Functional programming and NLP

Peter Ljunglöf

January 1, 2002

Abstract

Most of today's NLP software is developed using either a logic programming language such as Prolog, or a low-level imperative language such as C or C++. In this paper I will try to argue why the paradigm of functional programming (as opposed to logic and imperative programming) matters for natural language processing.

One reason for why functional programming hasn't caught on among computational linguists could be that the functions are often explained from a mathematical perspective, and that can scare away non-mathematicians such as linguists.

Another reason could be that linguists often learn predicate logic as the basis for NL semantics, and logic programming is closer to predicate logic than other programming paradigms.

In this paper I will try to argue that functional programming is a good paradigm for many natural lanugage tasks. In a way it is a version of John Hughes' influencing paper on why functional programming matters [3], but specialized for natural language processing.

1 Elements of functional programming languages

In a real functional programming language functions are first-class objects. This is the main ingredient, and gives rise to higher-order functions and a very modular way of programming.

But programming with higher-order functions is very error-prone, and therefore most modern functional languages (such as ML and Haskell) has a static polymorphic type system. Indeed many programming languages have type systems, but it's almost only functional languages that have polymorphic types – i.e. that a function can be applied to arguments of many different types.

A functional programming language can be strict or lazy. A strict language (such as ML) evaluates the arguments of the functions before the function is applied, but a lazy language (such as Haskell) doesn't evaluate until the value is really needed. Laziness can lead to great efficiency improvements, and it is even possible to write perfectly functioning programs that will not terminate under strict evaluation.

1.1 A minor introduction to Haskell

In this paper I will use Haskell syntax. Function application is written $f\ a$ and not $f(a)$; and application with two arguments is written $f\ a\ b$ and not $f(a, b)$. To improve readability Haskell has operators like $(*)$, $(+)$ and $(:)$, and they are just infix functions on two arguments; i.e. $3 + 4$ is an application of $(+)$ to the arguments 3 and 4. An operator can be used as an ordinary function by putting it inside parentheses; so $(+)\ 3\ 4$ is the same as $3 + 4$.

Parentheses are used to disambiguate and change precedence; so $(3+4)*5$ is not the same as $3+(4*5)$, and $f\ (g\ a)\ b$, $f\ (g\ a\ b)$ and $f\ g\ a\ b$ are three different things – the first is an application of the function f to the two arguments $g\ a$ and b; the second is f applied to the single argument $g\ a\ b$; and the third is f applied to the three arguments g, a and b.

Data structures are build by *constructors*, which are functions (or constants) which are not evaluated. A list is either the empty list, $[\,]$; or the compound list $x : xs$, where x is the head element and xs is the tail list.

Assume now that we want to write a function to square each element in a list of numbers. We can do this by *pattern matching* on the argument list.

$$\begin{array}{lcl} \mathit{squares}\ [\,] & = & [\,] \\ \mathit{squares}\ (n : ns) & = & (n * n) : \mathit{squares}\ ns \end{array}$$

And here is an example run in the Haskell interpreter Hugs.[1]

```
> squares [1,2,3,4,5]
[1,4,9,16,25]
```

To translate a list of characters (also called a string) into lower case, we can define the following function.

$$\begin{array}{lcl} \mathit{lowercase}\ [\,] & = & [\,] \\ \mathit{lowercase}\ (c : cs) & = & \mathit{toLower}\ c : \mathit{lowercase}\ cs \end{array}$$

The function *toLower* is built-in and translates a character to lower case. And an example run.[2]

```
> lowercase "Pelle and Lisa"
"pelle and lisa"
```

1.2 Functions as first-class objects

A functional programming language is of course built on functions. In fact, functions are the only way to compute things in a functional language. But what's the issue with functions? It is possible to define functions in an imperative language like C. The main difference is that in a real functional language, the functions are first-class objects, which means that they can be given as arguments to other functions, and be returned as results.

[1] As in most languages, the term $[1, 2, 3, 4, 5]$ is just syntactic sugar for the list $1{:}2{:}3{:}4{:}5{:}[\,]$.

[2] The string `"hej"` is just a shorthand for the list of characters `['h', 'e', 'j']`.

Higher-order functions

The example functions defined earlier are instances of a very general transformation on lists – to transform each element separately. This can be implemented as a function that takes a function as argument.

$$\begin{array}{lcl} map\ f\ [\,] & = & [\,] \\ map\ f\ (x : xs) & = & f\ x : map\ f\ xs \end{array}$$

The *map* function takes as argument a function on the elements of a list and returns a function on lists. We can use this function to define the previous functions.

$$\begin{array}{lcl} square\ x & = & x * x \\ squares & = & map\ square \\ lowercase & = & map\ toLower \end{array}$$

A function on two arguments can also be seen as a function that, given an argument, returns a function on one argument. This means that $f\ x\ y$ is the same as $(f\ x)\ y$, i.e. function application associates to the left.

Lambda abstraction

Another concept that follows with functions as first-class objects is lambda abstraction. We can define anonymous functions by abstracting over the arguments. With this feature we can define the *squares* function without any auxiliary definition *square*

$$squares \quad = \quad map\ (\lambda x \rightarrow x * x)$$

We can also define functions local to another function – i.e. give a local name to an anonymuos function.

$$\begin{array}{lcl} squares & = & map\ square \\ \quad \textbf{where}\ square\ x & = & x * x \end{array}$$

1.3 Static typing

Since $f\ (g\ a)\ b$, $f\ (g\ a\ b)$ and $f\ g\ a\ b$ are three completely different things, programming with functions could be very error-prone. This is the main reason why most functional programming languages are typed, as opposed to e.g. Prolog.

But to make use of the possibilities of higher-order functions, the type system is more complex than the type system of C and Pascal. In functional programming we can have *polymorphic* type, which means that a function works with many different types. E.g. the *map* function has a very polymorphic type:

$$map \quad :: \quad (\alpha \rightarrow \beta) \rightarrow [\,\alpha\,] \rightarrow [\,\beta\,]$$

α and β are called type variables, and what the type says is that it can be applied to a function and a list, if the function can be applied to the elements of the list. The resulting list has elements of the function's argument type. Our example functions then have the following types:

$$\begin{array}{lcl} \mathit{lowercase} & :: & [\,\mathsf{Char}\,] \to [\,\mathsf{Char}\,] \\ \mathit{squares} & :: & [\,\mathsf{Integer}\,] \to [\,\mathsf{Integer}\,] \end{array}$$

In Haskell one can introduce new data types with the **data** declaration and tpye synonyms (abbreviations for existing types) with the **type** declaration. As an example, the String type is an abbreviation of lists of characters.

$$\textbf{type}\ \mathsf{String} \quad = \quad [\,\mathsf{Char}\,]$$

The types can be abstracted over one or several other types, such as in the list type (there can be lists of characters or lists of integers etc). As another example we can introduce general trees abstracted over the internal nodes and the leaves.

$$\begin{array}{lcl} \textbf{data}\ \mathsf{Tree}\ \alpha\ \beta & = & \mathsf{Node}\ \alpha\ [\,\mathsf{Tree}\ \alpha\ \beta\,] \\ & | & \mathsf{Leaf}\ \beta \end{array}$$

1.4 Laziness

The third component of some functional languages is laziness. This means that function applications are evaluated on demand only.

One effect of this is that we can work with infinite objects; such as the list of all positive integers.

$$\begin{array}{lcl} \mathit{posints} & = & 1 : \mathit{map}\ \mathit{next}\ \mathit{posints} \\ \quad \textbf{where}\ \mathit{next}\ n & = & n + 1 \end{array}$$

This is an infinite list of constantly increasing integers. But this leads to no problem, as long as we only work with finite subparts of that list.

```
> take 20 posints
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

In a strict language such as ML or Prolog, this call would never terminate. Another feature of lazy evaluation (or call-by-need as it is sometimes called) is that it evaluates a given constant only once, even if it occurs in several places. Because of this we will not end up doing the same computation several times.

An effect of lazy evaluation is that side-effects need some thought to introduce – we cannot put a call to *print* (which prints a value to the screen) anywhere in a program, since we cannot be sure when that part of the program will be evaluated, in fact it may never be evaluated. Haskell uses a concept from category theory called monads to incorporate side-effects, but this is not the topic of this paper, so I leave it with this.

2 Case study: Parser combinators

Parser combinators are an example of an domain specific embedded language. It is an language for describing parsers, or context-free grammars, embedded in the host language Haskell. It's domain specific because it regards only the domain of parsers or grammars.

A parser is (in our setting) a type abstracted over the type of terminals and the type of parse results.

$$\textbf{type}\ \mathsf{Parser}\ s\ \alpha \quad = \quad \ldots$$

We leave the definition of the type abstract, and leave it to the interested reader to read more about different parser types in e.g. Wadler's classical paper [7], or some of the later writings [4, 5, 6]. Now we can introduce a small set of combinators, with which we can describe any context-free grammar. So we need combinators for sequencing, alternation and recognizing a sequence of terminal symbols.

$$\begin{array}{lcl}
(<\star>) & :: & \mathsf{Parser}\ s\ (\alpha \to \beta) \to \mathsf{Parser}\ s\ \alpha \to \mathsf{Parser}\ s\ \beta \\
(<|>) & :: & \mathsf{Parser}\ s\ \alpha \to \mathsf{Parser}\ s\ \alpha \to \mathsf{Parser}\ s\ \alpha \\
(?>) & :: & [\,s\,] \to \alpha \to \mathsf{Parser}\ s\ \alpha \\
\mathit{zero} & :: & \mathsf{Parser}\ s\ \alpha
\end{array}$$

With these four combinators we can describe any context-free grammar. $(<\star>)$ takes two parsers and combines them in sequence. The result of the first parser has to be a function that can be applied to the result of the second parser, and the result of the combined parser is the application of the function to the argument. $(<|>)$ combines two parsers in parallel, i.e. introduces a choice in the grammar. $ss\ ?>\ a$ is a parser that recognizes the sequence of terminal symbols ss, returning the result a. The parser *zero* is there mostly for completeness, it is a parser that always fails. It is a left and right unit for $(<|>)$, and since alternation is associative they form a monoid together.

Grammar writing can be much simplified by defining auxiliary combinators. One important parser is *succeed* a, which recognizes the empty sequence, and has a as parse result.

$$\begin{array}{lcl}
\mathit{succeed} & :: & \alpha \to \mathsf{Parser}\ s\ \alpha \\
\mathit{succeed}\ a & = & [\,]\ ?>\ a
\end{array}$$

Another one is the combinator $(<\$>)$, which takes a function and a parser, and gives a parser which recognizes the same language as the given one, applying the function on each result.[3]

$$\begin{array}{lcl}
(<\$>) & :: & (\alpha \to \beta) \to \mathsf{Parser}\ s\ \alpha \to \mathsf{Parser}\ s\ \beta \\
f\ <\$>\ p & = & \mathit{succeed}\ f\ <\star>\ p
\end{array}$$

[3]The alert reader may notice the similarity in the type signature with the *map* function on lists. And indeed it is possible in Haskell to introduce a very polymorphic map function which can be applied to both parsers and lists (and other types). This is done via the type class system, which will not be discussed in this paper.

(<⋆>), (<$>) and (<+>) all associate to the left, which means that f <$> p <⋆> q <⋆> r is equivalent to ((f <$> p) <⋆> q) <⋆> r. Also (<⋆>), (?>) and (<$>) bind harder than (<+>), which means that p <⋆> q <+> f <$> r is the same as (p <⋆> q) <+> (f <$> r).

2.1 Montague semantics

Already in [1], Frost and Launchbury showed how to implement a simple Montague-style grammar in a functional language. In this section we show how to do this in a simple manner.

First-order logic

First we need a data type for first order formulas. We skip the disjunction and the negation, since we will have no use of these in our small language fragment.

```
data Formula = Pred Term [Term]
             | And Formula Formula
             | Implies Formula Formula
             | Exists (Term → Formula)
             | Forall (Term → Formula)
```

The Pred constructor creates an atomic formula, a constant, a predicate or a relation. The first argument is the name of the predicate and the second is a list of its arguments. The quantifiers use a function from terms to formulas to be able to make use of the built-in variable binding in a functional language. The type of terms is simply strings.

```
type Term = String
```

To simplify things we introduce the type of predicates as a function from terms to formulas.

```
type Predicate = Term → Formula
```

We also need a function to create a predicate from a term representing the name of the predicate.

```
predicate   :: Term → Predicate
predicate p = λx → Pred p [x]
```

Furthermore we lift the binary connectives to work on the level of predicates instead of just formulas.

```
andP          :: Predicate → Predicate → Predicate
andP p q      = λx → And (p x) (q x)
impliesP      :: Predicate → Predicate → Predicate
impliesP p q  = λx → Implies (p x) (q x)
```

In the same manner we introduce the type of binary relations as a function from two terms to a formula, and the *relation* function to create a relation from its name.

```
type Relation   =  Term → Term → Formula
relation        ::  Term → Relation
relation r      =  λx y → Pred r [x, y]
```

A Montague-style parser

Now we are ready to define a grammar. First we introduce the type of our Montague-style parser, which is a parser that recognizes terms.

```
type MonParser α  =  Parser Term α
```

A sentence is a parser returning a first order formula. The result of the noun phrase is applied to the result of the verb phrase.

```
sentence  ::  MonParser Formula
sentence  =   noun_phrase <⋆> verb_phrase
```

A verb phrase is a parser returning a predicate. It is either an intransitive verb or a transitive verb, followed by a noun phrase.

```
verb_phrase     ::     MonParser Predicate
verb_phrase     =      intrans_verb
                <|>    flip (·) <$> trans_verb <⋆> noun_phrase
```

To compose the transitive verb with the noun phrase we apply *flip* $(\cdot)$, which are built-in Haskell functions. *flip* takes a function on two arguments and flips them around, and $(\cdot)$ is standard function composition. This means that *flip* $(\cdot)$ takes two functions as arguments and composes the second with the first.

```
intrans_verb  ::  MonParser Predicate
intrans_verb  =   predicate <$> terminal ["arrives", "departs"]
```

An intransitive verb is a predicate, and here we make use of a helper function *terminal*, which takes a list of terms and recognizes any of them, returning the recognized term as result.

```
terminal          ::     [Term] → MonParser Term
terminal []       =      zero
terminal (t : ts) =      words t ?> t
                  <|>    terminal ts
```

The *words* function is a built-in function for tokenizing a string into a list of words. This is useful for multi-word terms such as the transitive verbs "leaves for" and "flies to".

```
trans_verb  ::  MonParser Relation
trans_verb  =   relation <$> terminal ["serves", "leaves␣for", "flies␣to"]
```

Since a verb phrase is a predicate, a noun phrase has to be a function taking a predicate to a formula. It is either a person-name or a quantifier followed by (and applied to) a noun.

```
noun_phrase  ::  MonParser (Predicate → Formula)
noun_phrase  =   person_name
             <+> quantifier <*> nbar
```

A person-name simply takes a predicate and applies it to a given term. The ($) function is standard function application, and thus *flip* ($) takes first the argument and then the function to be applied.

```
person_name  ::  MonParser (Predicate → Formula)
person_name  =   flip ($) <$> terminal ["sas", "boston", "europe", "flight␣714", "lunch"]
```

A noun is a predicate, and we can follow a noun with a prepositional phrase, or have an adjective in front. Both the prep phrase and the adjective are also predicates, so we can just apply the lifted conjunction *andP*.

```
nbar  ::  MonParser Predicate
nbar  =   noun
      <+> andP <$> noun <*> prep_phrase
      <+> andP <$> adjective <*> nbar
```

Nouns and adjectives have the obvious definitions.

```
noun       ::  MonParser Predicate
noun       =   predicate <$> terminal ["company", "city", "flight"]
adjective  ::  MonParser Predicate
adjective  =   predicate <$> terminal ["large", "green", "hijacked"]
```

A preposition is a relation, which means that if we follow it by a noun phrase we have the same structure as for a transitive verb; so we use the flipped function composition.

```
prep_phrase  ::  MonParser Predicate
prep_phrase  =   flip (·) <$> prep <*> noun_phrase
prep         ::  MonParser Relation
prep         =   relation <$> terminal ["in", "at", "with"]
```

A quantifier finally is a function over two predicates, and therefore we can make use of the lifted connectives.

```
quantifier  ::  MonParser (Predicate → Predicate → Formula)
quantifier  =   ["every"] ?> (λp q → Forall (impliesP p q))
            <+> ["some"] ?> (λp q → Exists (andP p q))
```

Example run

Now assume that we have a *parse* function, which takes a parser and a list of terms, and prints out the results in a nice way, we can show an example run here. (Recall that the *words* function tokenizes a string).

```
> parse sentence (words "sas serves lunch")
serves(sas,lunch)

> parse sentence (words "sas serves every city")
forall x (city(x) => serves(sas,x))

> parse sentence (words "sas serves every city in europe")
forall x (city(x) & in(x,europe) => serves(sas,x))

> parse sentence (words "some company serves every city in europe")
exists x (company(x) & forall y (city(y) & in(y,europe) => serves(x,y)))
```

3 Discussion

In our case study we have showed that at least some parts of natural language processing fit nicely in a functional framework. We have made use of all three ingredients of a lazy functional programming language that was mentioned in sections 1.2 – 1.4.

Functions as first-class objects

A nice abstraction was to introduce the types of predicates and relations, which are functions from terms to formulas. The lifted functions *andP* and *impliesP* then became higher-order functions, taking functions as arguments. But we can use them as any other function. The *flip* and the $(\cdot)$ functions are standard higher-order functions. Also the abstract type of parsers has to be higher-order, since the results can be of any type, including functions.

Static typing

Since the example grammar is typed, the compiler will not let us combine phrases in the wrong way. If we try to apply just the $(\cdot)$ function and not *flip* $(\cdot)$ in the prepositional phrase, the type-checker will report a type error there. In an untyped language we wouldn't notice that we made an error until we really tried to parse a prep phrase. And for a large grammar this can be important – it can take quite a while until some type error in some small subpart of the grammar is noticed.

Laziness

Laziness is not used in a directly obvious way, except that we know that only the work that is needed will be done. If we only want to know if something is a sentence, the parse result will not be calculated. And if we want to know whether a sentence is ambiguous or not, there is no need for more than two interpretations.

Also, some implementations of the parser type needs a lazy programming language to not fall into infinite recursion.

3.1 Other examples

Of course there are other examples of NLP where functional programming is a good tool, and here is a short list.

- Functional morphology. Higher-order functions, statis typing and laziness can be a good basic framework for writing morphological specifications for different languages. One simply tries to find a good set of combinators in the same manner as above, to describe certain morphological data in a language.
- Morphological and syntactic tagging. One can define combinators with which one can describe very general rule templates, and then learn rules from a corpus.
- Parsing and semantics. The example before is of course simple, but also more elaborate grammars and semantics can be defined. Parsing algorithms often get very short and simple when writing them functionally.

Functional programming lets us separate things in a clear way – we can define combinators for what we want to do, and not be diverted by the underlying implementation. In this way we embed a small special-purpose language in a general-purpose language – we don't have to write compilers etcetera for the small language. This is called "domain specific embedded languages" [2].

References

[1] R. Frost and John Launchbury. Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32(2):108–121, 1989.

[2] Paul Hudak. Modular domain specific languages and tools. In *Fifth International Conference on Software Reuse*, pages 134–142, Victoria, Canada, June 1998.

[3] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[4] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323– 343, July 1992.

[5] Peter Ljunglöf. *Functional parsing*. Licenciate thesis, Gothenburg University, Gothenburg, Sweden, 2002. To appear.

[6] Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS*, pages 1–17. Springer-Verlag, 1996.

[7] Philip Wadler. How to replace failure by a list of successes. In *Second Int. Conf. on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. Springer-Verlag.