

TIP in Haskell

— another exercise in functional programming

Colin Runciman
University of York*

1 Introduction

Several years ago, Peyton Jones [3] tested some of the claims made for functional programming by re-implementing a well-known parser generator (YACC) in a lazy functional language (SASL) and comparing the result with the original imperative implementation. His conclusions were positive so far as the expressive power of functional programming was concerned — laziness and higher-order functions both proved valuable — but he bemoaned SASL’s lack of type-checking, abstract data types and modules, and also the difficulties of correcting errors and optimising performance in the presence of lazy evaluation.

This paper describes another experiment in the same spirit, but with a new programming language (Haskell) and a new application (TIP). Between the development of SASL (circa. 1975) and the development of Haskell (circa. 1990) there were several developments in functional programming systems, and we expect to see the benefits. Working on a fresh application presents fresh problems, and also retains the element of rethinking a procedural program in a functional language, not just translating between functional languages.

Haskell is a language recently designed by an international team, with the intention that it should become a common lazy functional language providing “faster communication of new ideas, a stable foundation for real applications development, and a vehicle through which others would be encouraged to use functional languages” [2]. Like several other functional languages, it has a polymorphic type system, abstract data types and modules. More distinctively, Haskell includes *type classes* for the systematic expression of ad hoc polymorphism, *array types* whose values are aggregated collections of indexed values, and an I/O scheme requiring every program to be a function whose result is a list of *requests* and whose argument is a list of *responses*.

TIP [8] is a software package for use in interactive terminal-based programs; it is an important part of a commercially successful tool-kit for the rapid construction of UNIX applications. TIP enables the application programmer to work with a

*Author’s address: Department of Computer Science, University of York, Heslington, York YO1 5DD, United Kingdom. Electronic mail: `colin@uk.ac.york.minster`

virtual terminal, abstracting away from the low level protocols and detailed screen-update strategies that are involved in communication with an *actual terminal*. There were two particular reasons for choosing TIP as a functional programming exercise:

1. it is an *abstraction of interactive I/O*, posing questions about program structure and modularity;
2. it requires the expression of a *complex and changing state with an indefinitely long lifetime*, posing questions about data abstraction and space-efficiency.

These are the questions we shall concentrate on. In the course of examining them, the I/O system and array types of Haskell will be discussed. Apparent opportunities to make use of the type class mechanism will also be discussed.

2 The Original TIP

The original TIP is implemented in C. We shall therefore refer to it as C-TIP to distinguish it from H-TIP, the Haskell version. An application program uses C-TIP by calling appropriate routines¹ from a large number available. Most routines are used to make changes to a *virtual screen* without any immediate effect on the *actual screen* of the computer terminal. But calling the special routine `trrefresh()` causes TIP to transmit an efficient update sequence to the actual screen so that its contents become those of the virtual screen. There is also a *virtual keyboard* in which multi-character input sequences, as transmitted by some keys of the *actual keyboard*, are mapped to numeric constants with mnemonic names.

For the application developer, the “Level 1” C-TIP interface can be summarised as in Figure 1. Note the predominance of `void` as result type; the significant outcome of calling one of these `void`-typed routines is not the result it returns, but the state-change it accomplishes within the C-TIP package. The collection of routines may appear modest, but some of them are quite sophisticated: boxes, for example, including lines as degenerate cases, may intersect yet be drawn or erased independently. According to the manual Level 1 is “sufficient for TIP to be used effectively on any terminal but with no more than one visual attribute [single colour, single font etc.]”. H-TIP is therefore restricted to Level 1, not to avoid particular problems that other levels pose², but to limit the scale of the programming exercise.

3 The Application Interface of H-TIP

Our first concern in the development of H-TIP must be to formulate the application interface. What is an appropriate functional equivalent of the C-TIP routines?

¹ The term *function*, though conventionally applied to C routines, is misleading when they are called almost entirely for their side effects.

² A *complete* re-implementation of TIP in Haskell would present a few difficulties; some of these are mentioned at the end of the paper.

<code>void tbell()</code>	<i>ring bell on next refresh</i>
<code>void tbox(xl,y1,xh,yh)</code>	<i>draw box given opposite corner co-ords.</i>
<code>void tnobox(xl,y1,xh,yh)</code>	<i>erase box ditto</i>
<code>void tttitle(xl,y1,xh,yh,s)</code>	<i>draw box with given title</i>
<code>void tclear()</code>	<i>clear the virtual screen</i>
<code>void tmove(x,y)</code>	<i>absolute move of virtual cursor</i>
<code>void trelmove(x,y)</code>	<i>relative move ditto</i>
<code>void tprintf(x,y,w,f,...)</code>	<i>print values given place, width and format</i>
<code>void tputc(c)</code>	<i>print given character</i>
<code>void tputs(s)</code>	<i>print given string</i>
<code>int tread()</code>	<i>read key from virtual keyboard</i>
<code>void trefresh()</code>	<i>send update to actual terminal screen</i>
<code>void tstart()</code>	<i>set initial conditions: must be first call</i>
<code>void tstop()</code>	<i>reset and release resources: must be final call</i>
<code>int txmax()</code>	<i>max column number on screen</i>
<code>int txpos()</code>	<i>current column of virtual cursor</i>
<code>int tymax()</code>	<i>max row number on screen</i>
<code>int typos()</code>	<i>current row of virtual cursor</i>

Figure 1: Level 1 C-TIP Interface.

3.1 Packages as Collections of “Stately” Functions?

Perhaps the routines should become what Fairbairn and Wray call *stately functions* [13]: the manipulation of state implicit in a procedural interface can be made explicit in a functional interface by the addition of state-valued arguments and by returning states as (components of) result values. Some C-TIP routines carry out I/O operations as a side-effect: H-TIP input streams can be incorporated into the state, and result structures can be further extended to allow for output. For example, the types of `tstart`, `tputc` and `trefresh` under this scheme might be:

```
tstart  :: String -> TipState
tputc   :: TipState -> Char -> TipState
trefresh :: TipState -> (TipState, String)
```

where the `String` argument of `tstart` represents the keyboard input stream, and the `String` component in the result of `trefresh` is the appropriate update sequence that should be transmitted to the terminal.

Although this approach of making state transitions and I/O traffic explicit in the function signatures is workable, it has two unfortunate consequences:

1. the application program is obliged to “carry around” the TIP state, to ensure that it is duly supplied as argument whenever a TIP function is used and to extract its new value from the result returned;
2. the application program is further obliged to conduct input and output associated with the actual terminal, acting as a low-level messenger conveying lists of characters to and from TIP.

These obligations are surely unwelcome news for the application programmer, who hoped using TIP was going to make life simpler! True, the necessary programming can be facilitated by suitable *combining forms* for stately I/O functions [9], or by adopting a *monadic* style of programming [10]. But the fact remains that by including functions with extended signatures in the interface, essential tasks that are properly the responsibility of the TIP package are instead put upon the application. This is both insecure and inconvenient.

3.2 Packages as Higher Order Functions

A quite different approach is to represent the entire package as a single function: this takes an *abstract program* as its argument and yields a *concrete program* as its result. In Haskell the type of a (concrete) program is

```
main :: [Response] -> [Request]
```

so an H-TIP application might be expressed as an abstract program of type

```
application :: [TipResponse] -> [TipRequest]
```

where the types `TipRequest` and `TipResponse` are suitably defined to represent the calls and results of C-TIP routines. This means that a TIP package represented as a higher order function should have type

```
tip :: ([TipResponse] -> [TipRequest]) -> [Response] -> [Request]
```

so that a complete concrete program can be expressed simply by applying `tip` to the abstract application.

```
main = tip application
```

Comparing this scheme with the one we considered before:

1. the manipulation of TIP state and all the associated low-level I/O is safely and conveniently encapsulated in the `tip` function where it belongs;
2. application programmers are on familiar ground, writing programs in which I/O is expressed using a request & response model — whatever auxiliary functions and programming techniques they usually use for I/O are likely to remain applicable.

So this is the scheme we shall adopt.

3.3 Abstract Requests and Responses

Working from the C-TIP routine headers of Figure 1, it is a straightforward exercise to write most of the corresponding definitions of `TipRequest` and `TipResponse` shown in Figure 2. The introduction of the type `YX` for screen co-ordinates is perhaps the most immediately apparent change, but other less obvious changes are more significant. The new relationship of function and argument between TIP and the application means, first, that there is no need for requests `Tstart` or `Tstop`: `tip` simply wraps the evaluation of its argument in an appropriate context. Secondly,

```

data TipRequest =
  Tbell
  | Tbox YX YX
  | Tnobox YX YX
  | Ttitle String YX YX
  | Tclear
  | Tmove YX
  | Trelmove YX
  | Tprintf YX Int String [String]
  | Tputc Char
  | Tputs String
  | Tread
  | Trefresh
  | Treq Request
  | Tmax
  | Tpos

data TipResponse =
  Tkey Char
  | Tres Response
  | Tvoid
  | Tyx YX

type YX = (Int,Int)

```

Figure 2: Level 1 H-TIP Interface.

since `tip` controls *all* I/O on behalf of the application (the function producing actual requests and consuming actual responses is the result of `tip`) it is necessary to provide some way of passing across the interface any I/O requests and responses that are *not* to do with the terminal (eg. those to do with file-handling). This is the reason for including `Treq` in the definition of `TipRequest`, and `Tres` in the definition of `TipResponse`.

Abstraction and Polymorphism

The `printf()` routines in C are classic examples of *ad hoc polymorphism*, something which Haskell classes are designed to express. Moreover, there is in Haskell a standard class `Text` of printable types. It is therefore surprising that one cannot define in H-TIP a `Tprintf` request with anything like the flexibility of C-TIP's `tprintf()` routine.

Suppose, for now, that we are willing to do without the interpretation of format control characters, using the `%` character in format strings as a simple place-holder. Then we might hope to make the following definition.

```
| Tprintf YX Int String [(Text a) => a]
```

But this is illegal: a context such as `(Text a)` cannot be embedded in a type; we cannot have “a list of values, each of some textual type”, the best we can hope for is “a list of values, each of the *same* textual type”. Reluctantly accepting this restriction, we revise the definition.

```
| (Text a) => Tprintf YX Int String [a]
```

But this too is illegal: types of constructor functions cannot have contexts in their own right. A context *could* be placed on the type of `TipRequest` as in

```
data (Text a) => TipRequest a =
  ...
  | Tprintf YX Int String [a]
  ...
```

but this defines a whole family of different request types; we want just one.

In fact, we are forced to give up the idea of polymorphic `Tprintf` requests altogether, making the values just strings (say).

```
| Tprintf YX Int String [String]
```

An application must itself apply `show`, or some other function, to values of `Text` types in order to form suitable `Tprintf` arguments. This requirement can be eased by providing a function to do the job.

```
tprintf :: (Text a) => YX -> Int -> String -> [a] -> TipRequest
tprintf yx w f xs = Tprintf yx w f (map show xs)
```

It seems strange that `tprintf` can exhibit polymorphism of a kind forbidden to `Tprintf` — though even `tprintf` restricts each value list to be of uniform type. A slight loss of polymorphism and one exceptional case in the application programmers' interface may not seem too bad, but being unable put the `show` application inside H-TIP, where each `TipRequest` is processed, could become a more serious problem. What if a full implementation is required that *does interpret* codes in the formatting string? Must the *application* be responsible for applying functions (supplied by H-TIP) to decompose the string, and to interpret its contents? What if the format also depends on information maintained within H-TIP? The lack of polymorphism in message passing interfaces, despite what type classes provide, is a significant limitation.

4 State and State-Transition

In order to service application requests, H-TIP has to maintain information about the state of the virtual terminal and the state of communication with the actual terminal. The overall definition of state in H-TIP is

```
data TipState = TS String [Response] Display YX YX Bool
```

where the `String` is a suffix of what the program receives along the `stdin` channel (from the actual keyboard), the `[Response]` is a suffix of the concrete response stream, the `Display` records the current contents of actual and virtual screens, the two `YX` co-ordinates are those of the actual and virtual cursors, and the humble `Bool` indicates whether the alarm is set to ring when the screen is updated.

Transitions corresponding to each `TipRequest` are coded in the defining clauses of a function `trans`.

```
trans :: TipState -> TipRequest ->
      (TipState, ([Request], TipResponse))
```

This stately signature is strongly reminiscent of the rejected scheme for the application interface — but we are now *within* the TIP package. The `tip` function itself can be defined along the following lines.

```
tip application rps =
  rqs where trqs = application trps
        (rqss,trps) = mapstate trans (stinit rps) trqs
        rqs = concat rqss
```

The function `mapstate trans` implements a state machine, which is applied to an initial state `stinit rps` and a list of TIP requests `trqs` forming the input stream for the machine. A dual output stream `(rqss,trps)` comprises actual I/O requests `rqss` (bundled into lists since there may be zero or many concrete I/O requests as a result of each abstract one) and the TIP responses `trps`.

4.1 Structure and Manipulation of Displays

In C-TIP, display information is represented by an array of line descriptors together with minimum and maximum line numbers for changes since the last refresh. Similarly, line descriptors are arrays of character cell descriptors with bounds on column numbers where changes have been made.

```
typedef struct { cell *col; subscr lmin, lmax; } line;

typedef struct { line *row; subscr smin, smax; } screen;
```

(These declarations have been slightly simplified: the real `line` and `screen` structures also contain wide-scale fonts, and the real `screen` structure contains things like cursor and alarm state too.)

Self-supporting Delta Structures

The close similarity of the `screen` and `line` structures suggests that, even if there is some variation in the operations performed on them, the two data structures should ideally be instances of the same polymorphic type. This type might be defined by

```
data DeltArray a = DA (Array Int a) (Maybe (Int,Int))
```

where the introduction of a `Maybe` type [6]

```
data Maybe a = Just a | Nothing
```

is preferred to the C-TIP representation of “no changes” which assigns special values to the bounds [4]. Now the `Display` and `Line` types can be defined as follows.

```
type Display = DeltArray Line
type Line    = DeltArray Cell
```

In order to define operations conveniently over the entire nested `DeltArray` structure, without breaking the data type abstraction, the module implementing `DeltArray` types is *self-supporting*: the polymorphic functions it defines meet their own requirements for functions over the parametric component type. A simple example by way of illustration is a function

```
daiap :: Int -> (a->a) -> DeltArray a -> DeltArray a
```

which yields an array with a pending change recorded for some index position, given a *change function* over the component type: the result of a partial application of `daiap` to two arguments is itself a possible change function. The benefits of this can be seen in the concise expression of update operations: for instance, given indices `x,y::Int` and function

```
f :: a -> Cell -> Cell
```

it is easy to formulate a corresponding `Display`-level function.

```
daiap y . daiap x . f :: a -> Display -> Display
```

A more complex example is a polymorphic function `daupd` which extracts a *delta value* representing all pending changes in a `DeltArray` and also computes a new `DeltArray` with these changes made and none pending. To do this it requires as its first argument a *delta function* for the component type — and a couple of other functions to convert between the two levels of delta values.

```
daupd :: (a->b->(a,b)) -> (Int->c->a) -> (a->c) ->
        c -> DeltArray b -> (c, DeltArray b)
```

The point to note is that any result of a partial application of `daupd` to three arguments is itself a possible delta function.

The discipline of self-support involves identifying the signatures of required functions, and ensuring that they are provided. This sounds like the role of type classes; but classes as currently defined in Haskell cannot express it, partly because they are restricted to a single type parameter³. Although we can *begin* an instance declaration of the form

```
instance (Delta a) => Delta (DeltArray a) where ...
```

we cannot complete it — or even define the `Delta` class adequately in the first place — because the relevant functions involve other parametric types. Even with multiple type parameters, we could only formulate specific classes such as `Delta`; there is no way to formulate the general class of self-supporting types.

Arrays and Indexed Trees

In the definition of the `DeltArray` type above, the underlying structure was shown as a Haskell `Array`. Since arrays are used in C-TIP, and arrays are regarded as an important provision of Haskell, the choice may seem obvious. However H-TIP actually uses balanced indexed trees because⁴ an explicitly defined data structure gives the option to share substructures, in ways not possible using arrays, and to exercise closer control over the timing of display evaluation (see section 4.2). So the definition of the `DeltArray` type becomes

³ Recent contributions posted to the `haske11` mailing list have proposed generalisations; in particular, a posting from Mark Jones at Oxford described his *Gofor* implementation of Haskell which supports multi-parameter classes.

⁴ Also, the prototype Haskell compiler used for H-TIP implements arrays as lists, incurring access and update costs linear in the size of the array.

```
data DeltArray a = DA (IndexTree a) (Maybe (Int,Int))
```

where the `IndexTree` type can be defined as follows.

```
data IndexTree a = Leaf a | Fork Int (IndexTree a) (IndexTree a)
```

The `Int` value held in a `Fork` construction is the maximum index for which the associated element is in the left `IndexTree` branch. Construction of an `IndexTree` from other values is by applying the function

```
itgen :: Int -> a -> IndexTree a
```

where `itgen n x` yields a tree with `n` leaves, all sharing the value `x`.

Why is the index type specified as `Int`? Shouldn't it rather be any type of `class Ix` as for arrays? No, it is not sensible to put `Ix` type indices in an `IndexTree`, since there is no efficient way to obtain `Ix` mid-points: in the context `Ix a` we have a function

```
index :: (a,a) -> a -> Int
```

but no inverse. Precisely because of the `index` functions, it would be straightforward to provide polymorphic indexing in a higher level data type. But in H-TIP, `Int` indices are fine.

Why bother to keep indices in the tree anyway? Wouldn't omitting the indices significantly reduce storage space occupied by trees? (Not only would each `Fork` construction be smaller, but also there would be more opportunity for structure sharing — eg. substructure representing blank parts of a screen.) This is a space/time trade-off. Storing indices speeds access by avoiding the need to refer to index bounds and also avoids a series of repeated mid-point calculations.

On Cells, Bits and Bags

Now consider individual character cells in the display. A `cell` in C-TIP is a `struct` with four fields: character values and fonts for each of the virtual and actual screens.

```
typedef struct { char vc,ac; font vf,af; } cell;
```

In Level 1 TIP it suffices to think of a `font` as having just two possible values: one indicates that the associated `char` is an ordinary character code, and the other that it is an eight bit byte code for a section in the edge of a box or boxes. C-TIP encodes edges in such a way that overlapping boxes, including boxes with some edges in common, can be created and deleted independently provided that any two boxes with a shared edge have centres on opposite sides of it. Figure 3 shows the effect of a `nobox()` call when an edge is shared by boxes centred (a) on opposite sides, and (b) on the same side. Something like the shared edge restriction is inevitable if the edge cells have a fixed size representation, because an application can request an arbitrarily deep overlay of boxes. Ideally TIP would maintain *bags* (or *multisets*) of edge segments to represent these cells. C-TIP uses a `char`-sized approximation because having a display state of fixed size simplifies memory management — a single allocation suffices, performed by `tstart()` — and enables the application programmer to predict accurately how much space C-TIP will need.

In H-TIP the similar but separate nature of the virtual and actual parts of the cell is expressed by defining a `Cell` to be a `HemiCell` pair.

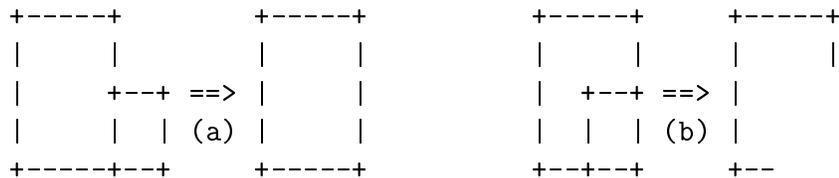


Figure 3: The shared edge restriction in C-TIP.

```
type Cell = (HemiCell, HemiCell)
```

Viewing the font information as the *tag* for the union of character codes and edge codes, suggests that the `HemiCell` type should be defined as follows.

```
type HemiCell = Plain Char | Fancy Bits
```

But this would perpetuate the shared edge restriction. In Haskell, dynamic memory management is (a) *inevitable* and (b) *free* (at least so far as programming is concerned), so why not do the job properly with bags? There isn't a `Bits` type with the desired characteristics anyway! So we redefine

```
type HemiCell = Plain Char | Fancy (Bag Char)
```

to support unrestricted sharing of edges. The implementation of bags must include a cheap operation to extract the associated set, because the function to derive appropriate printed characters from edge cells has to be efficient. One solution is to represent multisets as a sequence of non-empty sets $s_1 \supseteq s_2 \supseteq \dots \supseteq s_n$ in a list: the associated set is empty if the list is empty; otherwise the associated set is that at the head of the list.

Of course, we should like the bag implementation to be polymorphic and to be instantiated automatically given a suitable set type. But the class system cannot express the general case, *again* because of the restriction to a single parametric type. A putative instance declaration beginning

```
instance (SetEl s e) => BagSetEl [s] s e where ...
```

is not even syntactically well-formed. If we try hard, we can express a special case where sets and elements are identified in a single Boolean system (this is true in the H-TIP setting — for example, the character '+' can be used to represent the set {'-', '|'}) and the list of ordered sets is the only implementation of bags admitted. Figure 4 outlines the relevant declarations. The return does not seem great for the effort involved.

4.2 The Pragmatics of State Transition

Although the previous section described the `TipState` type in some detail, the *size* of its values was hardly mentioned. Excluding the residues of lazily evaluated input streams the `Display` component is dominant. On a largish physical display, such as an A3 workstation, there may be several thousand character cells. A *fully evaluated* `Display` value contains double this number of `HemiCell` structures in the worst case, and these are only the leaves of the `DeltArray` and `IndexTree` structures with their many numeric indices and bounds.

```

class (Eq a) => Booly a where
  add :: a -> a -> a
  sub :: a -> a -> a
  nul :: a

class Baggy a where
  ins :: a -> [a] -> [a]
  del :: a -> [a] -> [a]
  set :: [a] -> a

instance (Eq a) => Baggy a where ...

```

Figure 4: Polymorphic bags — a special case.

Laziness and Space Leaks

In the initial `Display`, however, a single blank `Cell` structure is shared across the entire `YX` space. Moreover, under the normal regime of lazy evaluation, a `Display` value is *most unlikely* to be fully evaluated. Branches of `IndexTrees` corresponding to parts of the screen not in use will be held as closures — unevaluated expressions. This sounds like good news. In practice, however, it is disastrous because the closures involved become ever more complex and enormous. Our intuition about state transition may suggest that previous states are “left behind”, and that the memory they occupied can be recovered. But under lazy evaluation, components of the current state may be held as closures including references to the previous state; this in turn may have component closures referring to the *previous* previous state, and so on. In short, the entire state-history of the program may steadily accumulate in memory. Indeed, in experiments with a “naturally lazy” version of H-TIP, using a benchmark test involving many `Tbox` and `Tnobox` requests over large screen areas, garbage collection was found to take 95% of the time⁵ and the rapidly expanding heap soon exhausted the available memory space of several megabytes.

The observation of this kind of *space-leak* in a lazy functional program is not new [7], and it has been made before in the context of long-running interactive programs [12]. In each application there are particular pitfalls to avoid — an example in H-TIP is inadvertently releasing a `TipState` to the application as part of a closure in a `TipResponse` — but the nature of the underlying problem is much the same.

Normalisation and Closure Space

Even if the *problem* has been discussed before, not much has been said about *solutions*. A special primitive, such as a hyperstrict identity function, is sometimes used to force evaluation. Even without such a primitive, a programmer can resort to tricks such as testing a structure for equality with itself! But such techniques are too indiscriminate to deal with values where some components, such as input streams, must remain unevaluated. Also, since a forcing function necessarily traverses every part of its result, it is a very expensive solution when the argument is a large structure most parts of which are already evaluated. Some languages permit constructor definitions to be annotated to show *strictness* in selected arguments. Bird and Wadler [1] assume a polymorphic primitive `strict` that can be applied

⁵ The semispace garbage collector repeatedly made copies of the large closures.

to *any* function: evaluation of `strict f x` reduces `x` to head normal form before applying `f` to it.

The solution adopted for the TIP program exploits the class mechanism to obtain both polymorphism and selective control of evaluation. We introduce a class `Norm` of types over which a predicate `normal` is defined.

```
class Norm a where normal :: a -> Bool
```

The intention is that `normal` functions are so-defined that the result of an application `normal x` is always `True`, but computing this truth guarantees that `x` is evaluated to at least a specified minimal extent. One way to declare a data type to be a `Norm` instance is simply to define `normal` to yield `True` for each possible outermost construction. The `IndexTree` type is a suitable example.

```
instance Norm (IndexTree a) where
  normal (Leaf _) = True
  normal (Fork _ _ _) = True
```

To avoid keeping an `IndexTree` with branches that are only (history-retaining?) closures any function building an `IndexTree` should not use the lazy `Fork` constructor but rather a *normalising constructor* `fork`.

```
fork :: Int -> IndexTree a -> IndexTree a -> IndexTree a
fork m lt rt | normal lt && normal rt = Fork m lt rt
```

If the `IndexTree` has a `Norm` component type, a normalising leaf constructor can be defined similarly.

```
leaf :: Norm a => a -> IndexTree a
leaf x | normal x = Leaf x
```

From this example it should be clear how `Norm` can be used to define data structures with selectively strict constructors, say. But we have more than that. The definitive constructors are lazy, and remain available. And the strictness of `leaf` is not fixed, for there are as many `leaf` functions as there are `Norm` types; some may not evaluate their argument at all, others may evaluate it fully, and still others may determine whether to evaluate one component of their argument by examining another.

In H-TIP, `normal` functions are applied *only* in `normal` definitions and in guards (where their role of forcing evaluation is akin to that of argument patterns). Use of `normal` in guards is further restricted to definitions of normalising constructors, apart from a single use in the main state-transition function. Consequently, normalisation is easily separated from other mechanisms in the program. The `HemiCell` structures with their edge multisets are left lazy, but in all other respects each successive `Display` is fully evaluated as a result of state normalisation after each transition. The benchmark test mentioned earlier was repeated for a normalising version of the program. The heap grew rapidly to about 100K bytes but then remained at about that size, sometimes shrinking, sometimes expanding.

5 Conclusions & Future Work

Many software packages written in a procedural language are similar in form to TIP — a collection of procedures operating on a shared state internal to the package, some also performing I/O. For example, there are database packages, graphics packages, equation-solving packages and so on. The use of an abstract response-request type in functional versions of such packages is an attractive alternative to the extended signature approach. Extended signatures may be more flexible if an application makes use of more than one package. However, by making the concrete request type a parameter of each abstract request type, it is possible to compose layers of several different response-request packages for use by a single application. For example, if `database` is a function representing a package, an abstract application program of type

```
application :: [DbResponse (TipResponse Response)] ->
              [DbRequest (TipRequest Request )]
```

can be made concrete by the following definition.

```
main = tip (database application)
```

Polymorphic data types with functional constructors are well established in functional languages: in H-TIP they once again proved pleasant to work with. The polymorphic type system caught most errors in programming with data structures; the one exception was an invariant violation involving `IndexTree` indices — this was easy to track down because it had such bizarre effects on the actual display! In due time, there will be more sophisticated implementations of Haskell arrays. A future compiler might compile an array-based version of H-TIP to code performing safe destructive updates: since there is just one abstract terminal and the interactive application is serial, it should be possible to make the use of display arrays *single threaded* or even *linear* [11].

It is disappointing to realise that despite the complex machinery of the present Haskell class system, there are constraints which curtail its use in practice. During the development of H-TIP, the restriction of classes to a single type parameter cropped up several times, and the limited scope for ad hoc polymorphism in data type constructions prevented the definition of an interface as flexible as the original. On the other hand, classes did provide a handy way to control space problems caused by lazy evaluation. Such problems are likely to arise in a wide range of applications, and it is important to devise programming techniques to solve them in a systematic, efficient and general way that does not intrude on a natural lazy style. The `Norm` class worked well for TIP, but surely more refined schemes are awaiting discovery.

We still do not have the tools to determine whether a functional program has a space leak of some particular class and, if so, which are the offending structures. A stable heap of 100K bytes is certainly better than a fast-growing one of several megabytes, but is the 100K byte figure what we should expect, and if so by what calculation? Even basic information from compilers about the size of individual constructions could be useful in this respect; ideally implementations would support *heap profiling* [5].

Finally, some of the functionality of the original TIP package could not be implemented in standard Haskell. For example, in C-TIP an optimisation inhibits

`refresh()` operations when the application user has “typed ahead”, interrupts can be generated from the keyboard and handled appropriately, and limits can be set to the time between key-strokes. There is as yet no consensus about the best way to program this kind of behaviour in a functional language.

Acknowledgements

Thanks to Dave Wakeling and Iain Checkland for comments and suggestions. H-TIP was implemented using a prototype Haskell compiler developed at Glasgow as part of the GRASP project; particular thanks to Kevin Hammond for speedy response to reported problems. The work described in this paper was supported by SERC (Science and Engineering Research Council) as part of the FLARE (Functional Languages Applied to Real Exemplars) project.

References

1. R. Bird and P. Wadler. *Introduction to functional programming*. Prentice-Hall, 1988.
2. P. Hudak and P. Wadler (editors). Report on the programming language haskell, a non-strict purely functional language (version 1.0). Technical report, Department of Computing Science, University of Glasgow, 1990.
3. S. L. Peyton Jones. YACC in SASL — an exercise in functional programming. *Software — Practice and Experience*, **15**(8):807–20, 1985.
4. C. Runciman. What about the *natural* numbers? *Computer Languages*, **14**(3):181–91, 1989.
5. C. Runciman and D. Wakeling. Problems and proposals for time and space profiling of functional programs. In *Proceedings of the 1990 Glasgow Workshop on Functional Programming*, pages 237–45. Springer-Verlag, 1991.
6. M. Spivey. A functional theory of exceptions. *Science of Computer Programming*, **14**(1):25–42, 1990.
7. W. Stoye. *The Implementation of Functional Languages Using Custom Hardware*. PhD thesis, University of Cambridge Computer Laboratory, 1985.
8. H. W. Thimbleby. The design of a terminal independent package. *Software — Practice and Experience*, **17**(5):351–67, 1987.
9. S.J. Thompson. Interactive functional programs. Technical Report 48, Computing Laboratory, University of Kent at Canterbury, 1987.
10. P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78. ACM Press, 1990.
11. D. Wakeling and C. Runciman. Linearity and laziness. In *Proceedings of 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 215–40. Springer-Verlag, 1991. LNCS 523.

12. S. C. Wray. *Implementation and Programming Techniques for Functional Languages*. PhD thesis, University of Cambridge Computer Laboratory, 1986.
13. S. C. Wray and J. Fairbairn. Non-strict languages — programming and implementation. *The Computer Journal*, **32**(2):142–51, 1989.