Application Programming Interface for *fxp*

The following is basically an excerpt from Andreas Neumann's Ph.D. Thesis [1], Chapter 2, Section8.

The *fxp*'s application interface is a functional variant of an event-based interface that vitally relies on SML's parametrized modules for customization.

The application defines a data type AppData representing the part of its state affected by the event handlers. We call this information the *application data*. During parsing, a value of this type is maintained by the parser. Upon triggering an event, the event handler receives the application data as an additional parameter and returns it – possibly modified – to the parser. The modified application data is then an argument to the next event, and so forth. In order to distinguish this kind of event handlers from the imperative variant we call them *hooks*. The effects of all hooks are thus accumulated into one value of type AppData.

In order to implement hooks in *fxp*, the parser must be provided with the type of the application data and with the functions implementing the hooks. The easiest way of doing so is to pack them into a structure Hooks, fulfilling the signature in Figure 1. The purposes of the single hooks are listed in Table1. A hook expects as argument the current application data and – except for hookFinish – the information belonging to the signaled event, and returns the modified application data.

The data types providing the event-specific information passed to a hook are defined by structure HookData. Each information item contains at least the position in the document where the event occurred; for some events even two positions are specified: a start position and an end position. E.g., type ErrorInfo describes an error by means of its position and an error description. A start-tag is described by the positions of its first and last character, the index of the element, the list of attribute specifications the eventual spaces following the element name and a boolean flag, indicating whether it is an empty-element tag:

```
type ErrorInfo = Errors.Position * Errors.Error
type StartTagInfo = StartEnd * int * AttSpecList * UniChar.Data * bool
```

The information set provided through *fxp*'s hooks is sufficient for producing a character-by-character identical copy of the document instance. Except for white space and parameter entity references within declarations, this is also possible for the DTD.

In addition to type AppData, a Hooks structure must define a type AppFinal. This type is the result type of the parser: At the end of the document, the parser calls hookFinish in order to *finalize* the accumulated application data. This is useful because values of type AppData often contain auxiliary information: E.g., if

```
signature Hooks =
  sig
    type AppData
    type AppFinal

    val hookError      : AppData * HookData.ErrorInfo      → AppData
    val hookWarning    : AppData * HookData.WarningInfo    → AppData

    val hookStartTag   : AppData * HookData.StartTagInfo   → AppData
    val hookEndTag     : AppData * HookData.EndTagInfo     → AppData

    val hookData       : AppData * HookData.DataInfo       → AppData
    val hookCData      : AppData * HookData.CDataInfo      → AppData
    val hookCharRef    : AppData * HookData.CharRefInfo    → AppData

    val hookProcInst   : AppData * HookData.ProcInstInfo   → AppData
    val hookComment    : AppData * HookData.CommentInfo    → AppData
    val hookWhite      : AppData * HookData.WhiteInfo      → AppData
    val hookDecl       : AppData * HookData.DeclInfo       → AppData

    val hookDocType    : AppData * HookData.DtdInfo        → AppData
    val hookSubset     : AppData * HookData.SubsetInfo     → AppData
    val hookExtSubset  : AppData * HookData.ExtSubsetInfo  → AppData
    val hookEndDtd     : AppData * HookData.EndDtdInfo     → AppData

    val hookGenRef     : AppData * HookData.GenRefInfo     → AppData
    val hookParRef     : AppData * HookData.ParRefInfo     → AppData
    val hookEntEnd     : AppData * HookData.EntEndInfo     → AppData

    val hookXml        : AppData * HookData.XmlInfo        → AppData
    val hookFinish     : AppData                           → AppFinal
  end
```

Figure 1: The Hooks signature.

an application collects information from the document entity, it must ignore all events occurring within entity replacement texts. For this reason, the hooks might maintain a counter indicating the nesting depth of included entity references. hookFinish then removes this counter because it is of no interest to the application. Another example for the benefits of finalization is the following: An application might collect section titles in order to compile a table of contents. Since the application data is an accumulating parameter, these titles are collected in reverse order. AppFinal can be used for reestablishing the original order.

1 Functorizing the Parser

The parser is therefore implemented as an SML *functor*, expecting the Hooks structure as argument. The application can then generate several instances of the parser for different purposes. In addition to the hooks, the parser functor is parametrized with three other structures:

Options: This structure supplies the parser with its options. It is useful to have each instance of the parser run with its own set of options: E.g., XML catalogs frequently have no DTD. Therefore a catalog is parsed in non-

Hook	XML Event
hookError hookWarning	error warning
hookStartTag hookEndTag	start-tag end-tag
hookData hookCData hookCharRef	segment of character data CDATA section character reference
hookProcInst hookComment hookDecl	processing instruction comment element, attribute, notation or entity declaration
hookDocType hookSubset hookExtSubset hookEndDtd	start of the DTD start of the internal subset start of the external subset end of the DTD
hookGenRef hookParRef hookEntEnd	general entity reference parameter entity reference end of an included entity
hookXml hookFinish	start of the document entity end of the document

Table 1: The hooks in *fxp* and their purposes.

```
funsig Parse ( structure Dtd      : Dtd
               structure Hooks    : Hooks
               structure Resolve  : Resolve
               structure Options  : Options ) =
  sig
    val parseDocument : Uri.Uri option → Dtd.Dtd option
                        → Hooks.AppData → Hooks.AppFinal
  end
```

Figure 2: The Parser functor.

validating mode, even if the main document is validated.

Resolve: This structure provides a single function for resolving an external identifier to a URI:

```
val resolveExtId : Base.ExternalId → Uri.Uri
```

In the simplest case this function returns the system identifier that is part of the external identifier. If the parser supports XML catalogs, however, this is the function that searches the catalog.

Dtd: The implementation of the DTD tables can be provided by the application. In most cases this is the Dtd structure provided by *fxp*, but the application can also provide a more efficient implementation, or enhance the functionality of the DTD tables. For instance, the operations on the DTD tables might be wrapped into functions producing debugging or statistical information. On the other hand, the application can *hard-code* element types or attributes to fixed indices. E.g., in order to collect `href` attributes in an XHTML document, one might use the following implementation of the Dtd argument structure:

```
structure HrefDtd =
  struct
    open Dtd

    val hrefData = UniChar.String2Data "href"

    fun initDtdTables () =
      let val dtd = Dtd.initDtdTables()
          val _ = AttNot2Index dtd hrefData
      in dtd
      end

    val hrefIdx = AttNot2Index (initDtdTables()) hrefData
  end
```

A `href` attribute will then always have index hrefIdx; searching and comparing can be done with this constant rather than the list of characters [0wx68,0wx72,0wx65, 0wx66]. This is reasonable also because element types are passed to the hooks by means of their indices in the DTD, not by their names.

The signature of the parser functor is given in Figure 2. It defines a single function parseDocument which, given an optional URI of a document and an optional DTD, parses that document – if no URI is given, the document is read from the standard input. It returns the finalized value of the application data received in its third argument, modified by the hooks during parsing.

If the optional Dtd argument is given as NONE, the parser initializes the DTD tables with the initDtdTables function. In this case hooks have no access to the DTD because it is not provided as an argument to them. For many applications this is not necessary indeed. If the hooks need to access the DTD, the application must initialize the tables itself, incorporate them into the application data and pass them to the parser in its Dtd argument.

2 Building Applications with *fxp*

fxp provides a rich information set through its hooks interface. Many applications, however, are only interested in a subset of that information: A formatter is probably not interested in comments, and a querying tool will only search the document instance and ignore the DTD. Therefore we provide a set of hooks that simply return the application data unchanged:

```
structure IgnoreHooks =
   struct
      fun hookError(a,_) = a
      fun hookWarning(a,_) = a
      fun hookStartTag(a,_) = a
      ...
      fun hookXml(a,_) = a
      fun hookFinish a = a
   end
```

These functions are polymorphic: They do not depend on the type of the application data and can thus be used with arbitrary types. The only exception is hookFinish which requires types AppData and AppFinal to be equal. An application must now only define the hooks that have a different behavior. E.g., the hooks for a validator are implemented in a few lines, because it only prints errors and warnings but ignores all other events:

```
structure CheckHooks =
   struct
      open TextIO Errors IgnoreHooks

      type AppData = OS.Process.status

      fun message(pos,msg) = output(stdErr,Position2String pos ^": "^msg)

      fun hookError (_,(pos,err)) =
         OS.Process.failure before message(pos,errorMessage err)
      fun hookWarning (status,(pos,warn)) =
         status before message(pos,warningMessage warn)
   end
```

Except for hookError and hookWarning, all hooks are taken over from structure IgnoreHooks. Another example is the application already mentioned on page 4: It collects all attributes named `href`. For this purpose we defined a structure HrefDtd, hard-coding this attribute name to a constant index hrefIdx. This is used in the definition of the hooks:

```
structure HrefHooks =
   struct
      open Base HrefDtd IgnoreHooks

      type AppData = UniChar.Vector list
      type AppFinal = AppData

      fun findHref nil = NONE
        | findHref ((idx,attPres)::rest) =
            if idx <> hrefIdx then findHref rest
            else case attPres
                     of AP_PRESENT(vec,_) => SOME vec
                      | AP_DEFAULT(vec,_) => SOME vec
                      | _ => findHref rest

      fun hookStartTag (a,(_,_,attSpecs,_,_)) =
         case findHref attSpecs
           of NONE => a
            | SOME x => x::a

      val hookFinish = rev
   end
```

In order to identify a `href` attribute, function findHref need only compare its index with hrefIdx. Note that due to the accumulative nature of the AppData argument, the attribute values are collected in reverse order. Therefore function hookFinish is defined to be the list reversing function and reestablishes the original order.

3 Implementing a Tree-Based Interface

The hooks interface of *fxp* its event-based; nevertheless a tree-based interface can easily be implemented on top of the hooks. We present bellow an implementation of such an interface. For brevity, it has only a restricted information set: It ignores the DTD, comments, processing instructions and the entity structure of the document. It is, however, easy to extend the implementation to supply all this information.

The Tree data type is simple: A tree is either a piece of text or an element consisting of a start-tag and a list of trees as content. The application data represents the partial document tree constructed so far. It contains in its stack component for each ancestor element of the current position, its start-tag together with the list of its – already complete – left siblings; component content holds the children of the current element that are known so far. At the beginning of the parse both components are empty. After the whole document has been parsed, the stack must be empty and a single element tree must have been constructed. In this case function hookFinish returns that element; otherwise it raises an exception.

The three functions hookData, hookCData and hookCharRef add the piece of text reported to them to the content of the current element. The hook for a start-tag pushes that tag together with the content of the current element onto the stack. The element started by that tag now becomes the current element. Function hookEndTag reverses the content of the current element in order to compensate for the reversing effect of accumulation. Its tag is popped from the stack and combined with its content. The constructed tree is then prepended to the content of the parent element which now becomes the current element.

4 Other examples

fxp comes with a few application examples which resides in the `src/Apps` directory. The interested reader can find there further examples of using the *fxp*'s programming interface.

References

[1] Andreas Neumann. Parsing and Querying XML Documents in SML. University of Trier, 2000. Available at `http://www.informatik.uni-trier.de/~neumann/Papers/thesis.ps.gz`

```
structure TreeData =
   struct
      exception IllFormed

      type Tag = int * HookData.AttSpecList
      datatype Tree = TEXT of UniChar.Vector
                    | ELEM of Tag * Content
      withtype Content = Tree list
   end


structure TreeHooks =
   struct
      open IgnoreHooks TreeData UniChar

      type AppData = Content * (Tag * Content) list
      type AppFinal = Tree

      val appStart = (nil,nil)

      fun hookStartTag ((content,stack), (_,elem,atts,_,empty)) =
         if empty then (ELEM ((elem,atts),nil)::content,stack)
         else (nil,((elem,atts),content)::stack)

      fun hookEndTag ((_,nil),_) = raise IllFormed
        | hookEndTag ((content,(tag,content')::stack),_) =
         (ELEM (tag,rev content)::content',stack)

      fun hookData ((content,stack),(_,vec,_)) =
          (TEXT vec::content,stack)
      fun hookCData ((content,stack),(_,vec)) =
          (TEXT vec::content,stack)
      fun hookCharRef ((content,stack),(_,c,_)) =
          (TEXT(Data2Vector [c])::content,stack)

      fun hookFinish ([elem],nil) = elem
        | hookFinish _ = raise IllFormed
   end


structure ParseTree :
   sig
      val parseTree : Uri.Uri option → Dtd.Dtd option → TreeData.Tree
   end
= struct
      structure Parser = Parse ( structure Dtd           = Dtd
                                 structure Hooks         = TreeHooks
                                 structure ParserOptions = ParserOptions ()
                                 structure Resolve       = ResolveNull)
      fun parseTree uri dtd = Parser.parseDocument uri dtd TreeHooks.appStart
   end
```

Figure 3: A simple tree-based interface on top of hooks.