Haskell Ready to Dazzle the Real World

Martijn M. Schrage Arjan van IJzendoorn Linda C. van der Gaag

Department of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
{martijn,afie,linda}@cs.uu.nl

Abstract

Haskell has proved itself to be a suitable implementation language for large software projects. Nevertheless, surprisingly few graphical end-user applications have been written in Haskell. Based on our experience with the development of the Bayesian network toolbox Dazzle, we argue that the language is indeed very well suited for writing such applications. Popular language features, such as higher-order functions, laziness, and light syntax for data structures, turn out to hold their ground in a large interactive end-user application. Haskell, combined with the truly platform-independent GUI library wxHaskell, is ready for building real-world applications.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.3.3 [*Language Constructs and Features*]

General Terms Languages, Design

Keywords Haskell, Graphical user interface, Bayesian networks, wxHaskell, Application

1. Introduction

Haskell has come a long way from its first versions in the early nineties to the language it is now. The Glasgow Haskell Compiler [12] is a complete and stable compiler that produces efficient code, and is suitable for implementing industrial-strength software applications. With the development of the wxHaskell library [9], Haskell now has a truly platform-independent GUI library at its disposal. Nevertheless, most large Haskell applications seem to be either development tools or server-side applications, such as the GHC compiler itself, or the version management system darcs [14]. Other examples include a Haskell webserver [10], and the Flippi [2] and Postmaster [15] systems. Only very few large end-user applications with a graphical user interface have been built, which may leave critics of the language wondering whether Haskell is suitable for building such systems.

We argue that this small number of end-user applications has nothing to do with the language Haskell itself. While developing the Bayesian network toolbox Dazzle, we found that Haskell is very well suited for writing a large GUI-based application targeted at a user community of non-Haskell programmers.

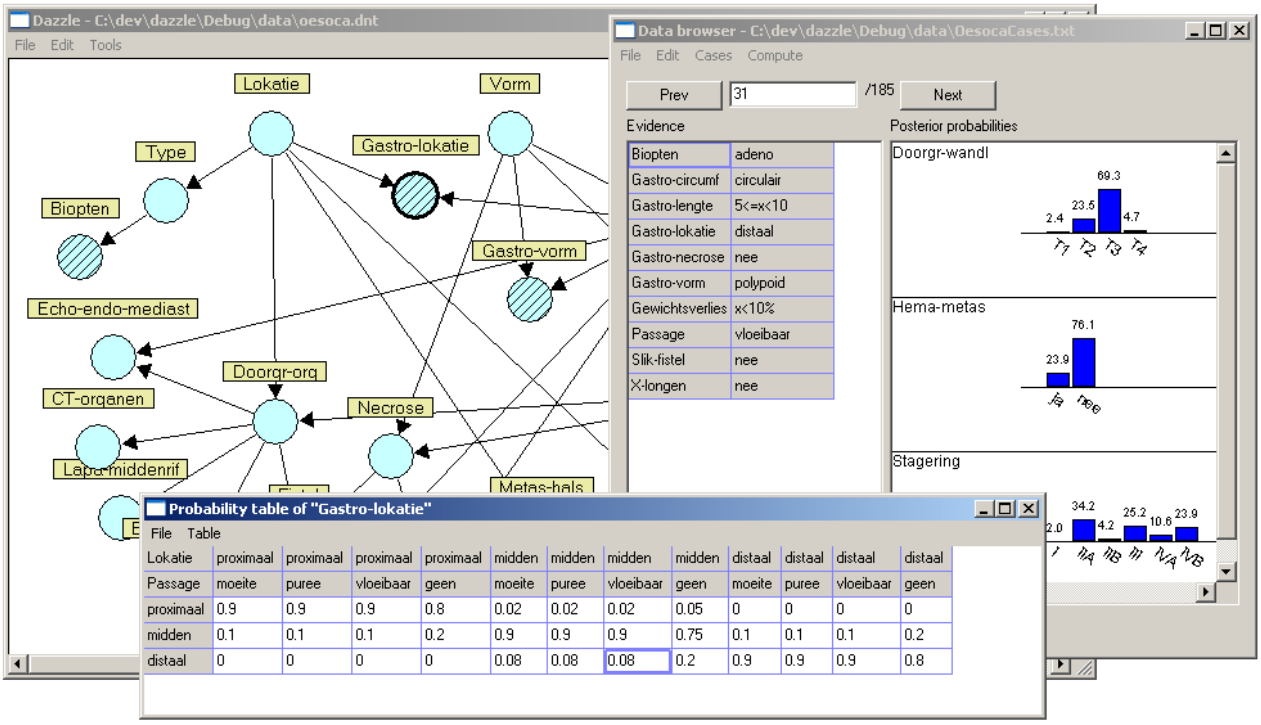
Haskell'05 September 30, 2005, Tallinn, Estonia.

Dazzle is a tool for constructing, editing and analyzing Bayesian networks, developed by the Decision Support Systems group of Utrecht University. It is a multi-platform application and is currently being used on both Windows and Linux platforms. Section 2 provides an overview of Bayesian networks and of Dazzle.

One of the most striking observations has been the speed of the development process. Often, features were implemented well before the planned deadline, which is rarely the case in software projects. The algorithms in Haskell stay close to their pseudo-code counterparts found in literature, both in elegance and size, and even improve them with respect to generality. Therefore, the source code is considerably smaller than that of similar systems. For comparison, Dazzle consists of 10,000 lines of code whereas the Java-based Elvira system [4], which has roughly the same functionality, consists of 220,000 lines of code.

In this paper, we provide experimental evidence that the Haskell features that work so well in toy examples still hold their ground in a large interactive application. Furthermore, we identify several patterns for which we have developed library modules that may be of interest to other Haskell implementors as well. More specifically, the following points are the contributions of this paper.

- We show that the advantages of the core features of Haskell (e.g. higher-order functions, laziness, and light syntax for data structures) also hold for a large GUI-based application. To our initial surprise, most beneficial were basic Haskell features rather than special language constructs or advanced types. In Section 3 we discuss the Haskell features that helped us most, followed in sections 4, 5, 6 and 7 by illustrations from the source code.
- Dazzle keeps track of two kinds of documents: a Bayesian network and patient data. Both kinds of documents need file management and undo handling, for which we developed a persistent-document abstraction, described in Section 4.
- Time-consuming pure algorithms pose a problem in a GUI application because user-interface events cannot be handled during their evaluation. In Section 6 we introduce an abstraction that keeps the user interface responsive and offers a progress bar and a cancel button, while keeping the algorithm itself pure.
- For the user interface, we have developed a set of higher-level controls on top of the wxHaskell GUI library. Together, these eXtended and Typed Controls form the XTC library, which is described in Section 7 and will be part of a future version of wxHaskell. The typed interface of XTC offers more ease of use and safety.
- Our application provides proof that the wxHaskell library scales to large applications. Although monadic GUI libraries are sometimes frowned upon as being too imperative for a functional language, we experienced that wxHaskell combined with the XTC library provides an effective means for building a sophisticated user interface.



Lokatie	proximaal	proximaal	proximaal	proximaal	midden	midden	midden	midden	distaal	distaal	distaal	distaal
Passage	moeite	puree	vloeibaar	geen	moeite	puree	vloeibaar	geen	moeite	puree	vloeibaar	geen
proximaal	0.9	0.9	0.9	0.8	0.02	0.02	0.02	0.05	0	0	0	0
midden	0.1	0.1	0.1	0.2	0.9	0.9	0.9	0.75	0.1	0.1	0.1	0.2
distaal	0	0	0	0	0.08	0.08	0.08	0.2	0.9	0.9	0.9	0.8

Figure 1. A typical Dazzle session.

2. Bayesian networks and Dazzle

Bayesian networks [7] in essence are models of uncertainty. While studying uncertainty traditionally is a field of statistics, it is currently showing an increasing impact from the field of computer science. With the advance of modern algorithmic techniques and powerful implementation constructs in fact, the field of computer science has rendered reasoning with complex probabilistic models feasible. Decision-support systems are now able to handle real-life problems in which uncertainty is predominant.

A Bayesian network is a concise model of a probability distribution over a collection of statistical variables that describe the characteristics of an application domain. It consists of a graphical structure, in which each node represents a statistical variable. The arcs in the structure capture the causal influences among the variables. For each variable, a probability table is specified that describes the strengths of these influences. A Bayesian network can be used to compute any probability of interest over its variables. For a diagnostic application, for example, the various features of a problem case are entered into the network, which is then used to compute the probability distribution over the possible diagnoses. Alternatively, if a diagnostic category is entered, the network can predict the features that will most likely be observed in a problem case typical of this category.

Bayesian networks are most suitable for addressing problems in domains that are scientific in nature. At Utrecht University, the Decision-Support Systems group is developing models for a range of problems in the biomedical domain, such as the staging of cancer of the oesophagus, the prediction of handicaps in premature newborns, and the early detection of classical swine fever in pig herds. These networks are being developed with the help of domain experts and are projected for use in a real-life setting. Other potential applications are weather forecasting and simulating the effects of climatological changes. Although most Bayesian networks have been constructed for applications of a scientific nature, other examples include models for predicting prices on the stock market and for printer trouble shooting.

Both to support the construction of real-life Bayesian networks and to arrive at fully operational decision-support systems, a Bayesian network toolbox is indispensable. Such a toolbox should provide not just the formalisms for capturing Bayesian networks and the associated algorithms for probabilistic reasoning, but also tools for the (automated) construction, editing and evaluation of networks. A number of commercial Bayesian network toolboxes are currently available, such as the well-known Hugin system [6]. Furthermore, a number of public-domain tools are available, such as Genie/Smile [3] and Elvira [4]. Until recently, the Decision-Support Systems group had worked with a public-domain Lisp-based system from the nineties to which various features had been added over the years, by a range of programmers. The system had become rather unmanageable and unacceptably slow. Since the currently available toolboxes lacked various features that were important to the group's ongoing research activities, we decided to re-design the toolbox, using the Haskell programming language.

The Dazzle toolbox (see Figure 1) now offers a variety of tools to support the (automated) construction, editing and evaluation of Bayesian networks. For probabilistic reasoning, the toolbox builds upon the C++ library of the public-domain Smile system. The graphical interface is closely tailored not just to the end-users of Bayesian networks but, even more importantly, to the network designers. As such, it offers a larger variety of options for editing, storing and retrieving partially constructed models than other currently known toolboxes. The graphical interface effectively shields the Haskell specifics from the end-users and designers, some of whom have no background in computer science. Experience with the Dazzle toolbox shows that it effectively and efficiently supports the construction of large real-life Bayesian networks.

3. Haskell features that matter to Dazzle

Over fifteen years ago, John Hughes advocated the importance of higher-order functions and laziness in his paper "Why functional programming matters" [5]. Although, at the time, not many large applications in a functional language existed (and in fact neither did the language Haskell), we found that his statements indeed remain to hold for large systems. In addition to higher-order functions and laziness, we identified several other features that we found to be important:

- Light syntax for data types
- Purity
- Static typing

Note that these features correspond to the often mentioned strong points of Haskell. In this sense, Dazzle confirms that these features retain their value in a large GUI-based application.

For the largest part, the source code conforms to the Haskell '98 standard with the exception of the XTC library, which makes use of scoped type variables and multi-parameter type classes. In developing Dazzle, we did not use many advanced Haskell features. Of course, this does not say anything about the usefulness of these features: Dazzle is a specialized application with mostly numerical algorithms, and other applications may very well benefit from Haskell extensions.

We take a closer look at the above-mentioned features, which is followed by several more concrete examples in the subsequent sections.

Higher-order functions

Higher-order functions are essential to a programming pattern that we used several times. First, we simply implement an algorithm for a specific instance of a problem. Then, once the algorithm has been implemented and tested, an aspect of it is captured by a function that is passed as a parameter. The result is a more general algorithm that can be parameterized for several instances of the problem. The process leads to more reuse, and at the same time, the refactorings required to abstract from the specific feature often make the algorithm more elegant. Examples of the result of this process are the classifier-learning algorithm presented in Section 5 and the persistent-document abstraction in Section 4.

Laziness

Apart from many small uses of laziness, the implementation contains several more interesting examples. The classifier-learning algorithm examines a search space of alternative classifiers, which is represented by a lazy data structure and is evaluated only as far as necessary. Another example is the glueing together of the pure and lazy hill climber and the monadic and interactive progress bar (see Section 6). The loopy-propagation inference algorithm (not discussed in this paper) also heavily relies on laziness. This algorithm produces a lazy list of approximations, and is parameterized with a function representing the stop condition, much like Hughes's implementation of Newton-Raphson square roots.

Light syntax for data types

List manipulations are commonplace, but appear most notably in the numerical algorithms of classifier learning and loopy propagation. Many algorithms on Bayesian networks involve formulas consisting of sums or products of complex terms. List comprehensions together with the `sum` and `product` functions turned out to provide an elegant and concise mechanism for implementing these algorithms. The final code generally stays close to the mathematical specification.

In addition, there are several tree-based algorithms that benefit from the light syntax for traversing and constructing abstract data structures. The hill climber from the classifier module is an example of this.

Purity

The absence of side effects is useful for reasoning about and understanding a program. This is especially important for a large software project, since a substantial part of development time tends to be spent on inspecting code, rather than writing it. Furthermore, not having side effects makes it possible to safely abstract any part of a function and replace it by a parameter.

As there are no destructive updates in a pure language, data structures can safely be shared in the heap. This fact has greatly simplified the implementation of the undo facility, which is part of the persistent-document abstraction. After each edit operation, we store the previous document value in a history list. An undo operation simply amounts to selecting one of the previous documents. The memory consumption by the document list is kept small due to sharing between successive documents.

Static typing

Even though the process of type checking does not prevent logical errors, it seems to catch the majority of programming errors. Once a function is accepted by the type checker it is often correct.

A good example of where we use the Haskell type system to catch more programming errors can be found in the XTC library (see Section 7). The typed interface guarantees that values retrieved from user-interface widgets have the correct type. Furthermore, user selections in widgets are represented by values rather than indices. As a result, XTC helps to make the user-interface code safer and also shorter since many conversions are performed automatically.

4. Persistent documents

Both the network and the collection of cases in the data-browser tool are documents that are stored on disk. For both kinds of documents we want to provide undo and redo operations. To avoid code duplication we have devised a small and very flexible framework called `PersistentDocument` that manages storage and undo/redo operations for a document. The reason for combining these two is because there are (subtle) interactions between them. When a document is saved, for example, the documents in the undo and redo buffers have to be marked as dirty (that is, changed with respect to the disk version). We describe the framework because of its generality, but also because the implementation relies on Haskell being pure, which makes it possible to share data structures in the heap.

To use the framework the programmer has to supply several call-back functions. Some of these have to do with updating the user interface after a change, and others with prompting the user of the application for input. The framework takes cares of all administration and calls the user-supplied functions at appropriate times.

4.1 The type of persistent documents

Since we want to do destructive updates of a document, we use an IO reference:

```
type PersistentDocument doc = IORef (PDRecord doc)
```

The `PersistentDocument` type is polymorphic in the type of the document that it manages. The record `PDRecord` contains all the information the framework requires:

```
data PDRecord doc = PD
   { document       :: doc

   -- file management
   , fileName       :: Maybe String
   , dirty          :: Bool

   -- undo & redo
   , history        :: [(String, Bool, doc)]
   , future         :: [(String, Bool, doc)]
   , limit          :: Maybe Int

   -- call-back functions
   , updateUndo     :: Bool -> String -> IO ()
   , updateRedo     :: Bool -> String -> IO ()
   , updateSave     :: Bool -> IO ()
   , updateTitleBar :: Maybe String -> Bool -> IO ()
   , saveToDisk     :: String -> doc -> IO Bool
   , saveAsDialog   :: Maybe String -> IO (Maybe String)
   , saveChangesDialog :: IO (Maybe Bool)
   }
```

The file name and dirty bit are stored for file management. The file name can be `Nothing` in case the document has never been saved to disk. This can be presented to the user as a document with name "Untitled". The dirty bit is usually visualized by writing "(modified)" or "*" behind the file name.

The `history` and `future` lists store undo and redo information respectively. They do not only store a specific version of the document, but also the corresponding dirty bit and a message describing the edit action leading to that version of the document. This message can be shown to the user to indicate what will be undone or redone (e.g. "Undo remove node"). The undo buffer may be limited to a certain size using the `limit` field.

There are seven call-back functions that need to be provided by the library user. Default implementations for applications using wxHaskell are provided for all but `saveToDisk`. This call-back is supposed to write the document to disk, and we cannot give a default implementation since nothing is known about the document type. The update functions update the user interface with new information. The function `updateTitleBar` gets the current file name and dirty bit and presents this information in the window title bar for example. The other update functions update the respective menu items.

The remaining two call-back functions prompt the user for input; the `saveChangesDialog` asks whether the user wants to save the changes to the document upon closing. `Nothing` is returned if the user cancels, and otherwise a boolean constant indicating whether to save or not. The `saveAsDialog` asks the user for a file name and returns `Nothing` in case the user cancels.

The framework takes care of handling cancellation by the user at different points. If the user tries to close an untitled document that has been modified, he or she will be asked whether to save the changes or not; if so, a save-as dialog is shown. This process can be cancelled at any point and may even be cancelled by the system, for example if the disk is full.

Note that the types of the call-back functions are not wxHaskell specific. This means that the framework can be used equally well with other GUI libraries; it can even be used to build an application with a textual interface. The defaults provided use the wxHaskell library, but they are defined in a separate module and are not required to use the framework.

The undo and redo buffers are implemented naively as lists of past and future documents. For our specific network data structure, the documents in these lists are shared as much as possible, and thus in a sense only the differences between the documents are stored. Heap profiling shows that making many changes to a large network without a limit on the undo buffer costs little memory. In an impure language such a naive implementation would store a complete copy of the document every time, even if there is only a small change, and memory usage could easily become a problem. Possible solutions are to either mimic sharing or use invertible edit actions to store the difference between documents. Both are less elegant and more error-prone than the simple implementation in Haskell.

4.2 Operations on persistent documents

The framework offers many functions to manipulate persistent documents and these functions in turn call the supplied call-back functions. Here we will look at four typical operations:

```
setDocument ::
  String -> doc -> PersistentDocument doc -> IO ()
superficialSetDocument ::
  doc -> PersistentDocument doc -> IO ()
isClosingOkay ::
  PersistentDocument doc -> IO Bool
undo ::
  PersistentDocument a -> IO ()
```

The function `setDocument` changes the document stored in the persistent document data structure. It appends the current document to the undo buffer with the given message (e.g. "remove node"), clears the redo buffer and marks the document as dirty. The user interface will then be updated to reflect these changes.

The function `superficialSetDocument` also sets the document but neither changes the undo buffer, nor sets the dirty bit. This is useful if something as volatile as the selection is part of the document; in most applications, changes to the selection cannot be undone and thus the document should not be marked dirty when the selection changes.

The function `isClosingOkay` checks whether it is safe to close the document. If the document has been modified, the user is prompted with a dialog that asks whether changes should be saved. If the answer is yes, a save dialog may follow. If in the end the document was saved, `True` is returned. If the answer is no or the user cancelled the process somewhere down the line, `False` is returned.

Finally, `undo` takes the first element of the undo buffer and uses it as the new document. The old document is moved to the redo buffer. Here is the actual code of the `undo` function:

```
undo :: PersistentDocument a -> IO ()
undo pDocRef =
 do { pDoc <- readIORef pDocRef
    ; when (not (null (history pDoc))) $
 do { let (msg, newDirty, newDoc) = head (history pDoc)
          newPDoc = pDoc
            { document  = newDoc
            , dirty     = newDirty
            , history   = tail (history pDoc)
            , future    = ( msg, dirty pDoc
                           , document pDoc)
                              : future pDoc
            }
    ; writeIORef pDocRef newPDoc
    ; updateGUI pDocRef
    }}
```

To conclude, the persistent-document framework deals with a lot of subtle issues when writing an editor. It takes care of file management, undo and redo operations, and their interactions. The framework leaves the choice of the GUI library open and its implementation relies on Haskell being pure and sharing data structures for efficient undo management.

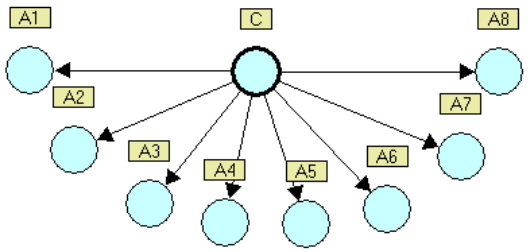


Figure 2. A naive classifier.

5. Learning classifiers

A classifier is a simple network that can be automatically derived from a data set. It consists of a class variable and several attributes. Figure 2 shows an example classifier with class variable "C" and attributes "A1" up to "A8". The purpose of a classifier is to predict the value of the class variable based on the values of the attributes. The figure shows a classifier with no arcs between the attributes. More advanced classifiers with better classification performance allow for trees or even polytrees on the attributes. Such structures on the attributes are referred to as *dependence structures*.

Often, not all attributes are equally relevant for the classification process, and the performance of the classifier can be improved by using only a selection of the attributes. Dazzle's *attribute-selection algorithm* employs a hill-climbing search to determine a (locally) optimal selection of attributes. To determine the performance of a classifier, the algorithm uses a scoring function, which weighs both the size of the classifier (smaller is better), and how well it models the data set. The search can start with a classifier containing all attributes and remove attributes until the score degrades (backward search) or start with an empty classifier and add attributes (forward search).

Altogether there are three parameters in searching for the best classifier: the dependence structure on the attributes, the direction of the search and the scoring function. In the literature on classifier learning, each combination of these parameters results in a separate algorithm. In contrast, thanks to higher-order functions and laziness, the Haskell implementation is a single algorithm that is parameterized with the necessary ingredients. It uses a lazy data structure to represent the search space.

Although the algorithms presented in this section are not novel, the interesting aspect is that they have been taken directly from the Dazzle sources. Often, an algorithm loses some of its elegance when it is incorporated in an application, because exceptions need to be handled and output needs to be relayed to the user. The classifier algorithms are an example of the contrary; elegant core algorithms can be part of the actual implementation.

Search strategy

The search strategy is encoded by a function that builds a search tree for a set of attributes. First, we introduce the data type for search trees.

```
data SearchTree a = SearchNode a [SearchTree a] String
```

Each node in a search tree of type `SearchTree a` contains a value of type `a`, a list of subtrees, and a message string that is shown during the search process. The type constructor `SearchTree` is a functor; a simple map function `mapSearchTree :: (a->b) -> SearchTree a -> SearchTree b` maps a function on all values in the search tree. We omit its definition here.

The search strategy that should be adopted during attribute selection is determined by a parameter of type `SearchTreeBuilder String`, which is a function that takes a list of attribute names and returns a search tree in which the nodes contain different subsets of this list of names. The general type of a search tree builder is:

```
type SearchTreeBuilder a = [a] -> SearchTree [a]
```

A forward search tree is a tree with no attributes in its root, and in which a child has one attribute more than its parent. Each leaf contains all attributes. The algorithm for building the backward search tree is similar.

```
forwardSearchTree :: SearchTreeBuilder String
forwardSearchTree attrs =
  makeTree [] attrs "Classifier without attributes"
 where makeTree bs as msg =
         SearchNode bs [ makeTree (a:bs) (as\\[a])
                                  ("Added attribute '"++a++"'")
                       | a <- as ] msg
```

Scores and dependence structures

In addition to the search strategy, the attribute-selection algorithm has two more parameters: the score function for the classifier and the type of dependence structure that is built on the attributes:

```
type ScoreFunction = DataSet -> Classifier -> Double

type StructureBuilder = DataSet -> Classifier
                     -> [NodeNr] -> Classifier
```

The score function takes a data set and a classifier and returns the score of the classifier. The tree builder takes a data set, a classifier, and a list of node numbers denoting the attributes on which a dependence structure must be built. The result is a classifier with a dependence structure on its attributes.

Attribute selection

Now we are ready to define the attribute-selection function itself.

```
attributeSelect :: DataSet -> String -> [String]
                -> ScoreFunction
                -> SearchTreeBuilder String
                -> StructureBuilder
                -> (Classifier, [String])
attributeSelect dataSet classVariable attributes
                scoreFun searchTree structureBuilder =
  let attributesSearchTree = searchTree attributes
      classifierSearchTree =
        mapSearchTree (makeClassifier
                         dataSet classVariable
                         attributes structureBuilder)
                      attributesSearchTree
  in  hillClimb classifierSearchTree
                (scoreFun dataSet)
```

The `attributeSelect` function creates an appropriate search tree by calling the search-tree builder argument on the list of attributes for the classifier. The resulting search tree contains lists of attributes in its nodes. By mapping `makeClassifier` onto the attribute search tree, we (lazily) get a search tree that contains classifiers in its nodes. The partially parameterized application of `makeClassifier` takes a list of attribute names, constructs a classifier with the appropriate dependence structure, and learns its probability tables from the data set:

```
makeClassifier :: DataSet -> String -> [String]
               -> StructureBuilder -> [String]
               -> Classifier
```

The function `hillClimb` takes as arguments a search tree containing values of a type `a`, together with a score function for values of type `a`. The result is a pair of the result of the search (in our case a classifier) and a list of strings explaining the steps of the search process. We present a simplified version first, which undergoes a minor modification in Section 6 to allow progress reporting and user cancellation of the algorithm.

```
hillClimb :: SearchTree a -> (a -> Double)
          -> (a, [String])
hillClimb tr scoreFn = climb tr (scoreNode tr) where
  scoreNode (SearchNode val _ _) = scoreFn val
  climb (SearchNode val children msg) score =
    let scored = [ (c, scoreNode c) | c <- children ]
        sorted = reverse $
                   sortBy (\(_,x) (_,y) -> compare x y)
                          scored
        (r, prgs) =
          case sorted of
            ((c,sc):_) ->
              if sc > score
              then climb c sc
              else (val, ["Local maximum."])
            [] -> (val, ["No more children."])
    in  (r, msg : prgs)
```

The local function `climb` gets two parameters: a search tree and the score of the root node of that search tree. The algorithm sorts the child search trees based on their score and if the highest score (`sc`) is higher than the current score, hill climbing continues at the corresponding child (`c`). Otherwise, the current score is a locally optimal score, and its corresponding value is returned as the result. If the sorted list of children is empty, the current value is also returned as the result.

6. A lazy progress bar

In a GUI-based application computations are performed in callback functions. Because these functions are evaluated in the main thread, the application will stop responding if a certain algorithm takes a long time. In order to stay responsive, the algorithm should perform GUI-library calls that temporarily return control to the event handler. As these calls are monadic, this creates a problem when the time-consuming function is pure. Other problems with pure functions are that progress cannot be monitored and that the computation cannot be cancelled. We solve these three problems by using an abstraction of progress, and letting a special progress dialog lazily evaluate the result.

The main idea behind the progress abstraction is that instead of evaluating the result of a computation directly, we evaluate an accompanying list of progress steps, which is produced lazily by the algorithm. Once the progress list has been fully evaluated, the result itself is also evaluated. Others have used the technique of lazily returning a list of steps [1, 18] but not in the context of an interactive application.

The hill climber, discussed in the previous section, returns a list of messages, which we can evaluate step by step. However, each step can still take quite a long time, since for a single message, all children in the search tree need to have their classifier evaluated and scored. To make the computation of the progress list more fine grained, we introduce a data type `ProgressItem`, in which we represent both messages and explicit progress steps. We introduce a type synonym `Progress` for a list of progress items.

```
type Progress = [ ProgressItem ]

data ProgressItem = Message String | Tick
```

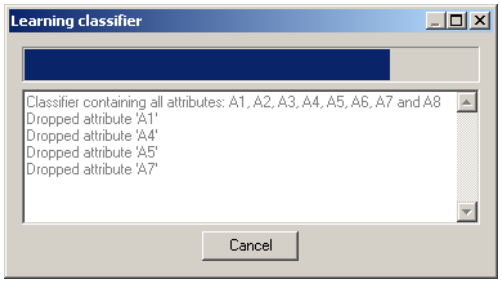


Figure 3. The lazy progress bar in action.

With a slight modification, we can get the hill climber to produce a `Tick` for each classifier that is scored.

```
hillClimb :: SearchTree a -> (a -> Double)
          -> (a, Progress )
hillClimb tr scoreFn = climb tr (scoreNode tr) where
  scoreNode (SearchNode val _ _) = scoreFn val
  climb (SearchNode val children msg) score =
    let scored = [ (c, scoreNode c) | c <- children ]
        ticks  = [ seq sc Tick | (_,sc) <- scored ]
        sorted = reverse $
                   sortBy (\(_,x) (_,y) -> compare x y)
                          scored
        (r, prgs) =
          case sorted of
            ((c,sc):_) ->
              if sc < score
              then climb c sc
              else (val, [Message "Local maximum."])
            [] -> (val, [Message "No more children."])
    in  (r, Message msg : ticks ++ prgs)
```

A list of ticks is returned along with the message in the second element of the result of `hillClimb`. Because of the application `seq sc Tick`, each `Tick` that is evaluated forces the evaluation of a classifier score, causing the corresponding classifier to be created, learned, and scored. Hence, after evaluating the ticks for each of the children of a certain node in the search tree, the list of scores can be sorted without delay.

Now we can evaluate the list of ticks one by one, and when the list has been fully evaluated, the result of the search algorithm has also been evaluated. The `progressDialog` takes care of this:

```
progressDialog :: Window a -> String -> Int
               -> (b, Progress) -> IO (Maybe b)
```

The `progressDialog` function takes as arguments a parent window, a title, a maximum number of ticks, and a pair containing a result and its progress. If the cancel button is pressed, the dialog returns `Nothing`. Otherwise, it returns `Just result`, where `result` is the fully evaluated result. Figure 3 contains a screenshot of the progress dialog. The dialog shows a bar for the progress and displays the messages in the progress list.

A progress dialog needs to know the maximum number of ticks a computation may require. In case of attribute selection, this number of ticks is easily computed from the search strategy. However, the number represents only a possible maximum amount of time. Often, the hill climber will terminate before the leaves of the search tree are reached, and hence the bar will not reach the end.

The implementation of the progress dialog is surprisingly simple. At its heart is a monadic loop that traverses the progress list. On a `Message`, the message text is displayed in the dialog, whereas on a `Tick`, the progress bar advances. After each step, a function is

called to handle GUI events. The loop continues only if the cancel button has not been pressed.

The progress-bar abstraction makes it easy to separate a computation from user-interface details. A multithreaded approach might seem more appropriate to handle responsiveness, but it does not handle progress indication, as the algorithm still needs to provide cues about its progress. The modifications shown in this section can be applied to other time-consuming algorithms in Dazzle as well.

7. XTC: eXtended and Typed Controls

The wxHaskell library [9] offers a standard set of widgets (called controls), such as buttons, text fields, radio buttons, and selection lists. It is built on top of the C++ library wxWidgets [16] and provides an interface of a much higher level. Programs using wxHaskell are considerably smaller than C++ wxWidgets programs. Still, data that is shown in a control is accessed in the form of strings and list selections are represented by integer indices, which are rather low-level abstractions for a strongly typed language such as Haskell. As a remedy, we developed the module `XTC`, which defines a set of extended and typed controls for wxHaskell. The controls are typed versions of existing wxHaskell controls.

As an example, we look at a text field in which a user can enter a value of type `Double`. It is created with `entry <- textEntry frame [ text := "3.14" ]`. The value is accessible through the `text` attribute, which is of type `String`. At any point where the value is read, we must parse the string to a double, and analogously, when setting the value, it must be converted to a string. Furthermore, if the value in the text field is not a proper floating point number, some kind of visual feedback would be desirable.

A second example illustrating the need for a higher-level abstraction is found in radio buttons and selection lists. These controls typically show a set of values, whereas the selection is returned as an index. After retrieving the selection, we have to perform a lookup on exactly the list of values that was used when setting the items for the control.

XTC controls keep track of typed values and items, rather than being string based. Selections in XTC controls consist of actual values instead of indices. In the near future, the XTC library will become part of the wxHaskell distribution. Although the implementation of the library is small (around 300 lines), it uses relatively advanced wxHaskell features that fall beyond the scope of this paper. Hence, we only discuss the programming interface of the controls and give examples of their use.

7.1 wxHaskell

Before we introduce the XTC controls, we discuss a few aspects of the wxHaskell library. Below is a small wxHaskell program that creates a window containing a text entry and a set of radio buttons. Figure 4 shows a screenshot of the program.

```
main = start $
 do { f <- frame [ text := "Sample" ]
    ; radio <- radioBox f Vertical
                       [ "Naive Bayesian"
                       , "Tree augmented naive Bayes"
                       , "k-dependence Bayesian" ]
                       [ text := "Dependence structure" ]
    ; textEntry <- entry f [ text := "3.14" ]
    ; set radio [ on select := handler textEntry radio ]
    ; set f [ layout := column 5 [ widget radio
                                 , widget textEntry ] ]
    }
 where handler textEntry radio =
       do { n <- get textEntry text   -- n :: String
          ; s <- get radio selection  -- s :: Int
          ; putStrLn $ n ++ ", " ++ show s
          }
```

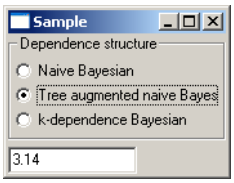


Figure 4. A sample wxHaskell program.

With the function `frame` we first create a window with title "Sample" to which we add a set of radio buttons and a text field. The text field is created with `entry` and contains the initial text `"3.14"`. The function `handler` is attached to selection events of the radio buttons. On selection, the value of the text entry (with type `String`) and the current selection of the radio buttons (with type `Int`) are printed. In the last step of the program, the layout of the frame is set to a column that contains the two controls, separated by a vertical space of 5 pixels.

Attributes and properties

Constructor functions such as `frame` and `entry` take a list of *properties* (e.g. `text := "Sample"`) that specify the behavior and appearance of the created control. Properties are constructed with the `:=` data constructor, which takes an attribute and a value. We can access and modify the values of properties with the functions `get` and `set`:

```
get :: forall a w. w -> Attr w a -> IO a
set :: forall w. w -> [Prop w] -> IO ()
```

The attribute type (`Attr w a`) has two parameters: `a` is the type of the value of the attribute, and `w` is the type of the widget of which it is an attribute. Because attributes are defined for classes of widgets, their types include a context. For instance, for `text` we have:

```
text :: forall w. (Textual w) => Attr w String
```

These complex types are awkward in a discussion, and therefore we will simply refer to the type of the attribute value as being the type of that attribute. Thus, we will refer to `text` as having type `String`, rather than `forall w. (Textual w) => Attr w String`.

Phantom types for inheritance

To model the inheritance relation from the underlying C++ library wxWidgets, wxHaskell employs so-called *phantom types* [8, 9]. Every wxHaskell object is represented by a value of type `Object a`, which has a type variable `a` that is not reflected in the value itself; regardless of its type, each object is represented by a plain machine address. The phantom type parameter in a wxHaskell type is used to represent its inheritance path. To encode the path, wxHaskell defines a dummy type without values for each object in the hierarchy. For example, for `Window` and `Frame`, we have:

```
data CWindow a
data CFrame a
```

Using the dummy types, the inheritance paths are encoded as follows.

```
type Window a = Object (CWindow a)
type Frame a = Object (CWindow (CFrame a))
```

Thus, a function taking a `Window a` argument will also accept a `Frame a`, but a function taking a `Frame a` will not accept a `Window a`. It is also possible to enforce an exact match: an argument of type `Window ()` must be exactly a window, and not a subclass. The encoding works for both co- and contravariant arguments. A more extensive discussion of inheritance encoding using phantom types can be found in Daan Leijen's PhD thesis [8].

7.2 Typed Controls

We discuss two typed controls in more detail: the `valueEntry` and the `radioView`. Together, these two controls cover most aspects of typed controls. The remaining XTC controls are discussed at the end of this section.

ValueEntry

A `ValueEntry` is a typed version of a `TextEntry`. It has the same appearance in the user interface, except that it changes color when its text cannot be parsed. Rather than setting the text in the value entry directly as a string, we set it to a certain value, which is converted to a string by the value entry. If we get the contents of a value entry, it parses its text and returns a typed value.

Similar to `mkTextEntryEx`, the constructor `mkValueEntryEx` takes a parent window and a list of properties as arguments. In addition, it takes a parameter of type `x -> String` for presenting its value and a parameter of type `String -> Maybe x` for parsing.

```
mkValueEntryEx :: Window a
               -> (x -> String) -> (String -> Maybe x)
               -> [Prop (ValueEntry x ())]
               -> IO (ValueEntry x ())
```

The type of the returned value entry has two parameters, `x` and `()`. The `x` denotes the type of its value and the `()` is the phantom type used to encode inheritance. Note that the `()` phantom type means that the result is exactly a value entry and not a subclass. In contrast, the type of the window parameter of `mkValueEntry` has a free type variable `a`, meaning that the parameter may be a window or a subclass.

XTC also provides a convenience function `mkValueEntry` that uses `show` and `read` for presenting and parsing the value.

```
mkValueEntry :: (Show x, Read x)
             => Window a -> [Prop (ValueEntry x ())]
             -> IO (ValueEntry x ())
```

In an ordinary text entry, the content of the entry is accessed through the attribute `value` of type `String`. In contrast, a value entry of type `ValueEntry x a` also has an attribute `typedValue` of type `Maybe x` that contains the typed content. A `Maybe` type is used because the text in the value entry might not correctly parse to a value of type `x`, in which case `get` returns `Nothing`. Unfortunately, the wxHaskell framework does not allow the `get` operation of an attribute to have a different type from the `set` operation. Hence, we need to use `Just` when setting the value. If `typedValue` is set to `Nothing`, nothing happens.

If we let `f` denote the frame, we can create a value entry that contains the value `3.14` with:

```
vEntry <- mkValueEntry f [ typedValue := Just 3.14 ]
```

The `get` operation for `vEntry` has type:

```
get vEntry typedValue :: IO (Maybe Double)
```

If the value entry contains an incorrect string when a user presses return or when the entry loses focus, the background color of the value entry is set to light grey.

RadioView

The `RadioView` is the typed counterpart of the `RadioBox` control. Instead of providing a list of strings for the radio items, and retrieving the selected item in the form of an integer, the radio view allows a list of values for its items. The selection is returned as one of these values, rather than an index. In order for the radio view to be able to show the value items, it needs a function of type `x -> String` that converts a value to its label. The function `mkRadioViewEx` creates a radio view:

```
mkRadioViewEx :: Window a -> (x -> String)
              -> Orientation -> [x]
              -> [Prop (RadioView x ())]
              -> IO (RadioView x ())
```

The constructor function takes as arguments the parent window, a label function, the orientation (`Horizontal` or `Vertical`), a list of radio items, and a list of properties. In addition to the attribute `selection` of type `Int`, a radio view of type `RadioView x a` has an attribute `typedSelection` of type `x`.

An example from Dazzle shows how radio views are used. The classifier-learning algorithm has a parameter that specifies the type of dependence structure that is built on the attributes. There are three alternatives: no structure (Naive), a TAN tree, or a k-DB polytree. The data type `DependenceStructure` represents these alternatives:

```
data DependenceStructure = Naive | TAN | KDB
```

For the textual representation that should appear in the user interface, we define a function `label`:

```
label :: DependenceStructure -> String
label Naive = "Naive Bayesian"
label TAN   = "Tree augmented naive Bayes"
label KDB   = "k-dependence Bayesian"
```

Now, we can create a radio view for the three alternatives:

```
treeRadio <- mkRadioView frame label Vertical
               [ Naive, TAN, KDB ]
               [ typedSelection := Naive
               , text := "Dependence structure" ]
```

We can obtain the selection through the `typedSelection` attribute, which returns a value of type `DependenceStructure`:

```
get treeRadio typedSelection :: IO DependenceStructure
```

The `typedSelection` is also set with a typed value rather than an index. Items are compared based on their labels to avoid requiring that items are an instance of `Eq`. If the radio view contains items with duplicate labels, the set operation selects the first of the duplicates.

Similar to the value entry, there is also a class-based function `mkRadioView`. Instead of taking a label function, it requires the type of its items to be an instance of the `Labeled` class:

```
mkRadioView :: Labeled x => Window a -> Orientation
            -> [x] -> [Prop (RadioView x ())]
            -> IO (RadioView x ())
```

The definition of the class `Labeled` is

```
class Labeled x where
  toLabel :: x -> String
```

Note that we do not use the `Show` class for displaying an item in a radio view. The reason for this is that a label is often a rather verbose representation that would be awkward to have as a `Show` instance.

Other controls

The XTC library offers four more typed controls: `ListView`, `MultiListView`, `ChoiceView` and `ComboView`. Because their behavior and interface is largely similar to the radio view, we only discuss them briefly.

A `ListView a` shows a list of values of type `a`, in which a user can make a selection. The list view is similar to the radio view, except for its appearance, and the fact that its items do not have to be set at creation time. This is reflected in the type of its constructor function:

```
mkListView :: Labeled x
           => Window a -> [Prop (ListView x ())]
           -> IO (ListView x ())
```

In contrast to `mkRadioView`, `mkListView` has no parameters for the orientation and the list of items. The orientation is always vertical, and the items are set through the `typedItems` attribute. For a list view of type `ListView a`, the attribute `typedItems` has type `[a]`.

A `MultiListView` is a list view that allows multiple elements in the selection. Instead of a `typedSelection` attribute, a `MultiListView a` has an attribute `typedSelections`, which has type `[a]`.

Finally, there are the `ChoiceView` and the `ComboView`. A choice view is a list view in which only the current selection is shown. The other items are available through a drop-down menu that appears when the choice view is clicked. The interface of the choice view is equal to the list view interface. A combo view is a choice view that, in addition to selecting from a list, allows textual editing of the selection. The interface is again equal to the list view.

Conclusion

The XTC library provides a higher-level abstraction on several existing wxHaskell controls. The XTC controls are both easier to use and safer, because conversions between the strings or indices and actual values are performed automatically. The library will be part of a future wxHaskell distribution.

8. General observations

Besides our experiences with specific language features discussed in the previous sections, we made a number of general observations in the process of developing an application in Haskell. These observations concern the performance and reliability of the implementation, as well as the development process itself. Although most of the claims made in this section are hard to substantiate, we believe they are worth sharing.

Performance

The code that GHC generates for our project is very efficient in terms of execution speed. Dazzle replaces the LISP-based system Ideal [17] and is an order of magnitude faster; operations that took hours now take minutes. For probabilistic inference we have reused the optimized C++ library SMILE that underlies Genie [3] to save development time early on in the project. Therefore, algorithms that rely heavily on inference will automatically be efficient. In the near future, we will implement inference in Haskell making the application independent of SMILE.

It is interesting to note that we have paid little attention to optimizing the code. We have always been able to focus on elegance and correctness, and this has resulted in efficient algorithms. Not having to worry about performance helped to create correct code quickly.

Memory performance, which for Haskell is closely related to execution speed, is also satisfactory. There are no space leaks that we are aware of, and the program uses a modest amount of memory (less than 32Mb). Heap profiles indicate that memory is released at appropriate times. Moreover, Dazzle users often run the application for hours on end, and we have had no reports on excessive memory usage or crashes.

Reliability

Stability is an important issue for all applications, especially those you trust your data with. In a strongly-typed programming language that uses garbage collection, two major classes of runtime errors are eliminated. What remains are the applications of partial functions to values outside the domain, and system-related errors such as an out-of-memory exception. These remaining exceptions are caught and displayed to the user urging him or her to tell the developers about the problem. After this, it is probably wise to save data and exit the program. This way of dealing with exceptions has proven to be useful and will be included in the wxHaskell distribution at some point. Fortunately, the mechanism has only been triggered three times in the year and a half that Dazzle has been in use. The amount of logical programming errors has also been small, especially when compared to the authors' experience with large C++ and Java projects.

Development environment

We have only used poor man's debugging tools such as the functions `trace` and `putStr` and interactive testing. Thanks to Haskell's abstraction facilities, algorithms can be expressed in small functions that can be tested independently.

We believe that the development environment for Haskell could easily be improved upon. The combination of your favorite editor and GHCi works well, but there are features that would make the programming process more pleasant. For example, we would like to be able to quickly see the type or definition of a function even when the current module cannot be compiled. The Clean IDE provides such an environment for the Clean programming language [13]. There are efforts to create an IDE for Haskell (e.g. Haste [19] and an integration with Eclipse [11]), and we hope that a usable integrated environment will soon become available. The convenient layout combinators of wxHaskell make the absence of a GUI builder less important.

9. Conclusion

The Dazzle toolbox is a large end-user application with a sophisticated graphical user interface. Except for an inference algorithm written in C++, the entire application is implemented in Haskell, using wxHaskell for its user interface. The implementation provides evidence that basic Haskell features scale well to larger applications. The features that proved especially beneficial are: laziness, higher-order functions, light syntax for data types, purity, and static typing.

Besides the application itself, the project has produced several spin-offs: a progress indicator for pure algorithms, an abstraction for persistent documents, and the XTC library for typed controls.

Future work will include the addition of more functionality as well as a Haskell implementation of the inference algorithm. With its own inference algorithm Dazzle will be independent of the C++ SMILE library. Furthermore, it will be interesting to compare the inference algorithms with respect to performance.

The development of Dazzle depends on the reliable and efficient compiler GHC and the platform-independent GUI library wxHaskell. With these two tools we believe that Haskell is ready for building real-world applications. In other words, functional programming still matters!

Acknowledgments

We would like to thank Bastiaan Heeren and his Advanced Functional Programming students for their useful comments on an earlier version of this paper. Furthermore, we thank the anonymous referees for their detailed comments and suggestions. For inference we use the SMILE reasoning engine for graphical probabilistic models contributed to the community by the Decision Systems Laboratory, University of Pittsburgh [3]. This research was (partly) supported by the Netherlands Organisation for Scientific Research (NWO).

References

[1] O. Chitil. Pretty printing with lazy dequeues. *Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):163–184, January 2005.

[2] P. Cowderoy. Flippi: a wiki clone written in haskell. `http://www.flippac.org/projects/flippi`.

[3] M. Druzdzel et al. Genie and SMILE. `http://www.sis.pitt.edu/~genie`.

[4] Elvira Consortium. Elvira: an environment for probabilistic graphical models. In *First International Workshop on Probabilistic Graphical Models (PGM02)*, Sept. 2002.

[5] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[6] Hugin Expert A/S. Hugin Expert. `http://www.hugin.com`.

[7] F. Jensen. *Bayesian Networks and Decision Graphs*. Springer, 2001.

[8] D. Leijen. *The λ Abroad – A Functional Approach to Software Components*. PhD thesis, Department of Computer Science, Universiteit Utrecht, The Netherlands, Nov. 2003.

[9] D. Leijen. wxHaskell – a portable and concise GUI library for Haskell. In *ACM SIGPLAN Haskell Workshop (HW'04)*. ACM Press, Sept. 2004.

[10] S. Marlow. Writing High-Performance Server Applications in Haskell, Case Study: A Haskell Web Server. In *Haskell Workshop*, Montreal, Canada, September 2000.

[11] Object Technology International, Inc. Eclipse platform – a universal tool platform. `http://www.eclipse.org`.

[12] S. L. Peyton Jones et al. The Glasgow Haskell Compiler. `http://www.haskell.org/ghc`.

[13] R. Plasmeijer, M. van Eekelen, et al. The Clean programming language. `http://www.cs.ru.nl/~clean`.

[14] D. Roundy. darcs. `http://abridgegame.org/darcs`.

[15] P. Simons. Postmaster esmtp server. `http://postmaster.cryp.to`.

[16] J. Smart, R. Roebling, V. Zeitlin, R. Dunn, et al. The wxWidgets library. `http://www.wxwidgets.org`.

[17] S. Srinivas and J. Breese. IDEAL: A software package for analysis of influence diagrams. In *Proceedings of the Sixth Uncertainty Conference in AI*, Cambridge, MA, September 1990.

[18] D. Swierstra. Combinator parsers: From toys to tools. In G. Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, 2001.

[19] D. Waern et al. haste – Haskell TurboEdit. `http://haste.dyndns.org:8080/news.php`.