Cycles without pollution: a gradual typing poem

Sam Tobin-Hochstadt[1] and Robert Bruce Findler[2]

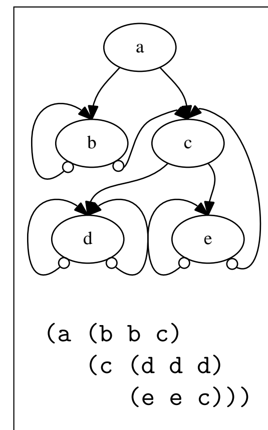
[1] Northeastern University
[2] Northwestern University

Abstract. It is fairly intuitive that adding dynamic typing to a statically typed language can add some convenience to a developer, at the price of weakening the guarantees of the type system. Surprisingly, the reverse is also true. This paper presents a programming poem à la Dan Friedman illustrating that temporarily circumventing the type system via a gradual typing cast can actually *strengthen* the guarantees of the type system overall.

The paper first presents a challenge to write a small but interesting program. It proceeds by exploring how it could be written in Haskell and in SML (showing and discussing runnable code). It then gives the Typed Scheme version in a slower, more tutorial style, since the reader is assumed to be more familiar with Haskell and SML than Typed Scheme. Along the way, the paper shows how the Typed Scheme version overcomes weaknesses in the earlier versions. The key insight is that gradual typing enables us to view a data structure with two different types, one during the creation and one during use. The type during creation allows more operations that are needed to build an element of the data structure, and the type during use exposes more invariants of the data structure that can then be used to guarantee properties of the clients (termination, in this case).

1 The challenge

Write a function that accepts the specification of an infinite regular tree as a series of equations (à la Courcelle [1]) and turn it into a direct representation of the regular tree. Specifications consist of either a triple that names an interior node and a pair of specifications for the left and right children, or a label that refers to an earlier node. For example, the box to the right contains a tree and its matching specification. The regular tree is shown with backwards arrows beginning with open circles to indicate the repeated structure. Node labels must not be duplicated in a specification, and each label must refer to some node.



Also, *design a function that, given one of these trees, determines how many steps there are along the shortest non-trivial path through the tree to a subtree that is isomorphic to the given tree, if any*. For example, the `b` subtree is one step away from an isomorphic tree, and there is no tree isomorphic to the tree starting at `a`. Since node labels are not duplicated in the specification, checking label equality is sufficent to detect isomorphism. The second function must always terminate.

2 A (nearly) perfect Haskell program

Writing this program in Haskell is natural. We represent tree specifications using the type STree and trees using the type ITree.

```
data STree = STree String STree STree | Link String
data ITree = ITree String ITree ITree
```

The function link converts tree specifications into regular trees and uses two mutually recursive helper functions, conv and find. The first just processes the tree, turning STree constructors into ITree constructors, until it hits a link label, at which point it calls find, which searches in the original tree for the corresponding node, and then returns it (after using conv to convert it, naturally).

```
link :: STree -> ITree
link main = conv main
  where conv :: STree -> ITree
        conv (STree str tl tr) =
             ITree str (conv tl) (conv tr)
        conv (Link str) =
             find main str []

        find :: STree -> String -> [STree] -> ITree
        find (STree str2 tl tr) str pending
           | str2==str = conv (STree str2 tl tr)
           | otherwise = find tl str (tr:pending)
        find (Link str1) str2 (p:ps) = find p str2 ps
```

The period function uses breath-first search to find the nearest occurrence of the same name in the tree, if one exists.

```
period :: ITree -> Maybe Int
period (ITree str tl tr) = bfs [(tl,1),(tr,1)] []
  where bfs :: [(ITree,Int)] -> [String] -> Maybe Int
        bfs [] visited = Nothing
        bfs ((ITree str2 tl tr,i) : rest) visited
         | str2==str = Just i
         | elem str2 visited = bfs rest visited
         | otherwise = bfs (rest ++ [(tl,i+1),(tr,i+1)])
                           (str2:visited)
```

There are two flaws in this program. First, the type system does not reflect the constraint that the specification of the tree is well-formed (i.e., labels on nodes are unique and references to nodes are all defined). Second, the period function does not always terminate. Indeed, it may not even terminate on a tree returned by link, since link may be given an infinite (non-repeating) specification tree. This latter issue is simply because we are programming in Haskell, and seems unavoidable there. So, let us turn to SML where the specification trees cannot be infinitely large.

3 An SML port of the Haskell version

One fairly simple way to port this program to a strict language is to add a few thunks to judiciously delay evaluation. This section explores how to do that for SML, using these data types for the specification and the trees.

```
datatype stree=STree of string * stree * stree | Link of string
datatype itree=ITree of string * (unit->itree) * (unit->itree)
```

The code that builds the tree looks very similar to the Haskell variant in the previous section, but with a few thunks and applications in the appropriate places.

```
(* link : stree -> itree *)
fun link main = let
  fun conv (STree (str,tl,tr)) =
      ITree (str,fn () => conv tl,fn () => conv tr)
    | conv (Link str) = find main str []
  and find (STree (str2,tl,tr)) str pending =
      if (str2=str)
      then conv (STree (str2,tl,tr))
      else find tl str (tr::pending)
   | find (Link str1) str2 (p::ps) = find p str2 ps
in conv main end
```

The `period` function is also very similar to its Haskell counterpart, but with a few extra thunk applications.

```
(* period : ITree -> int option *)
fun period (ITree (str,tl,tr)) = let
  fun elem n l = exists (fn x => (n = x)) l
  fun bfs [] visited = NONE
    | bfs ((ITree (str2,tl,tr),i) :: rest) visited =
      if (str2=str) then SOME i else
      if (elem str2 visited) then bfs rest visited
      else bfs (rest @ [(tl(),i+1),(tr(),i+1)]) (str2::visited)
in bfs [(tl(),1),(tr(),1)] [] end
```

This version improves on the Haskell version because the `period` function terminates for all `itrees` produced by `link`. But it does not terminate for all values with type `itree`, since there are members of `itree` that are completely misbehaved. For example, the labels on `change_up`'s children change over time.

```
val change_up = let
  val r = ref 0
  fun f () = (r := !r+1; ITree (toString (!r),f,f))
in ITree("a",f,f) end
```

As this example also shows, using an abstract type to represent `itree` does not suffice, since the client would have no knowledge of how the type is implemented, always leaving the possibility of trees like `change_up`. With that in mind, we turn to another possibile concrete type.

4 Another imperfect SML version

One way to ensure that the graph is just a simple datastructure, and thus does not have changing children, is to use reference cells to build a cyclic data structure, using this datatype.

```
datatype itree = ITree of string *
                 MaybeItree ref *
                 MaybeItree ref

and MaybeItree = I of itree | S of string
```

This version of `link` first sets up a `ref` cell with an empty list as a mutable `memory` and then, using the `conv` function constructs a finite tree with references that, using the `MaybeItree` type, contain either an `itree` in the case of proper children, or a `string` in the case of a `Link` in the specification. At each step, the corresponding nodes and strings are saved in the `memory`.

Once the `conv` function is finished, the tree is re-traversed, mutating each `string` reference to the correct `itree` by looking it up in the `memory`, thus creating the appropriate cycles.

```
(* link : stree -> itree *)
fun link main = let
  val memory = ref []

  fun conv (STree (str,tl,tr)) = let
      val t = I (ITree (str,ref (conv tl),ref (conv tr)))
      in memory := ((str,t) :: !memory);
         t
      end
    | conv (Link str) = S str

  fun lookup s = case (find (fn (s',_) => s=s') (!memory))
                   of SOME (_,t) => t

  fun help (ITree (str,tl,tr)) =
      ((case !tl of
            I it => help it
          | S str => tl := lookup str);
       (case !tr of
             I it => help it
           | S str => tr := lookup str))
in
    case conv main of
      I i => (help i; i)
end
```

The implementation of `period` for this version is quite similar to the earlier versions; the only differences are minor changes to reflect the different data structure.

```
(* period : ITree -> int option *)
fun period (ITree (str,ref (I tl),ref (I tr))) = let
  fun bfs [] visited = NONE
    | bfs ((ITree (str2,ref (I tl),ref (I tr)),i) :: rest)
          visited =
      if (str2=str) then SOME i else
      if (elem str2 visited) then bfs rest visited
      else bfs (rest @ [(tl,i+1),(tr,i+1)]) (str2::visited)
in
  bfs [(tl,1),(tr,1)] []
end
```

This version improves on all of the previous version in that the type system now ensures that elements of the `itree` type are all structures and thus the `period` function for this variant is guaranteed to terminate for any well-typed input.

Unfortunately, in order to use a data structure-based representation for `itree` (and thus, in order to guarantee that `period` terminates) the type system must also expose the fact that the data structure is built with `ref`s and therefore allow clients to mutate the graphs (which is arguably worse than allowing non-termination). Additionally, the representation exposes the `MaybeItree` type used in the construction of the tree, even though a finished tree will never have any children that are `S string`.[3] Because of this, there are numerous locations in both the `link` and `period` code with incomplete matches since we, the programmers, know better than the type system what is possible.

5 Gradual typing to the rescue

Using Typed Scheme [2] and contract checking, we can build a version of the code that,

- rejects ill-formed inputs to `link` with proper blame assignment,
- allows the type system to encode the knowledge that the elements of the type are simple structures (and thus help guarantee `period` terminates),
- without giving clients the ability to mutate the structure or exposing irrelevant details of the construction.

The plan is to use code like the second SML version, but improve it via contract checking and a gradual typing-style cast. Specifically, the `itree` type is much like the SML type, but the `link` function is exported with a type that refines its input to only accept well-formed graph specifications, and refines its output so that it hides the mutable nature of the `itree` type. Even better, the gradual type cast can hide the superfluous union type (the `MaybeItree` type in the SML version) from clients of `link` and `period`.

[3] In Caml, one can construct a cyclic tree directly which helps avoid the extra union, e.g.:

```
type itree = { name : string; mutable l : itree; mutable r : itree }
let rec t = { name="a"; l=t; r=t } in t end
```

5.1 Tree Construction

Type Scheme programs consist of a set of closed, first-order modules that encapsulate different parts of the program and act as the principals for blame assignment. Each module begins with a `#lang` line indicating the language of the module, in this case `#lang typed-scheme` indicates Typed Scheme. Next comes a type declaration for `maybe-itree`, saying that it can either be a `Symbol` or an `itree`.

```
(define-type-alias maybe-itree (U itree Symbol))
```

The `itree` type is introduced by a structure declaration whose fields are `name`, `left`, and `right` and have types `String`, `maybe-itree`, and `maybe-itree` respectively. The type `maybe-itree` is a union which is either a `Symbol` or an actual `itree`. This takes the place of the SML type `MaybeItree`. Finally, the `#:mutable` keyword indicates that the fields can be mutated.

```
(define-struct: itree ([name : Symbol]
                       [left : maybe-itree]
                       [right : maybe-itree])
  #:mutable)
```

In addition to introducing the type `itree` the `define-struct:` declaration introduces the constructor function `make-itree` for making new nodes, the selectors `itree-name`, `itree-left` and `itree-right` for extracting the value of the fields and `set-itree-left!` and `set-itree-right!` for changing the contents of the left and right children.

In contrast to the SML and Haskell code, there is no need for a new datatype with explicit injection and projections for the specification trees. Instead, we simply describe the type `STree` as either a `Symbol` or a list containing a `Symbol` and two `STree`s.

```
(define-type-alias STree
  (Rec ST (U Symbol (List Symbol ST ST))))
```

Next, the function `all-valid-links?` gives the precise specification for valid inputs to `link`, written in terms of `all-nodes` and `all-links` that return all of names of the nodes and all of the names that refer to other nodes, respectively.

```
(: all-valid-links? (STree -> Any))
(define (all-valid-links? s)

  (: all-nodes (STree -> (Listof Symbol)))
  (define (all-nodes s)
    (match s
      [(list s tl tr)
       (cons s
             (append
              (all-nodes tl)
              (all-nodes tr)))]
      [s '()]))
```

```
(: all-links (STree -> (Listof Symbol)))
(define (all-links s)
  (match s
    [(list s tl tr) (append (all-nodes tl) (all-nodes tr))]
    [(? symbol? s) (list s)]))

(andmap (λ: ([e : Symbol]) (memq e (all-nodes s)))
        (all-links s)))
```

The type of `link` exploits `all-valid-links?` predicate to build a refinement type [3], in this case refining `STree`, the input type for the `all-valid-links?` predicate.

```
(: link ((Refinement all-valid-links?) -> itree))
```

The implementation of `link` is very similar to the SML version earlier. The only significant change is the use of a mutable hash table instead of the `list ref` as in the SML code (simply because this is the idiomatic choice in PLT Scheme code). Hash-tables in PLT Scheme are constructed with `make-hash`, updated with `hash-set!` and accessed with `hash-ref`.

```
(define (link main)
  (: memory (HashTable Symbol itree))
  (define memory (make-hash))

  (: conv (STree -> maybe-itree))
  (define (conv s)
    (match s
      [(list str tl tr)
       (let ([t (make-itree str (conv tl) (conv tr))])
         (hash-set! memory str t) t)]
      [(? symbol? str) str]))

  (: ans maybe-itree)
  (define ans (conv main))

  (: help (itree -> Void))
  (define (help t)
    (match t
      [(struct itree (_ tl tr))
       (if (symbol? tl)
           (set-itree-left! t (hash-ref memory tl))
           (help tl))
       (if (symbol? tr)
           (set-itree-right! t (hash-ref memory tr))
           (help tr))]))
  (if (itree? ans)
      (begin (help ans) ans)
      (error 'link "link as parent")))
```

We then `provide` the identifiers from the module that we want other modules to be able to use. Notably, we do not `provide` the mutator functions for the `itree` structure.

```
(provide (except-out (all-defined-out)
                     set-itree-name!
                     set-itree-left!
                     set-itree-right!))
```

5.2 Tree Use

In a second module, also written in Typed Scheme, we begin by importing the structure type from the implementation module.

```
(require-typed-struct itree ([name : Symbol]
                             [left : itree]
                             [right : itree])
                      "itree.ss")
```

This specifies that we wish to `require` all of the relevant identifiers from the `"itree.ss"` module, with the appropriate types. The Typed Scheme implementation automatically inserts gradual typing casts to ensure that these types are upheld by both parties. Note that the specification of the types excludes the possibility that either the `left` or `right` field of an `itree` might be a `Symbol`. Therefore, code in the new module does not have to deal with this possibility, localizing the implementation details.

Next we refer to the just-imported `itree` type to require the `link` function.

```
(require/typed "itree.ss" [link (STree -> itree)])
```

This specifies that the `link` function consumes an `STree` and produces an `itree` according to the specifications described in the previous `require`.

We can now define the `period` function.

```
(: period (itree -> (U Number #f)))
(define (period it)

  (: bfs ((Listof (Pair itree Number)) (Listof Symbol)
          -> (U Number #f)))
  (define (bfs stack visited)
    (match stack
      ['() #f]
      [(cons (cons (struct itree (str2 tl tr)) i) rest)
       (cond
         [(eq? str2 (itree-name it)) i]
         [(memq str2 visited) (bfs rest visited)]
         [else (bfs (append rest
                            (list (cons tl (add1 i))
                                  (cons tr (add1 i))))
                    (cons str2 visited))])]))
  (bfs (list (cons (itree-left it) 1) (cons (itree-right it) 1))
       '()))
```

This definition never considers the case that an `itree` might have a `Symbol` as one of its children. It is otherwise a simple translation of the SML code, but one that is guaranteed not to fail at runtime.

We can now use the `link` function to construct an regular tree from our original s-expression specification.

```
(define at (link '(a (b b c)
                     (c (d d d)
                        (e e c)))))
```

Then the `period` function determines that the root of the original tree has no path back to itself, the left subtree has itself as a direct child, and the right subtree's grandchild is itself again.

```
> (list (period at)
        (period (itree-left at))
        (period (itree-right at)))
(#f 1 2)
```

If we had incorrectly implemented the `link` function, the contract checker would have caught the mistake and properly signaled blame. For example, leaving out the call to `help` in the body of `link` results in this message:

```
(interface for itree-left) broke the contract
(-> itree? itree?) on itree-left;
expected <itree?>, given: b
```

This error correctly blames the interface for the import of the `itree-rest` selector.

6 Conclusion

The use of contracts to mediate between typed and untyped code holds great promise for enabling migration from scripts to programs. But it also can be used in contexts where types are already present, and *strengthen* the guarantees provided by the type system.

In this paper, we have described a simple problem involving the construction and processing of regular trees, and implemented solutions in three languages: Haskell, SML and Typed Scheme. In the Typed Scheme solution, we are able to avoid the infinite trees of the Haskell and lazy SML versions, and avoid the unnecessary wrapping of both the SML data definitions. We are left with a simple data definition for clients, with the complexity of construction hidden behind the module boundary and enforced by Typed Scheme in conjunction with the PLT Scheme contract system.

Acknowledgments

Many thanks to Matthias Felleisen, Jesse Tov, Fritz Henglein, Felix Klock, and the STOP 2009 anonymous reviews for their feedback on this paper.

References

1. Courcelle, B.: Fundamental properties of infinite trees. Theoretical Computer Science **25**(2) (March 1983) 95–169
2. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: from scripts to programs. In: OOPSLA '06: Companion to the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, New York, NY, USA, ACM Press (2006) 964–974
3. Freeman, T., Pfenning, F.: Refinement types for ml. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. (1991) 268–277