implementation [14] it is not enforced that the first process adds exactly one element to the list of questions for each node of interest, nor that the second process produces exactly one answer for each question, and that the first process consumes exactly one answer for each question asked. Using the techniques described in this paper we may enforce these requirements.

We finally want to remark that the code we have written is by no means special once one gets used to the 'data-flow' view of lazy functional programming. Especially when trying to translate the results of an attribute grammar based development –which are of a data flow view by nature– into Haskell it is that one runs into the kind of problems we have addressed; information flowing backwards and forwards is likely to occur in such developments and we argue that a type system should not make it impossible to express this in a type-safe way.

9 ACKNOWLEDGMENTS

We want to thank Andres Löh, Tom Schijvers and Andrea Vezzosi for discussing System-F related issues with us.

REFERENCES

[1] Baars, A.I., Swierstra, S.D.: Typing dynamic typing. In: Peyton Jones, S. (ed.) Proceedingsof the seventh ACM SIGPLAN international conference on Functional programming. pp. 157–166. ACM Press (2002)

[2] Baars, A.I., Swierstra, S.D., Viera, M.: Typed Transformations of Typed Abstract Syntax. In: TLDI '09: fourth ACM SIGPLAN Workshop on Types in Language Design and Implementation. pp. 15–26. ACM, New York, NY, USA (2009)

[3] Baars, A.I., Swierstra, S.D., Viera, M.: Typed Transformations of Typed Grammars: The Left Corner Transform. In: Proceedingsof the 9th Workshop on Language Descriptions Tools and Applications. pp. 18–33. ENTCS (2009)

[4] van Binsbergen, L.T., Bransen, J., Dijkstra, A.: Linearly ordered attribute grammars: With automatic augmenting dependency selection. In: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation. pp. 49–60. PEPM '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2678015.2682543

[5] Bird, R.S.: Using Circular Programs to Eliminate Multiple Traversals of Data. Acta Informatica 21, 239–250 (1984)

[6] Engelfriet, J., Filé, G.: Simple multi-visit attribute grammars. Journal of Computer and System Sciences 24(3), 283 – 314 (1982), http://www.sciencedirect.com/science/article/\pii/0022000082900307

[7] Harper, P.R.: Practical Foundations for Programming Languages. Cambridge University Press, New York, NY, USA (2012)

[8] Kastens, U.: Ordered Attribute Grammars. Acta Informatica 13, 229–256 (1980)

[9] Launchbury, J., Peyton Jones, S.L.: Lazy functional state threads. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation. pp. 24–35. PLDI '94, ACM, New York, NY, USA (1994), http://doi.acm.org/10.1145/178243.178246

[10] Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. ACM Trans. Program. Lang. Syst. 10(3), 470–502 (Jul 1988), http://doi.acm.org/10.1145/44501.45065

[11] Montagu, B., Rémy, D.: Modeling abstract types in modules with open existential types. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 354–365. POPL '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1480881.1480926

[12] Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)

[13] Sheard, T., Pasalic, E.: Meta-programming with built-in type equality. Electron. Notes Theor. Comput. Sci. 199, 49–65 (Feb 2008), http://dx.doi.org/10.1016/j.entcs.2007.11.012

[14] Swierstra, S.D.: Linear, online, functional pretty printing (extended and corrected version). Tech. Rep. UU-CS-2004-025a, Inst. of Information and Comp. Science, Utrecht Univ. (2004)

[15] Swierstra, S.D., Chitil, O.: Linear, bounded, functional pretty-printing. Journal of Functional Programming 19(01), 1–16 (2009)

[16] Vytiniotis, D., Peyton Jones, S., Schrijvers, T., Sulzmann, M.: Outsidein(x) modular type inference with local assumptions. Journal of Functional Programming 21, 333–412 (2011), http://journals.cambridge.org/article_S0956796811000098