GenI: Natural language generation in Haskell

Eric Kow
INRIA/LORIA/UHP
eric.kow@loria.fr

Abstract

In this article we present GenI, a chart based surface realisation tool implemented in Haskell. GenI takes as input a set of first order terms (the input semantics) and a grammar for a given target language (e.g., English, French, Spanish, etc.) and generates sentences in the target language, whose semantic meaning corresponds to the input semantics.

The aim of the article is not so much to present GenI or to describe how it is implemented. Rather, we will focus on the aspects of functional programming (higher order functions, monads) and Haskell (typeclasses) that we found important to its design.

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

General Terms Languages, Design

Keywords Haskell, Applications, Computational Linguistics, Surface Realisation, Typeclasses, Monads, Profiling.

1. Natural Language Generation

Natural Language Processing (NLP) is the field that deals with the automatic processing of natural, human, languages like English, French, Spanish, etc. Some typical NLP tasks are, for example, speech recognition (e.g., as a component in a telephonic help system), information extraction (e.g., as part of a search engine in an electronic collection of scientific articles), speech synthesis (e.g., in a public announcement system), etc. In some way or other, most NLP systems transform input in natural language (either spoken or written) into some abstract representation which an algorithm can manipulate, or, vice-versa, transform abstract information into text or speech in some natural language. This second direction (from an abstract representation into a recognisable human language) is usually called Natural Language Generation.

GenI, the tool we are going to describe in this article, is a component within a natural language generation system. More precisely, GenI is a surface realiser, and deals with the last steps of the generation process. Once the generator has decided *what* to communicate (the semantic content of the output) and *how* (e.g., in which order the information should be conferred, and which are the different linguistic relations between the pieces of information that should be generated), the surface realisation module works at sentence level to actually 'build' the output, given a proper grammar for a target natural language.

Haskell'06 September 17, 2006, Portland, Oregon, USA.

In a concrete example, the main task of GenI consists of translating a logical formula such as

$$\texttt{loves(l,j,m)} \wedge \texttt{john(j)} \wedge \texttt{mary(m)}$$

into the sentence

John loves Mary.

when an English grammar is provided. If a French grammar would be specified instead, the output would be

Jean aime Marie.

More formally, the input accepted by GenI is a set of first order terms (the *input semantics*) and a grammar of the target language (GenI uses Feature Based Tree Adjoining Grammars, FB-TAGs, as input [22]). Before surface realisation, GenI first selects the grammar rules that correspond to the input semantics (e.g., `love(l,j,m) john(j), mary(m)`) and post-processes them into 'chart items' (GenI uses a 'chart based algorithm' for generation, see [8]). It is not relevant to this paper what a chart item is. What is important, as we will see below, is to note that different chart algorithms for surface realisation use different types of chart items.

After the initial selection step, GenI starts the surface realisation proper, where the bulk of the work lies. Most chart generation algorithms are variants of the following theme: systematically compare all the chart items with each other, and when possible, combine them into larger structures. The resulting structures are also chart items and need to be compared with the other known chart items. When there are no more comparisons to be made, the algorithm finishes, extracting suitable sentences in the target language from the completed structures in the chart.

The surface realisation task has been shown to be NP-complete [10]. This is in striking contrast with the parsing task (transforming an input sentence into some abstract, internal representation) which can be solved in polynomial time (at least, for the formalism we are using, FB-TAG). Intuitively, the reason for the difference is that there exist many different alternatives for exactly how every piece of information to be conferred can be expressed in the target language, and all these intermediate options should potentially be explored until one successful realisation is found. Nevertheless, some researchers have observed polynomial time behaviour of generation [5] on real linguistic data. Or in other words, when dealing with real applications the risk of an exponential behaviour seems to be low.

In any case, this potential computationally very expensive behaviour is a threat, and the algorithmic complexity of the task requires careful implementations. Clearly, if a natural language generator is to be integrated into any interactive application, it will need to be sufficiently reactive so that the user does not experience the frustration of constantly waiting for the system to respond. Seeing that surface realisation is only part of the generation problem, it is especially important to get satisfactory realisation run-times.

Our main aim in developing GenI is to investigate different algorithms for surface realisation, each of them with different possible

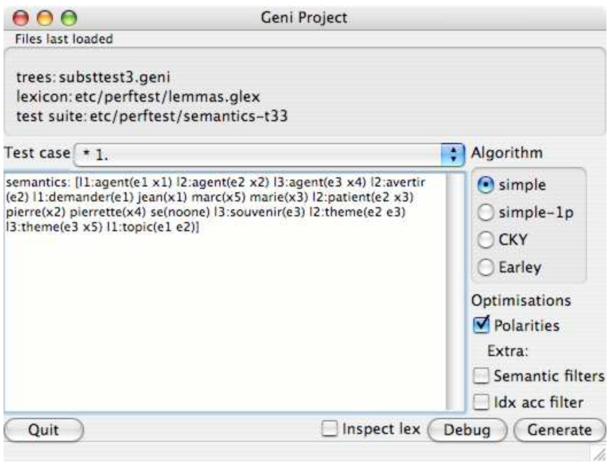

Figure 1. The main user interface

optimisations and heuristics, and to determine how they perform on actual data. This makes GenI a complicated piece of software:

- GenI implements different surface realisation algorithms, each with its own requirements of particular input, internal representations, etc.; and
- GenI explores different sets of optimisations, and some of them conflict with others or may only be used in the presence of another, some are common to all surface realisation algorithms while others are particular to one of them.

As should be clear from the previous paragraphs, the surface realisation problem is already complex in itself. And things get even more complicated because we are experimenting with different possible algorithms, optimisations and heuristics.

To give a rough feel for the size of the task, the entire source code of GenI is about 14 000 lines of literate Haskell (8 000 lines without comments). This includes two main surface realisation algorithms (a "simple" one and a tabular one based on the CKY parsing algorithm for TAG [21]) each with variants (a two-phase version of the simple algorithm, and an Earley-like version of the tabular algorithm, based on [18]) and optimisations (polarity filtering [11], index accessibility filtering [5], among others); a file format converter between XML, a GenI-specific text format and generated Haskell; as well as a console interface; a client/server user interface; and finally a graphical interface (Figure 1) with multiple debuggers[1].

Naturally, we would like to be able to keep GenI's code modular, and reuse as much common functionality as possible.

We write this paper to share our experiences implementing GenI because we found that Haskell, and the functional programming paradigm in general, helped us to keep our code clean and concise, despite the difficulty of the task. We shall also discuss what we learned from this experience, and what we still find to be problematic. Finally, we will wrap the paper up mentioning some small contributions to the Haskell community, these are solutions to some of the problems we encountered while developing GenI that we feel are general enough to apply to other domains.

[1] We need an interactive debugger for each algorithm because data is so complex that it is impractical to debug the surface realiser without a graphical representation. As each algorithm works differently and uses its own data, each requires its own debugger, see Figures 3 and 4.

2. Typeclasses

One important characteristic of GenI, from a software engineering perspective, is its complex type hierarchy. We need this hierarchy to reflect the large variety of data we process and the fact that this data have many components in common. For example, one way or another, the large majority of types are built from 'values': a (first-order) constant or variable.

To illustrate this, here is the representation of values and two data types, `FirstOrderTerm` and `GNode`, that are built over them:

```
data GeniVal = GConst String | GVar String

data FirstOrderTerm = (String, [GeniVal])

type AttValPair = (String, GeniVal)
data GNode   = GNode [AttValPair] [AttValPair]
```

In this section we show how Haskell typeclasses help us to quickly and concisely implement the tasks which are common to many of the types in our hierarchy.

2.1 Replaceable

During surface realisation, we frequently perform unification [16] between different instances of our data. Fortunately, the unification we require is not of the full structure-matching Prolog variety [9]. In our application, unification typically consists of unifying only a small part of the data and then propagating the unifier across the entire structure. Our first use of typeclasses was in capturing this notion of propagation. We do this with a class `Replaceable` which represents objects that contain `GeniVal`s that are local to that object. The object is `Replaceable` because it is possible that some of these values are replaced by other values, presumably during the propagation step after unification.

```
class Replaceable a where
 replace :: [(String,GeniVal)] -> a -> a
```

The first argument to `replace` is a list of substitutions to perform. Hence, `replace [("X", GVar "Y"), ("Z", GConst "bar")]` means that any instances of a variable named *X* should be replaced by the variable *Y*, and that any instances of variable *Z* should be replaced by the constant *bar*. We can express this straightforwardly in the implementation for `GeniVal` itself:

```
instance Replaceable GeniVal where
 replace sl v =
   let replaceOne (GVar x) (s1,s2) | x == s1 = s2
       replaceOne gx _ = gx
   in  foldl replaceOne v sl
```

Replacements on the types of our hierarchy is a relatively straightforward matter of calling `replace` on each of its relevant sub-components. Were we to insist upon factorising our code, we could also make use of polymorphic instances. Consider `AttValPair` and `FirstOrderTerm`, which we introduced above. Both are defined as a tuple `(String, v)` where `v` is either `GeniVal` or `[GeniVal]`. Their implementation can be thus described generically:

```
instance Replaceable v => Replaceable (String,v) where
 replace s (a,v) = (a, replace s v)
```

Two implementations for the price of one might not be stellar as an improvement, but let us next consider the implementation of `Replaceable [a]`:

```
instance Replaceable a => Replaceable [a] where
  replace s = map (replace s)
```

A single two-line implementation of `Replaceable [a]` allows us to capture at once replacements on lists of `GeniVal`, input semantics (a list of `FirstOrderTerm`), attribute-value pairs and lists of chart items.

2.2 Collectable

Another operation which we apply to many data-types is that of α-conversion, which consists of renaming variables in a term [4]. This allows us to avoid unification errors that come from treating variables in two different chart items as the same just because they have the same name. The first task in α-conversion is to determine what are the variables in the data-type. We do this with a class `Collectable`, whose function `collect` accumulates a set of variables:

```
class Collectable a where
  collect :: a -> Set.Set String -> Set.Set String
```

As before, implementing `Collectable` starts with the type `GeniVal` and can be automatically built up for the other types:

```
instance Collectable GeniVal where
 collect (GVar v) s = Set.insert v s
 collect _ s = s

instance Collectable v => Collectable (String, v) where
 collect (_,b) = collect b

instance Collectable a => Collectable [a] where
 collect l s = foldr collect s l
```

Implementing α-conversion is a simple matter of combining `Replaceable` and `Collectable`. To rename variables, we append a suffix to all open variables in the data type:

```
alphaConvert :: (Collectable a, Replaceable a)
             => String -> a -> a
alphaConvert suffix x =
  let vars  = Set.elems (collect x Set.empty)
      subst = map (\v -> (v, GVar (v ++ suffix))) vars
  in replace subst x
```

Typeclasses make it easy to express primitive operations out of which we build complex things like α-conversion. One enjoyable side-effect is that the primitive operations often wind up having a second life outside of their original intended use. `Replaceable` was meant to address our needs for a propagation step after unification, but it also turned out that we could drop it straight into α-conversion and build generic function where, as long as our data was both `Replaceable` and `Collectable`, we could also perform α-conversion on it. The `Collectable` typeclass was also reused in this way. It ended up being used in two separate optimisations for our surface realiser, an improved unification method and a technique for pruning the search space.

2.3 `Show`-like classes

A more basic concern in our surface realiser is being able to save data in a variety of formats. To save time parsing XML documents, we convert them into GenI's text format (doing so transforms a 16M file into a 1.4M one). To save even more time parsing our format, we are also experimenting with dumping the grammar into Haskell and compiling that into our generator (thus making our software a generator generator). Both tasks can be handled straightforwardly with typeclasses analogous to, and liberally "inspired" by, Haskell's `Show`.

```
class GeniShow a where
  geniShow :: a -> String
  geniShow x = geniShows x ""
  geniShows :: a -> ShowS

class HsShow a where
  hsShow :: a -> String
  hsShow x = hsShows x ""
  hsShows :: a -> ShowS
```

We will not show the implementation of both classes. Instead we will concentrate instead on `HsShow`, which dumps some data to Haskell. Note that this class looks very much like `Show` and in fact, is often just a wrapper to it; however, `Show` is not sufficient for our needs because its API documentation does not explicitly stipulate that the resulting `String` be syntactically correct Haskell [2]. We might just be misinterpreting the API, but in any case, it is useful for us to have a distinct typeclass for producing human-readable strings (`Show`) and one for producing GHC-readable ones (`HsShow`).

```
instance HsShow Char where hsShows = showChar

-- helper functions
parens, brackets :: ShowS -> ShowS
parens   s = showChar '(' . s . showChar ')'
brackets s = showChar '[' . s . showChar ']'

-- separators
unwordsByS :: ShowS -> [ShowS] -> ShowS
unwordsByS _   [] = id
unwordsByS sep ss = foldr1 (\s r -> s . sep . r) ss

uncommasS, unwordsS :: [ShowS] -> ShowS
uncommasS = unwordsByS (showChar ',')
unwordsS  = unwordsByS (showChar ' ')

-- lists and anonymous tuples
instance HsShow a => HsShow [a] where
 hsShows xs =
   brackets $ uncommasS $ map hsShows xs

instance HsShow a, HsShow b => HsShow (a,b) where
 hsShows (a,b) =
   parens $ uncommasS $ [ hsShows a, hsShows b ]

-- for algebraic data types
hsConstruct :: String -> [ShowS] -> ShowS
hsConstruct c args = parens $ unwordsS $ c:args
  parens $ showString c . showChar ' ' . unwordsS ss

instance HsShow GeniVal where
 hsShows (GConst xs) =
   hsConstruct "GConst" [hsShows xs]
 hsShows (GVar xs) =
   hsConstruct "GVar"   [hsShows xs]
```

The only things we really have to output are lists, anonymous tuples and algebraic data types. Lists and anonymous tuples are pretty straightforward. We `hsShows` their contents, separate everything by commas and wrap everything in brackets or parentheses respectively. Since the algebraic types are essentially tuples, things work more or less the same way for them. We output the constructor name as a `String`, apply `hsShows` on each of the arguments, separate everything by a space (not a comma), and wrap everything with parentheses. The resulting code may not look very pretty, (see Figure 2) but it is more or less readable, and it compiles. The important thing for us is that implementing instances of this for other types is a trivial task. For example, below is the implementation for `Ttree`, one of the data types in GenI. It looks virtually identical to the implementation of other types.

```
instance HsShowable Ttree where
 hsShows (TT a b c d e f g) =
```

```
hsConstruct "TT" [ hsShows a, hsShows b, hsShows c
                 , hsShows d, hsShows e, hsShows f
                 , hsShows g]
```

We do not claim that this is best way to go about the task. For example, there probably is a good way to automate the implementation of `hsShows` for types like the above, and also there are a few small problems with overlapping instances (section 2.4). But the fact that we could throw together a quick, dirty but workable solution in a few minutes is a testament to the usefulness of typeclasses.

2.4 Overlapping instances

One final remark about typeclasses: we found that Haskell98 typeclasses were not flexible enough for our needs. Consider `HsShow`, for example. Given an implementation of `HsShow Char` and `HsShow a => HsShow [a]`, we instantly have an implementation of `HsShow String`; however, this automatically derived instance gives output like `'h':'e':'l':'l':'o':[]`, where what we really would prefer to have is an implementation for `String` that produces the more readable `"hello"`. With normal Haskell98 typeclasses, we cannot just make an instance of `HsShow String` that does what we can because it would overlap with that of `HShow [a]`. Fortunately, the GHC extension for overlapping instances addresses exactly this problem.

2.5 GvizShow

To visualize data in our graphical debugger, we also have a typeclass for dumping data in Graphviz's dot format [1].

```
class GvizShow flag b where
  gvizShow :: flag -> b -> String -> String
```

The signature for this class is more complicated than that of the previous `Show`-like classes because

1. When combining sub-graphs into a single Graphviz document, we need to take care that the nodes of each sub-graph do not have the same names.
2. How an item is displayed depends on parameters specified through the graphical interface.

The first point is easy to deal with. We provide the `gvizShow` function with a `String` parameter that serves as a prefix to all node names of that sub-graph. When putting multiple sub-graphs into the same Graphviz dot file, we merely have to ensure that each sub-graph is printed with a different prefix.

Addressing the second point required us to use the GHC (Glasgow Haskell Compiler) extension which permits the use of multi-parameter type classes. Making the `flag` parameter polymorphic allows us extra flexibility in controlling the output of our debugger. For example, the debugger for the simple algorithm (figure 3) only provides a checkbox to show features or not. This translates into a `Bool` flag. On the other hand, the debugger for the CKY algorithm involves more complicated chart items (figure 4). The user has a drop down menu to select which derivation of the item to show, as well as checkboxes to show the full derivation, show the source tree and if showing the source tree, show the features of that tree. A `Bool` flag no longer suffices; we instead require a tuple `(Int, Bool, Bool, Bool)`.

3. Monads

3.1 Keeping a global `State`

Surface realisation requires a large amount of book-keeping to keep track of intermediary chart items and many other details. Furthermore each algorithm that we use requires its own brand of book-keeping and, as mentioned in the introduction, is associated with its own type of chart item. To cope with these, we use the `State` monad for book-keeping and parametrize it for each algorithm. Each algorithm `Foo` is associated with a large record `FooStatus` and all operations work under the `State FooStatus` monad. Here, for example, is an instantiation of this for the CKY algorithm:

```
data CkyStatus = CkyStatus -- details elided
data CkyItem   = CkyItem   -- details elided
type CkyState  = State CkyStatus

-- the particular rules used to combine chart items
-- used by the CKY algorithm
ckyRule1 :: CkyItem -> CkyState [CkyItem]
ckyRule2 :: CkyItem -> CkyState [CkyItem]
-- ..
ckyRuleN :: CkyItem -> CkyState [CkyItem]
```

Using the `State` monad allows us to cleanly separate the book-keeping from the bulk of the surface realisation work. This is especially useful when surface realisation involves a large number of different operations, because adding book-keeping to these operations simply consists of making them operate under the monad.

3.2 The `Maybe` and `List` monads

We found the seemingly modest `Maybe` and `List` monads to be a useful means of keeping our code concise. As is already well known, `Maybe` is particularly useful for representing chains of procedures that can fail. Two examples of this, in our case, come from the initial part of the surface realisation process, in which GenI selects grammar rules that correspond to the input semantics and does some post-processing on them. The post-processing involves many steps, and if one them fails, the entire post-processing fails. Ignoring the exact details, here is an example of a chain of post-processing steps written in `Maybe` monadic style:

```
unifyParams (l,e)
>>= unifyInterfaceUsing interface
>>= unifyInterfaceUsing filters
>>= enrich
```

As we can see, this chain of operations is not obscured by needless boilerplate `case` statements. It is readily apparant the code above is a sequence. Rearranging the sequence or inserting new elements into it is also very straightforward.

The `Maybe` monad is especially useful when the post-processing chains are arbitrarily long. For instance, one of the post-processing steps from above involves applying a list of modifications to a grammar rule. As before, should one of these modifications fail to apply, the entire procedure fails. The `foldM` function from the monad library allows us to express just that:

```
enrichBy :: Mod -> GramRule -> Maybe GramRule

enrich :: [Mod] -> GramRule -> Maybe GramRule
enrich mods g = foldM enrichBy g mods
```

Another well known monadic idiom is using `List` to represent non-deterministic computation. We have a different use for `List`, namely, to cleanly express a notion of "unpacking". In one of the algorithms implemented by GenI, chart items "pack" together the representation of several other chart items. The structure is recursive, as the packed items could also be packing items of their own. The expected advantage of this algorithm is a smaller chart with fewer items; however, it also makes things more complicated, because the chart items will have to be unpacked before they are returned as results. Since the packing is recursive, the chart items "multiply out". This is where the `List` monad comes in:

```
(TT [] "n0Vcs1" "Tn0Vcs1-191" [("arg0",(GVar "M")),("arg1",(GVar "G")),("arg2",(GVar "B1")),("label0",(GVar
"O")),("labelI",(GVar "H")),("rel",(GVa r "F1")),("sobjectI",(GVar "B1")),("sobjectL",(GVar
"C1")),("subjectI",(GVar "G")),("subjectL",(GVar "O")),("theta1",(GVar "G1")),("theta2",(GVar "H1")),("vbI"
,(GVar "M")),("vbL",(GVar "O"))] Auxiliar (Just [((GVar "O"),(GVar "F1"),[(GVar "M")]),((GVar "O"),(GVar
"G1"),[(GVar "M"),(GVar "G")]),((GVar "O"),(GVar "H1") ,[(GVar "M"),(GVar "B1")])]) (Node ((GN "n0"
[("cat",(GConst ["s"]))] [("cat",(GConst ["s"])),("idx",(GVar "A")),("inv",(GVar "B")),("label",(GVar
"C")),("mode ",(GConst ["ind","subj"])),("wh",(GConst ["-"]))] False [] Other False)) [(Node ((GN "n1"
[("case",(GConst ["nom"])),("cat",(GConst ["cl"])),("det",(GConst ["+ "])),("func",(GConst
["suj"])),("gen",(GVar "F")),("idx",(GVar "G")),("label",(GVar "H")),("num",(GVar "I")),("pers",(GVar "J"))]
[("cat",(GConst ["cl"]))] False [] Subs True)) []),(Node ((GN "n2" [("cat",(GConst ["vp"])),("gen",(GVar
"F")),("idx",(GVar "A")),("inv",(GVar "B")),("label",(GVar "C")),("mode",(GVar "D")),("neg-adv",(GConst
["-"])),("neg-nom",(GVar "W")),("num",(GVar "I")),("pers",(GVar "J"))] [("cat",(GConst ["vp"])),("gen",(GVar
"L")),("idx",(GVar "M")),("in v",(GVar "N")),("label",(GVar "O")),("mode",(GVar "P")),("neg-adv",(GVar
"Q")),("neg-nom",(GConst ["-"])),("num",(GVar "R")),("pers",(GVar "S")),("pp-gen",(GVa r "T")),("pp-num",(GVar
"U"))] False [] Other False)) [(Node ((GN "n3" [("cat",(GConst ["v"])),("gen",(GVar "L")),("idx",(GVar
"M")),("inv",(GVar "N")),("label ",(GVar "O")),("mode",(GVar "P")),("neg-adv",(GVar "Q")),("neg-nom",(GVar
"W")),("num",(GVar "R")),("pers",(GVar "S")),("pp-gen",(GVar "T")),("pp-num",(GVar "U "))]
[("aux-refl",(GConst ["-"])),("cat",(GConst ["v"])),("inv",(GConst ["-"]))] True [] Lex False)) [])]),(Node
((GN "n4" [("cat",(GConst ["s"]))] [("cat",(GC onst ["s"]))] False [] Other False)) [(Node ((GN "n5"
[("cat",(GConst ["c"]))] [("cat",(GConst ["c"]))] False [] Other False)) [(Node ((GN "n6" [("cat",(GConst
["que"]))] [("cat",(GConst ["que"]))] False ["que"] Lex True)) [])]),(Node ((GN "n7" [("cat",(GConst
["s"])),("idx",(GVar "B1")),("inv",(GConst ["-"])),("labe l",(GVar "C1")),("mode",(GConst
["ind","inf","subj"])),("princ",(GConst ["-"])),("wh",(GConst ["-"]))] [("cat",(GConst
["s"])),("cleft",(GConst ["-"])),("wh",( GConst ["-"]))] False [] Foot True)) [])])]))
```

Figure 2. Haskell code for a linguistic tree, generated by `hsShow`

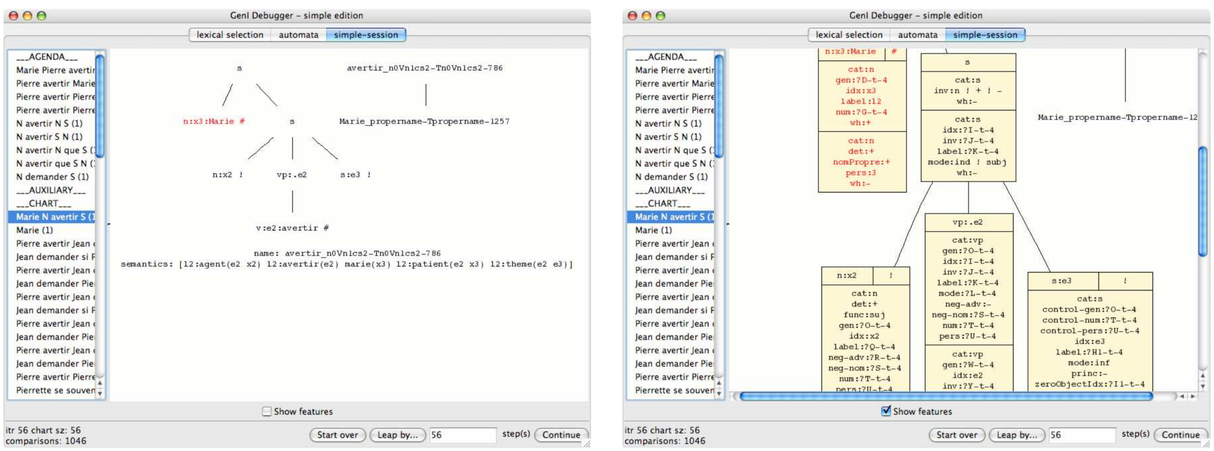

Figure 3. Debugger for simple algorithm

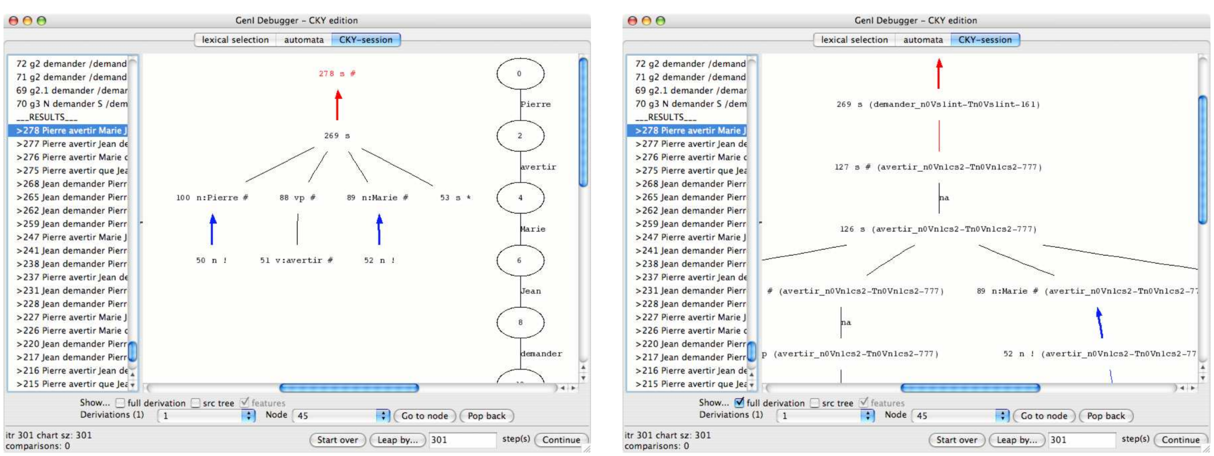

Figure 4. Debugger for CKY-based algorithm

```
unpack :: Tree [a] -> [Tree a]
unpack (Node px pks) =
  do x  <- px
     ks <- mapM unpack pks
     return (Node x ks)
```

The `List` monad helps to express the notion of unpacking in a clear and elegant manner. For comparison's sake, this is the equivalent code in non-monadic form, which is longer and harder to decipher:

```
unpack2 :: Tree [a] -> [Tree a]
unpack2 (Node px pks) =
 let next pk [] =
       map (\k -> [k]) (unpack2 pk)
     next pk ls =
       concatMap (\k -> map (k:) ls) (unpack2 pk)
 in case foldr next [] pks of
    [] -> map (\x -> Node x []) px
    ks -> concatMap (\x -> (map (Node x) ks)) px
```

While it might be possible to improve upon the non-monadic `unpack2`, we believe that it would be difficult to approach the simplicity of its monadic equivalent. This simplicity is especially important in the actual code used in GenI, because there, unpacking is combined with post-processing, and without the use of monads, things easily get out of hand. Keeping the unpacking process clear and concise was essential to understanding our own code.

3.3 The any monad idiom

As we mentioned in section 2.1, unification is a very frequently used operation in our application. It is also something that occurs in several different contexts. In most cases, we only want to know if unification fails or succeeds.

```
unify :: [GeniVal] -> [GeniVal]
 -> Maybe ([GeniVal], [(String,GeniVal)])
```

There are, however, variants to this theme. One such variant is the notion of inference rules, which produce a list of all possible results given a single chart item (or an empty list if the inference rule fails to apply). Another variant requires us to perform unification in a part of the surface-realisation process where failure is highly unusual and should be reported to the user because it might indicate that there are mistakes in the grammar provided as input. Adapting unification to these contexts consisted of returning our results in *any* monad instead of using `Maybe`. Where we would once return `Nothing`, we instead `fail` and identify the two conflicting `GeniVal` that caused unification to fail.

```
unify :: (Monad m) => [GeniVal] -> [GeniVal]
 -> m ([GeniVal], [(String,GeniVal)])
```

Doing this allows us to plug unification into the `Either String` error monad for our one-off operation with failures and also into the inference rule `List` environment. It also allows us extra flexibility in modifying our code. For instance, we might one day decide to create new monads via a `ListT` monad transformer for one reason or another. The fact that `unify` returns results in any monad means that it will continue to plug right in.

3.4 Monad transformers for modularity

Experimenting with surface realisation algorithms is a highly empirical process because their performance is mostly contingent on the grammar for the target language. Linguistic grammars are typically too large and complex to be able to predict how a new algorithm would behave. The Unix `time` command allows us to determine how long surface realisation tasks take, and the profiling tools provided in GHC tell us how much memory a surface realisation task consumes, but sometimes what we really want is more fine grained information: how many iterations does our algorithm run, how many chart items get produced, and so forth. To get a deeper understanding of how our software performs, we need some means of counting things. Furthermore we should preserve modularity by separating the counters from the main business of surface realisation, and at the same time allowing different algorithms to share the counting code.

Let us begin by tackling the issue of modularity. Consider the architecture proposed in section 3.1. Each algorithm `Foo` processes `FooItems` and does its book-keeping in a `FooStatus`, passing `FooStatus` around in a `State` monad.

```
-- version 1 no counter
type CkyState = State CkyStatus
ckyRule1 :: CkyItem -> CkyState [CkyItem]
```

Adding counting requires only a light modification. Rather than use `State` to pass `FooStatus` around, we use a `StateT` transformer with an embedded `State Int` monad to do the counting.

```
-- version 2 with counter
type CkyState a = StateT CkyStatus (State Int) a
ckyRule1 :: CkyItem -> CkyState [CkyItem]

incrCounter :: StateT st (State Int) ()
incrCounter = lift $ modify (+1)
```

The advantage of a `StateT` transformer is that it can be dropped directly in place of `State`. When we want to increment a counter, we simply invoke the `incrCounter` action defined above. Note also that since we use type synonyms, such as `CkyState` to hide the exact details of the transformer stack, adding the counters does not even require us to modify the type annotations of our inference rules. We merely change the type synonym, leaving the inference rules themselves none the wiser.

The modularity that comes from using `StateT` is crucial to us because it allows us to make the counting code more sophisticated. For example we would like to have multiple counters and the ability to easily add new ones whenever we need. Rather than using the `Int` to keep track of a single counter, we use a list of named counters as below:

```
-- version 3 with named counters
data Cnt = Cnt String Int
type CkyState a = StateT CkyStatus (State [Cnt]) a

incrCounter :: String -> StateT st (State [Cnt]) ()
incrCounter s = lift $ modify (map.helper)
 where helper (Cnt n v) | n == s = (Cnt n (v+1))
       helper c = c
```

When we want to increment a counter, we now provide its name, as in `incrCounter "iterations"`. No other modifications to our code are required.

Sharing this code among different surface realisation algorithms is a trivial consequence of the `FooStatus` architecture. For instance, here is how we add the counting feature to the "simple" algorithm:

```
type SimState a = StateT SimStatus (State [Cnt]) a
```

4. Higher order functions

Higher order functions are one of the key features in functional programming. Indeed, we have already seen their utility in our discussion of monads in section 3 and how they help us keep our code concise and modular. One variant on this theme is that higher order functions help us to break large and complicated functions into manageable chunks.

No matter the algorithm, GenI is in the business of producing chart items and whenever these chart items are produced, they need to be assigned somewhere so that the surface realisation algorithm knows what to do with them. We call the process of assigning new items to roles *dispatching*. Dispatching could be implemented as a simple cascade of `if then else` expressions, but this is highly impractical because:

1. Every new corner case would require an extension of the `if then else` cascade.
2. Some optimizations affect the dispatch process, but to find out if the optimizations are actually useful on real data, we need a convenient means of turning them on and off.
3. There are four different dispatch processes (one for each variant of our two surface realisation algorithms) which should share some code when possible.

GenI implements the dispatch process by breaking each juncture of the `if then else` cascade into a standalone function and using a higher order function to join these functions together. Dispatching can be seen as a sequence of filters. Each filter either traps a chart item and assigns it to a role; or lets the item through, potentially modifying it in the process. We capture the notion of a filter through use of the `Maybe` type. If an item is trapped, we return `Nothing`. If it passes through, we return `Just` the modified item.

```
type DispatchFilter s a = a -> s (Maybe a)
-- s is typically a monad like CkyState
```

To express the notion of sequencing dispatch filters, we created a dispatch filter combinator:

```
(>-->) :: (Monad s) => DispatchFilter s a
                    -> DispatchFilter s a
                    -> DispatchFilter s a
f1 >--> f2 = \x -> f1 x >>= maybe (return Nothing) f2
```

In addition to sequencing, we also needed a notion of choice, the dispatch filter equivalent of an if-then-else.

```
condFilter :: (Monad s) => (a -> Bool)
           -> DispatchFilter s a
           -> DispatchFilter s a
           -> DispatchFilter s a
condFilter cond f1 f2 = \x ->
  if cond x then f1 x else f2 x
```

Having sequence and choice combinators allows us to write the dispatch filters for each one of our algorithms in a clean manner with clear indications of what functionality is shared between them. Below are two examples of dispatch filters at work. The filters describe the dispatch process for two variants of the same algorithm. The actual details of the filters are not particularly important. What is interesting about them is that the relationship between filters is now clearly described and that some of the filters can be shared between dispatch processes. For example, both filter chains use a `dpRootCatFailure` filter.

```
dispatchSim1 :: DispatchFilter SimStatus SimItem
dispatchSim1 =
 condFilter isResult1
   (dpTbFailure >--> dpRootCatFailure >--> dpToResults)
   (dpTreeLimit >--> dpAux >--> dpToAgenda)

dispatchSim2 :: DispatchFilter SimStatus SimItem
dispatchSim2 =
 condFilter isResult2
   (dpTbFailure >--> dpRootCatFailure >--> dpToResults)
   (dpTreeLimit >--> maybeDpIaf >--> dpToAgenda)
```

5. Discussion

5.1 Lessons learned

We have illustrated how three features of Haskell (and more generally functional programming) have helped us develop the surface realiser GenI. Our experiences were that:

1. Typeclasses are an extremely useful means for implementing the same functionality over a complex hierarchy of types.
2. Monads are useful for much more than IO. The simpler monads like `Maybe` and `List` help us write code that is concise and easy to understand. More advanced monads, like the `StateT` transformer lend a very useful degree of modularity to the code.
3. Higher order functions are useful for breaking very large functions into bite-sized chunks.

5.2 Extra requirements

In this section, we explore the darker aspects of our Haskell experience by making note of the things which we either found to be difficult or which left us feeling somewhat uneasy. We cannot offer suggestions on how to solve these issues (we would have implemented them ourselves had we known what to do), but we hope that by identifying these difficulties we can call attention to things that would be troublesome for other ‘real-world’ Haskell users.

5.2.1 Profiling

Profiling is generally known by the Haskell community as a useful means for identifying time and space leaks in a program. GenI suffered from some rather severe space leaks. We found that the profiler was useful for identifying these leaks, but only in a limited way. The main problem was that we did not know how to use the large majority of the information that the GHC profiler gave us.

What we eventually settled on was using the global information provided by the profiler (total memory allocation and heap use over time) to get an impressionistic view of things: was our memory consumption getting worse or better, and by how much?

In the end, we implemented a dumb, iterative ‘trial and error’ profiling approach, mediated by a small test harness combining the profiler with the Unix `diff` utility and the revision control system `darcs`. This code profiling approach allowed us to get big improvements in performance. Indeed, we were able to reduce the memory allocation used to generate the sentence “John discusses with a nice excellent affectionate engineer” and its many variants down from 560M to 275M (mostly used when reading large input grammar files). Figure 5 shows the improvement in heap usage over this period of iterative optimization.

Even though profiling resulted in better performance, we found it to be very confusing. To start with, the only information we ended up using consisted of three numbers: time, total memory allocation and maximum heap usage. All other profiling information was discarded because we didn’t know how to interpret it.

We would have liked to use the profiling data to pinpoint the likely space-leaking parts of our code. But we ended up just guessing where the problem spots were and verifying our guesses empirically. Consider the heap graphs in Figure 5. Our first impression (the graph on the left) was that our `Btypes` module was the biggest source of memory consumption, so we tried making code in that module more strict. This had the desired effect of reducing both our memory consumption overall and the proportion consumed by that module (see the middle graph of Figure 5). But now the total memory consumption for `SimpleBuilder` has gone up. Because we could not interpret these kind of changes, we eventually gave up on using the detailed profiling information.

As usual, profiling had the side effect of forcing us to carefully study our code, and this led to the improvements that we finally

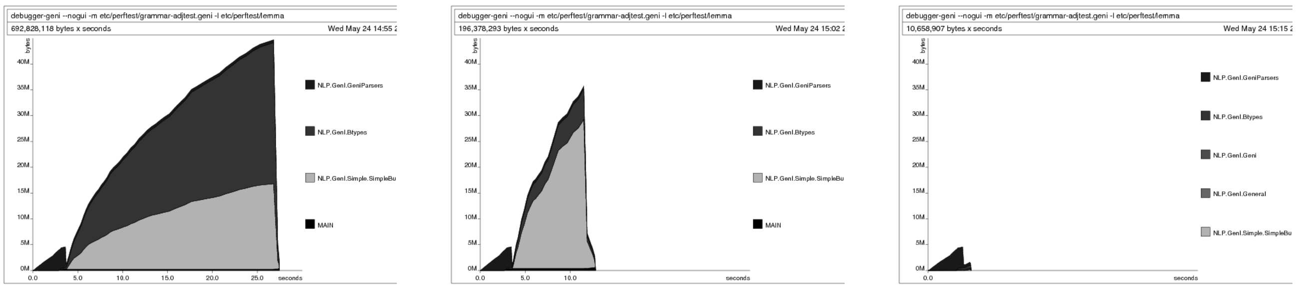

Figure 5. Heap usage over five days of profiling

obtained. But further help on narrowing down where the problems were would have been very welcome.

Most possibily, the problem is that we could not properly interpret the full output of the profiling tools, even though the information we were looking for is in there, somewhere. What would be especially helpful for the naive Haskeller is a rich guide presenting various profiling scenarios and best practices, as well as more documentation on how to make the most of the profiler output. Another thing we would look forward to is simply better profiling tools. For instance, the tool `ghcprof` for visualising call graphs is particularly interesting but cumbersome. Perhaps an updated, open source tool based on the Blobs diagram editor [20] could be a nice replacement. What might be even more useful, of course, is a full fledged profiling utility, whose user interface provides all the options with which to run a profilable Haskell program (lowering the learning curve in using the profiler) and which outputs call-graphs, heap profiling information, sortable cost center analyses all in the same user interface with a handy export-to-PDF feature and some means of saving the entire bundle for further perusal later on[2].

5.2.2 Timeouts

We adopt the asynchronous exceptions approach in [14] for timing GenI out when it has taken too long to find a result. The approach (figure 6) essentially consists of running a timer thread in parallel to the main action. If the main thread finishes before the timer thread, the latter is killed and all is well; however if the timer completes, a timeout exception is raised in the main thread.

We were able to use this mechanism to meet the essential goal of killing GenI off when its allotted time has elapsed. What we do not know how to do is to properly recover from a timeout, for example, by dumping out any statistical information we have collected, as well as any partial surface realisation results we may have found. One way to deal with this might be to extend the mechanism with an `on_timeout` action which gets executed when the timeout exception is raised. But we are only able to run `on_timeout` in the `IO` monad, whereas all our partial results and statistical information is in the `State` monad. There might be a way to extend our code, for example by lifting `IO` into our `State` monad, but that seems to be overkill for something as "simple" as a timeout.

[2] Yes, we know we might be asking for a lot, but wouldn't it be great?

5.3 Tools and libraries

Developing GenI in Haskell was easy not only because of the Haskell language itself or functional programming techniques in general, but also because of the availability of a few key libraries. We present here the libraries that we found to be useful, as well as the libraries that we are planning to use in the near future.

5.3.1 Libraries we used

QuickCheck The automatic unit-test generator QuickCheck [6] helped us develop fundamental operations such as unification on terms. Unification, in particular, was prone to subtle bugs both in the algorithm and the actual Haskell code. We use QuickCheck to verify properties about unification: namely that it should be symmetric (below) and reflexive; and that unification with anonymous variables should work correctly.

```
-- example QuickCheck for symmetry
prop_unify_sym :: [GeniVal] -> [GeniVal]
               -> Property
prop_unify_sym x y =
  let u1 = (unify x y) :: Maybe ([GeniVal],Subst)
      u2 = unify y x
      -- (all hasConst) makes it easier to
      -- compare the unifiers
      hasConst (GVar _, GVar _) = False
      hasConst _ = True
  in all hasConst (zip x y) ==> u1 == u2
```

This has helped us avoid bugs and called our attention to assumptions we had not realized we were making about our code. For instance, one detail we had simplified away in this article is that `GeniVal` can have other values than simple constants or variables ; they can represent anonymous variables or atomic disjunctions ("foo" ∨ "bar"). We implement atomic disjunction by modifying the `GConst` constructor for constants so that it accepts a `[String]` instead of simply `String`. By running QuickCheck, we realised that our unification does not verify the property of symmetry when this list is either empty or contains repeat elements. We eventually decided that this would never happen, and tightened our checks to filter these possibilities away; but we were nevertheless glad that this was brought to our attention.

```haskell
timeout :: Int -> IO a -> IO a
timeout secs action =
 do parent  <- myThreadId
    block $ do
     timeout <- forkIO (timeout_thread parent secs)
     Control.Exception.catchDyn
       (unblock $ do result <- action
                     killThread timeout
                     return result)
       (\exceptn ->
          case exceptn of
           TimeOut u | u == i ->
            return Nothing
           other ->
            killThread timeout >>= throwDyn exceptn)

timeout_thread parent secs =
 do sleep (secs * 1000000)
    id <- myThreadId
    throwTo parent (TimeOut id)
```

```haskell
timeout :: Int -> IO a -> IO a -> IO a
timeout secs on_timeout action =
 do parent  <- myThreadId
    block $ do
      timeout <- forkIO (timeout_thread parent secs)
      catch
        (unblock $ do result <- action
                      killThread timeout
                      return result)
        (\exceptn ->
           case exceptn of
            TimeOut u | u == i ->
             unblock on_timeout
            other ->
             killThread timeout >>= throwDyn exceptn)

timeout_thread parent secs =
 do sleep (secs * 1000000)
    id <- myThreadId
    throwTo parent (TimeOut id)
```

Figure 6. Left: Timeout mechanism from [14]. Right: The same, with a an `on_timeout` action.

HaXmL HaXmL is a set of tools for working with XML in Haskell [23]. The most useful of these was the DtdToHaskell converter, which translates a XML document type definition into a Haskell source file containing an hierarchy of types and the corresponding parsing code. This greatly simplified the task of parsing complicated XML data structures. For instance, invoking `readXml someXmlString` on a document containing

```xml
 <narg>
  <fs>
    <f><sym/></f>
    <f><sym/></f>
    <f/>
  </fs>
</narg>
```

gives us a much more reasonable Haskell structure

```haskell
Narg $ Fs $ [ F (Just Sym), F (Just Sym), F Nothing ]
```

The resulting simplification of our code is even more pronounced when the XML contains attributes.

Also, a useful side-effect of the DtdToHaskell tool was that it brought our attention to a number of bugs in the DTD. Most of these were errors of cardinality, errors where the DTD was being too permissive. For instance, we would sometimes say that an `f` node has arbitrarily many children, `sym*` when we really should have been saying that it has up to one child, `sym?`. Since the DTD was being overly permissive – all the documents we produced were validated – we never noticed the errors. However, when working with HaXmL and actually using the data types in our code, we would notice that something was amiss if the type `F` had a `[Sym]` instead of `Maybe Sym`.

HaXmL has saved us from both a large amount of repetitive XML processing, and errors in our DTD.

Parsec We originally implemented parsers for our various input files in Happy and Alex, but we eventually found this difficult to maintain. The main problem was that we had to separate the lexer, parser and the Haskell code which built data from them. Furthermore, type errors from the Haskell code embedded in our Happy files were difficult to deal with, because we only had compiler errors about the generated Haskell code and not the grammar from which they came.

Using the parser combinator library Parsec [13] allowed us to seamlessly integrate the data construction with parsing and lexing, and thus made our parsers much easier to maintain and extend. The trade-off, however, was that ambiguities in our grammar were more difficult to debug or even detect. We sometimes found ourselves liberally scattering the `try` keyword throughout our parser in a desperate push to just make things work.

WxHaskell Our graphical interface and debuggers were built with the help of WxHaskell [12], a Haskell wrapper to the cross-platform WxWidgets toolkit. Having a cross-platform (yet native) toolkit was useful because we aimed to obtain code executable in Linux, MacOS and Windows. Another advantage for us is that WxHaskell provides a layer of abstraction on top of the relatively low level widgets library. This allowed us to build and extend the user interface with relative ease.

5.3.2 Libraries we plan to use

We are primarily interested in libraries that help us write more efficient code, namely Edison for sequences and collections, HaLeX for automata, and ByteString for Strings. On the other hand, libraries for improving our graphical interface, such as Blobs are also welcome, as improving our debugger is a way of improving programmer and linguist productivity.

Edison We make frequent use of lists, sets and finite maps as implemented in the Haskell base libraries. The library Edison [15] contains many data structures that do the same job, but which may be better adapted to our various surface realisation tasks. Having typeclass interfaces to these data structures makes it easier to use them interchangeably, which is important to us because, as we claimed in section 3.4, optimising surface realisation algorithms is as much an empirical exercise as a theoretical one.

HaLeX Many of our algorithms use finite state automata. While we do have a naive implementation working, we would be interested in abandoning it in favour of a dedicated, efficient, third-party library. The HaLeX [17] library seems like a potential candidate for this job.

ByteString Continuing along the theme of efficient libraries, we notice that GenI uses an extremely large number of small `Strings`, and we wonder if using unboxed arrays could ultimately save space and time. We plan on investigating the recently released ByteString library [7]. Initial experiments have found that this actually degrades our performance with respect to the standard `Strings`, but there may be uses for the library yet.

Blobs The Blobs diagram editor [20] seems to be a promising candidate for replacing Graphviz as a visualiser. We hope that by using Blobs we will be able to simplify the installation procedure for GenI (by replacing the dependency on some third party-application with a Haskell module whose installation would presumably be simplified by something like cabal-get [3] in the future). More importantly, we would like to have a more natural interaction with the debugger, one which allows the user, for instance, to click on a node of a tree and display the features for just that node.

5.3.3 Libraries we might contribute

We believe that GenI is well on its way to becoming a generic platform for building not only surface realisers but natural language parsers, which do almost the same thing as surface realisers except in reverse. Here are two libraries that could be spun off the codebase:

Inference rule framework A surface realiser could be written as a list of inference rules which produce chart items. We believe that our code can eventually be generalised into an framework that executes inference rules as higher order functions and dispatches the produced chart items using the filters described in section 4. A similar framework has already been proposed in [19], along with a Prolog implementation. Our version would be useful for building natural language parsers or surface realisers in native Haskell.

Debugger widgets We make frequent use of widgets where the user selects an item from a list and that item is displayed as a graph in the main pane (figures 3 and 4). To keep our debuggers for separate algorithms factorised, we have also created a general debugger widget which handles basic notions like stepping and continuing. These could be used outside of the surface realisation task in any application that can be broken down into discrete steps.

The GenI source code is available via `darcs` and is released under the terms of the GPL. For more information, see the GenI homepage at `http://trac.loria.fr/~geni`.

Acknowledgments

Thanks to

- Sébastien Hinderer and Émilie Balland for general comments on the article, and also to Daniel Gorin for Haskell expertise and comments, part of which lives in the GenI timeout and statistical profiling mechanism.
- Carlos Areces, the original author of GenI, for his guidance on this paper.
- The reviewers of this article for their many useful and detailed suggestions.

References

[1] The DOT language. `http://www.graphviz.org/doc/info/lang.html`.

[2] Text.Show API documentation.

[3] CabalGet. `http://www.haskell.org/hawiki/CabalGet`, 2005.

[4] H. P. Barendregt. *The Lambda Calculus (Studies in Logic and the Foundations of Mathematics)*. North Holland, October 1984.

[5] J. Carroll and S. Oepen. High efficiency realization for a wide-coverage unification grammar. In *Proceedings of the Second International Joint Conference on Natural Language Processing (IJCNLP05)*, 2005.

[6] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[7] D. Roundy et al. Data.ByteString / FPS. `http://www.cse.unsw.edu.au/~dons/fps.html`, 2006.

[8] M. Kay. Chart Generation. In *34th ACL*, pages 200–204, Santa Cruz, California, 1996.

[9] K. Knight. Unification: a multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, March 1989.

[10] A. Koller and K. Striegnitz. Generation as dependency parsing. In *Proceedings of the 40th ACL*, Philadelphia, 2002.

[11] E. Kow. Adapting polarised disambiguation to surface realisation. In *17th European Summer School in Logic, Language and Information - ESSLLI'05, Edinburgh, UK*, Aug 2005.

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

[13] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[14] S. Marlow, S. Jones, and A. Moran. Asynchronous exceptions in Haskell. 2000.

[15] C. Okasaki. An overview of Edison. In *ICFP 2000 (Haskell Workshop)*, 2000.

[16] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[17] J. Saraiva. Halex: A haskell library to model, manipulate and animate regular languages. In *Proceedings of the ACM Workshop on Functional and Declarative Programming in Education (FDPE/PLI'02)*, 2002.

[18] Y. Schabes and A. K. Joshi. An Earley-type parsing algorithm for Tree Adjoining Grammars. In *Meeting of the Association for Computational Linguistics*, pages 258–269, 1988.

[19] Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1&2):3–36, 1995.

[20] A. van IJzendoorn et al. Blobs diagram editor. `http://www.cs.york.ac.uk/fp/darcs/Blobs`.

[21] K. Vijay-Shankar and Aravind K. Joshi. Some computational properties of tree adjoining grammars. In *Proceedings of the 23rd annual meeting on Association for Computational Linguistics*, pages 82–93, Morristown, NJ, USA, 1985. Association for Computational Linguistics.

[22] K. Vijay-Shanker and A. Joshi. Feature based tags. In *Proceedings of the 12th ACL*, pages 573–577, Budapest, 1988.

[23] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34–9, pages 148–159, N.Y., 27–29 1999. ACM Press.