PFTL: A Systematic Approach For Describing Filesystem Tree Processors

Nuno Ramos Carvalho[1], Alberto Manuel Simões[1], José João Almeida[1], Pedro Rangel Henriques[1], and Maria João Varanda[2]

[1] University of Minho
Braga, Portugal
{narcarvalho,ambs,jj,prh}@di.uminho.pt
[2] Polytechnic Institute of Bragança
Bragança, Portugal
mjoao@ipb.pt

Abstract. Today, most developers prefer to store information in databases. But plain filesystems were used for years, and are still used, to store information, commonly in files of heterogeneous formats that are organized in directory trees. This approach is a very flexible and natural way to create hierarchical organized structures of documents.

We can devise a formal notation to describe a filesystem tree structure, similar to a grammar, assuming that filenames can be considered terminal symbols, and directory names non-terminal symbols. This specification would allow to derive correct language sentences (combination of terminal symbols) and to associate semantic actions, that can produce arbitrary side effects, to each valid sentence, just as we do in common parser generation tools. These specifications can be used to systematically process files in directory trees, and the final result depends on the semantic actions associated with each production rule.

In this paper we revamped an old idea of using a domain specific language to implement these specifications similar to context free grammars. And introduce some examples of applications that can be built using this approach.

1 Introduction

A directory tree (hierarchical organized), with all sort of heterogeneous documents, is a common artifact. It can be found in any computer (being it an high-performance server or a small smart-phone). These artifacts are very common because they have a very simple formal definition, usually without types (just two, folders and files), which means that at any time a document can be placed almost anywhere. Also, they are inexpensive to build, hardware-wise (most of the computers already have a filesystem available) and software-wise (no particular skill or software is required, it is only needed to create directories and include files in them). A directory tree contains an high amount of rich information: not only the content of each file but also the information inherent to the tree structure.

Many problems can be solved processing filesystems trees. Generally these problems belong to one of the following families:

- extracting information, from one specific file, or from a set of files;
- bind, aggregate, concatenate information already available in the tree;
- decorate or enrich the filesystem tree with new information (usually stored in new files).

Common to all of these families of problems is a set of challenges and tasks that need to be overcome when implementing specific tools. Solving these issues every time that we need to process filesystem trees can be hard work and time consuming.

There are tools that help the systematic traversal of file systems (to just mention one, the Perl programming language has a module named `File::Find` that does just that: runs a call-back function for each folder or file found), but they just ease the process of stepping inside folders, and listing the files that need to be processed.

Unfortunately these tools do not help the programmer handling the files or directory structure semantics. In order to have some extra information about the structure being processed a new domain specific language (DSL), named PFTL[3], was designed. Its main goal is to do what other tools already do (process directory trees structure, a file at a time), but also make the processing code aware of the full directory structure, by allowing the description of the directory tree structure in an elegant and formal way. This work is based upon a previous prototype used to automatically build an entire website from the contents stored in plain files [2, 7].

DSL [5, 4, 8, 6] are tailored to specific application domain and offer users more appropriate notations and abstractions. Usually DSL are more expressive and are easier to use than general purpose languages (GPL) for the domain in question, with gains in productivity and maintenance costs.

Our main objective is to use a simple and practical approach to process documents organized in a hierarchical way. The documents are stored in files and the organization of the files is achieved by using a tree of directories. The formalization of these descriptions allows to associate semantic actions to each sentence (filesystem tree definition) producing any desired arbitrary result.

In the process of defining a formal language to specify directory structures and actions to trigger, we obtained a language resembling a common grammar, as defined by Bison or yacc: files are terminal symbols and folders are non terminal symbols.

There are major advantages of using a grammar-based approach [1, 9] in opposition to a typical recursive transversal:

- Semantic actions can return values up in the parsing tree for future use; once a semantic action is executed and produces a result, this result is returned as the value of the non-terminal symbol so, it can be used later in the parent rule.

  When making the parallelism with the filesystem, one can process a file, and return the result up to its containing directory. There, that result can be composed with results from processing other files or subdirectories. And this composed result can be returned for that folder parent.

---

[3] To baptism a child is a big responsibility. At the time of writing we did not have a real name for the language. PFTL is just an acronym to Process Filesystem Tree Language. It might change in the future as vowels are missing to make it pronounceable.

This functionality is clearly illustrated in the *fs2latex* example described in the examples in section 6. It uses semantic actions to produce simple LATEX code snippets that are composed together automatically into a complete LATEX document.

– Since our grammar-like description is based on symbols, that represent file types, not files names, for every file and directory processed its type needs to be determined. To perform this task we implemented a special tool that can ascertain types of files or directories. We call this particular program the *TypeOf Oracle* [4].

This means that, when processing files, there will be information about which type of file is being processed. This feature is illustrated in the CROSS project example (section 6.3), where specialized inspectors are chosen to process individual files based on their types.

In the next section of this paper the PFTL language will be introduced. It will be explained how it can be used to describe a directory tree structure, with associated semantic actions. In section 4, we discuss how a compiler that is able to process these descriptions and to produce any desired result was implemented. In section 5, the *TypeOf Oracle* will be described. Section 6 promotes the use of this witchcraft by presenting some real applications that were conjured using this approach in a clean and elegant way in a very short time span. We conclude with some comments on the obtained results and forecast some future work.

2   Related Work

Many frameworks already provide mechanisms to transverse directory trees and perform some kind of arbitrary task. Just to illustrate some examples:

– `File::Find` is a module written in Perl that allows the transverse of a directory tree while provided a user defined function to process each element of the tree (file or directory).
– `SimpleFileVisitor` is a class written in Java that provides more or less the same functionality, give perform some arbitrary task recursively for some directory tree.

Many more examples can be found of similar tools, but they all share the same philosophy, very abstractly: given a function f, and a starting path p, apply f to all files (and/or directories) in p recursively. Comparing these type of tools with PFTL we can state some major differences, and also clearly motivate the interest in this new approach:

– With PFTL we describe the type of files or directories that are to be processed, and how, instead of blindly processing every file. This means that there can be types of files or directories that are not processed, or are processed by different functions.
– Also, since the target files for processing are chosen based on type, we can have many heterogenous processors that share the same directory tree.

---

[4] It can be as simple as to return the file mime-type information, or sub-classed by the user to detect more complex types if required.

- The type of file or directory is determined before calling any processing function, which means that it can influence the way the element is processed.
- PFTL uses *lazy evaluation*, it only calculates next elements to be processed as required, this means that processing files or directories can give origin to other files that will also be candidates for processing later.
- PFTL syntax is ruled base, tools written with this language are simple and elegant, easy to maintain, and they tend to keep that way even when complex procurement tasks are required while when using other tools the complexity of the code tends to increase.
- There is no easy way to tell these common tools to do some kind of processing for a group of files or directories, instead of independent files.

3 PFTL Description

The main goal of PFTL is to allow a formalized description of the structure represented in the filesystem and the tasks needed to be performed in order to process it.

The design of a new DSL is usually made to make programming of very specific tasks easier for the end-user. Specially if the code needed to implement the same behavior in a GPL would obscure the relevant code that would deal with the program main task. Also, it improves programs correctness, and maintainability while decreasing developing time.

In this specific case, our goal is to make programming tree processors easier, faster and maintainable. As with almost any DSL a syntax definition for the new language is required. Typically this syntax is designed based on one of these three options:

1. Borrowing a syntax that is already defined and is well known in the area.
2. Designing a completely new syntax, that is invented and applied for the first time.
3. Use a syntax that is already known and used in other contexts or areas, and that can be used as a metaphor, i.e. a syntax that can be applied in an different area from the one that it was originally intended for.

We opted for the third approach. It is clear to us the similarities between our description of the filesystem tree structure, and grammars. Therefore, instead of creating a new syntax, we adopted a formalism similar to grammars so it could be easy for other people to quickly understand the syntax.

Continuing with the grammar metaphor, to describe a filesystem tree processor we use the following formal approach:

$$processor = (N, T, P, S, I)$$

Where:

- N is the list of non-terminal symbols, $\forall\, nt \in N : nt \in L$;
- T is the list of terminal symbols, $\forall\, t \in T : t \in L$;
- P is the production set;
- S is the starting point, or axiom, $S \in N$;
- I is a set of special instructions specific for the compiler;

– L is the complete set of symbols that can be used, in practice its the set of types returned by the *TypeOf Oracle*.

N and T are sets of keywords, written using only alphanumeric characters, for example: `Name`, `Book`, `Chapter`, `File`, etc, that belong to L, where L is the set of possible types that the *TypeOf Oracle* can produce. S is the first non-terminal symbol that appears in the production set, and P is defined as:

$$P = p*$$

where:

$$p = N \times rhs \times A$$
$$| \ T \times A$$
$$rhs = (T \cup N)*$$
$$A = \{semantic \ action\}$$

This means that our production set P is a list of productions p. Each one of these productions is either a non-terminal symbol followed by a rhs and an action A, or simply a terminal symbol followed by an action A. The rhs is a mixed list of terminal and non-terminal symbols in which this specific non-terminal symbol derives. Each of these symbols can be followed by a single $*$ (asterisk), which implies that the symbol can be found more than once. A semantic action is a snippet of code that implements the desired semantic action for each rule.

A very simple example of a production p without a defined semantic action is:

```
Directory ---> File*;
```

The special arrow (`--->`) is just syntactic sugar to distinguish between the symbol on the left, and the mixed list of symbols that non-terminal symbol `Directory` derives in a list of `File` symbols. Each production should terminate with a semicolon (`;`).

A semantic action can be added to any production enclosed in curly brackets before the closing semicolon. Therefore, a complete production looks like:

```
Directory ---> File* { print "Found directory" };
```

I is a set of special instructions that can be included in the processor description, but these are specific instructions for the compiler. They are related with the *TypeOf Oracle*, and therefore they will be described in section 5, that is dedicated entirely to this subject.

3.1 Semantic Actions

Semantic actions can be added to production rules to achieve any kind of effect while processing the filesystem. Currently, the code for the actions needs to be written in Perl (the host language for our DSL). These blocks can be written exactly as any other Perl program, and they can use other tools and modules to perform any arbitrary task. The only particular thing about this code is that a set of special variables with valuable information are automatically defined before the semantic action is called.

The list of special variables that can be used in the actions block are defined below:

- $_t$ the type of the left hand side (as returned by the *TypeOf Oracle*);
- $_p$ full path to the name of the file or directory being processed;
- $_n$ the name of the file or directory being processed (if it is a directory, its name is the last directory name in the path);
- $_c$ includes the content of the file being processed (undefined when processing directories);
- $_v[i]$ is a list, where each position is related to one of the symbols in the production (right hand side), and hold their processed values (or returned value).
  That is, given our processing is depth first, all files and sub-directories are processed before the parent directory is processed. Therefore, when processing the parent, this array will have the result of processing each of its child.
- $_l[i]$ is a list of associative arrays (or hash tables) that represent the right side of the derivation rule), one associative array for each symbol.
  For each one of these associative arrays there are the keys _p, _n, _c and _v, which have the same meaning as the variables defined above.
- $_j$ is the result of joining the right hand side of results (by default results are concatenated).

A simple example of a production rule using a special variable is:

```
Text { print $_c };
```

This rule means that when the terminal symbol `Text` is found, a side effect is produced by the semantic action, printing the `$_c` special variable, i.e. printing the file contents. More illustrating examples of semantic actions and the use of special variables can be found in section 6.

Keep in mind that the semantic action is written in Perl, so any kind of arbitrary side effect can be produced. Given the following rule for example:

```
Text {
  $db->execute("INSERT INTO Texts VALUES ($_n, $_c)");
};
```

a database would be populated with the name and content of the set of text files being processed.

4 The PFTL Compiler

In order to process a filesystem tree using the language described in the previous section, a special program, similar to a compiler, is required. This compiler takes a PFTL program and the initial path to the directory tree to be processed, and produces some kind of result that depends entirely on the tasks performed in the semantic actions. An abstraction of the compiler architecture is illustrated in figure 1.

The first task of the compiler is to parse the source program and build a tree representation of the structure defined in the program. Once this tree is built the compiler can start processing the filesystem tree.
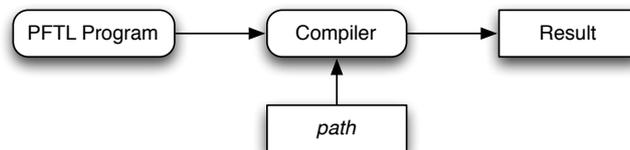
Fig. 1: PFTL compiler architecture overview.

Again, the grammar metaphor was used here. In the next step the compiler processes each file and directory individually, and tries to match these symbols (terminal and non-terminal) with the production rules in the derivation tree.

In order to compare the current element being processed (directory or file) with the production rules tree, the elements need to have an associated type. This association is made by a special program, the *TypeOf Oracle*, which is described in detail in section 5. For now, think of it as the *lexer*, that analyses the text and discovers the token types.

These types have the same names of the symbols that were used to describe the directory structure in the production list of the PFTL program. In sum, and keeping the grammar metaphor, the compiler looks at directories and files as sentences and tries to find in the production set a derivation tree that matches this sentence. If the derivation tree is found, the corresponding semantic action for that production is executed.

For the parsing stage of the compiler a yapp base parser was implemented in Perl [3] and for the second stage a simple grammar-like engine was implemented.

5   The *TypeOf Oracle*

The production set in a PFTL program is written using a set of terminal and non-terminal symbols. These symbols represent the type of files (or directories) being processed. To compute this type, a special tool is used: the *TypeOf Oracle*, that given a file determines its type.

The core of this tool is a set of functions, that try to correctly guess the type of the file being processed. With this set a queue is created in runtime so that functions have a notion of priority and are executed in the desired order. PFTL also provides a specific syntax to add functions to the beginning (higher priority) or the end (lower priority) of this queue, or to force a specific function in the set to be ignored. Behind the hood there is already a set of functions available out of the box. Of course it is possible to write our own functions, or ignore the functions used by default.

The process of giving a type to an element (being this a directory or a file) always starts by checking if there is any special META information specifying the element type. This is always the first step and it can not be overridden (although it can be ignored). After testing if the META information is available, the set is then processed in order, like a queue, which means that function A will try to assign a type to the element being processed, and only if it fails B will tried. This behavior is illustrated in figure 2. The

Fig. 2: Default *TypeOf Oracle* queue.

user can add, remove or ignore in this set of functions. Table 1 summarizes the different options available.

Directive	Effect
`%t_add T`	add new function `T` to the beginning of the queue
`%t_append T`	add new function `T` to the end of the queue
`%t_ignore T`	ignore function `T` in the set

Table 1: Functions available to manipulate the *TypeOf Oracle* set of functions.

Where `T` is the name of a function defined in the same scope as the processor.

Finally, once the queue is processed and if a type has not been found, the last typifier is called, this simply returns if the element is a file or a directory.

When a typifier returns a true type value, a string, the process stops and the returned type is used. Most of the times this flow is enough, but in some cases we want to continue processing the queue, even if a valid type was already returned. For example if we are trying to find a more specific type for a XML file. If after processing the rest of the queue we can not find a more specific type, then the previously found will be used.

6 PFTL Example Programs

This section introduces some applications that were implemented in PFTL and that can be executed using the compiler described in section 4.

6.1 Creating a LaTeX Book

The goal of this example application is to implement a tool that can process a directory tree containing LaTeX, and other files, in order to build a book.

The first level of the directory states the title of the book and, inside this directory, every directory is a chapter (named upon the directory name). Finally, inside each chapter directory, all files are considered content for that specific chapter.

These files will be handled in different ways: if an image file is found, the LaTeX code is added to include this image; if a plain text file is found the content of this file is included in the document in a `Verbatim` environment; and if a LaTeX file is found its content is included directly in the resulting file. The full application program is shown below:

```
S ---> Book {
    write_file('book.tex', $_[1]);
};
Book ---> Chapter* {
    "\\documentclass{article}\n\\begin{document}\n"
      . $_[1]
        . "\\end{document}"
};
Chapter ---> tex png* txt* {
    "\\section{$_n}\n" . $_j
};
tex { $_c };
png { "\\includegraphics{$_n}\n" };
txt { "\\include{$_n}\n" };
```

This program states that the beginning of the tree is a `Book`. A `Book` derives in a collection of `Chapters`, where each `Chapter` derives in any combination of LATEX, images, or plain text files. Each production rule in the program has an associated semantic action that is producing the required LATEX code to build the final document.

Please note the advantage of using this approach, taking benefit of the composition that is possible to achieve for the various production rules. The `tex`, `png` and `txt` rules are good examples of this, they compute some results on their own, that are returned to the tree and used later in another production rule. In the `Chapter` rule the result of performing all the actions for the symbols in the right hand side of that production are used to produce the content of the LATEX file by using the special variable `$_j` that contains the result of concatenating all the computed results.

6.2 Creating a *World Atlas*

Imagine we are storing information about countries in the world, and how countries are divided in a hierarchical way. So, the root node of our tree will be the `/World`. On the first level the world is divided in continents, and on the next level in countries. One possible way to do this division in a directory structure is as follows:

```
+-- /World
    +-- /Asia
    +-- /Europe
        +-- /Portugal
            +-- info.txt
            +-- flag.png
            +-- anthem.mp3
        +-- /Spain
    ...
```

We want to create an HTML file with all this information (the text present in the *info* file, a link to the anthem music file and a thumbnail of the country flag). A simple program to do it can be written as:

```
World ---> Continent* {
    $res = "Continents: <ul>";
    foreach $_l[1] {
      $html.="<li> $_->{_n} </li>";
    }
    write_file("index.html", $res);
};
Continent ---> Country* {
    $res = "Countries: <ul>";
    foreach $_l[1] {
      $html.="<li>a href='$_->{_n}'>$_->{_n}</a> </li>";
    }
    return $res."</ul>";
};
Country ---> info flag anthem {
    write_file("$_n.html", $_j);
};
info { "<pre>$_c</pre>" };
flag { "<img src='$_n' />" };
anthem { "<a href='$_p'>Anthem</a>" };
```

In this processor we are building an *index.html* that contains unsorted lists of countries and continents. For each country we are creating a new HTML file with the information provided for each country.

6.3 Real World Examples

Due to the major benefits of using PFTL, it was already adopted in real world scenarios.

The CROSS Project

The CROSS project aims at developing new program understanding and analysis techniques and combine them for quality assessment of open source code. In this context one task particular goal was to devise a tool that could process every file in a software package accordingly to the file type, it could be a documentation file, a source code file, a mix of both, a README file, a Makefile, etc. We can look at a software package as a directory tree, in which there are files of heterogeneous types that may be divided in directories.

This was an excellent opportunity to test our tool with a more complex application. The goal of this tool is to process every file in a distribution package, and for each file according to its type perform some specific task of information discovery. This example clearly takes advantage of the feature discussed earlier, of discovering information in a well known context, this means that for example the tool will only try to discover information in files were that data is expected to be. In practice this will result in less false positives and better results.

To prove the use of this approach we chose a specific distribution, a well known package – a Perl Module package file. The idea is to have a PFTL program that is able to process all the files in a package, and act accordingly, i.e. call a special program that is specialized in gathering a specific type of information. Our main program, still with no semantic actions could look a bit like:

```
Package ---> Meta Makefile Readme Changes License Lib*;
```

Now the idea is to add semantic actions to each production rule to call the required tools for each type of file, for example:

```
Readme {
    my $i = Cross::Inspector::Readme->new(path=>$_p);
    my $r = $i->process;

    $db->store('Readme',$r);
};
```

This illustrates the major advantage on the adoption of PFTL instead of a typical tree processing tool. A package may contain one or more files that *The TypeOf Oracle* labels as `Radme` files, and that is acceptable because it can be true, there can be an independent file, or a documentation section, etc. But in any case the tool specialized in retrieving information from these sources is called. This increases the accuracy of the information gathering tools, because they are only called for files that are prone to provide useful information. And of course, the *TypeOf Oracle* accuracy can also be improved if required.

"Museu da Pessoa"

In this case a simpler prototype of PFTL was used, but the advantages of adopting this approach was already clear. In this particular museum an heterogenous collection of documents (from images, to texts, or sound files with interviews) was available in files, spread across directory trees. And the goal was to provide a view of this knowledge in a website. With a simple description of the content, and small semantic actions to process specific type of files and tool that was able to create a entire website from the museums' collection was quickly implemented. This implementation is so easy to maintain and to add features, and that can be executed whenever new content is added to the collection that this tool still builds most of the site that is available today.

With the immense quantity of different content formats, and different ways to compose this content to build HTML pages this would have never been possible with a typical *apply function f to all files in path p recursively* approach. See [2] for more details.

7 Conclusion

Directory trees of files are a common artifact for storing information, because they are easy to create and filesystems are generally available. Usually the main problem is the

lack of a systematic way to process it. Conventional approaches use generic traversal algorithms (depth-first or breathe-first, is just irrelevant), where the framework does nothing more than entering and exiting folders. All the semantic on the directory tree processing is passed to the user code, that should check current directory depth, file types, and so on.

With PFTL this task gets simplified. The directory structure is no longer a simple tree, with nodes and leafs, but an annotated tree, where nodes and leafs have types. Describing formally processors for this structure is simple, especially taking into account the fact that most programmers are familiar with parsing tools (like yacc or Bison) and therefore can easily grasp the way PFTL works.

The implemented examples show that PFTL is versatile, and can be used effectively in very distinct types of operations, from data-mining to document generation tasks.

Future work will include a broader range of type detection functions, better diagnosis tools and, hopefully, a full featured manual.

Acknowledgments

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.
2. J. João Almeida, J. Gustavo Rocha, P. Rangel Henriques, Sónia Moreira, and Alberto Simões. Museu da pessoa – arquitectura. In *Encontro Nacional da Associação de Bilbiotecários, Arquivista e Documentalistas, ABAD'01, Porto*, Maio 2001.
3. C. Frenz. *Pro Perl Parsing*. Apress, 2005.
4. Tomaz Kosar, Pablo Martinez Lopez, Pablo A. Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Inf. Softw. Technol.*, 50(5):390–405, April 2008.
5. M. Mernik, J. Heering, and T. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316 – 344, 2005.
6. Nuno Oliveira, Maria João Varanda Pereira, Pedro Rangel Henriques, and Daniela da Cruz. Domain specific languages: A theoretical survey. In *INForum'09 — Simpósio de Informática*, pages 35 — 46, Lisboa, Portugal, September 2009. Faculdade de Ciências da Universidade de Lisboa.
7. Alberto Manuel Simões, José João Almeida, and Pedro Rangel Henriques. Directory Attribute Grammars. In *VI Simpósio Brasileiro de Linguagens de Programação*, pages 297–308, 2002.
8. Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.
9. William M. Waite and Lynn Robert Carter. *An Introduction to Compiler Contruction*. Harper-Collins, 1993.