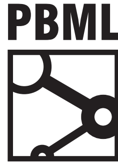

The Prague Bulletin of Mathematical Linguistics

NUMBER 101 APRIL 2014 55–69

Pipeline Creation Language for Machine Translation

Ian Johnson

Capita Translation and Interpreting

Abstract

A pipeline is a commonly used architecture in machine translation (MT). Statistical MT can require, for example, many computational steps during training and decoding which are best viewed as a pipeline. These pipelines can be tedious to construct and are error prone since little or no type checking is performed, and no clear interface is defined for a pipeline's components. At times this can manifest itself as components requiring knowledge of how a preceding component's output is formed in order to consume it. Moreover, collaboration and sharing of pipelines or components becomes difficult since the components themselves may need to be changed in order to be used by others.

In order to alleviate these problems a specialised language has been designed called Pipeline Creation Language (PCL). PCL allows users to construct pipelines that have components with well defined and compatibility checked interfaces. Components can be defined in packages, so individual components or packages of components, including entire pipelines, can be shared and composed with others. PCL supports operators which allow components to be executed sequentially, in parallel, or conditionally.

1. Introduction

Machine translation seems to attract constructions that are best seen as pipelines. For example, statistical MT (SMT) tends to use pipelines to build up the computation needed for its training and decoding phases. Pipelines typically contain many components that can:

- Have no clear interface: What types of input data are required for this component to compute which types of output data? Also, what types of data are required to instantiate the component? Or,

© 2014 PBML. All rights reserved. Corresponding author: ian.johnson@capita-ti.com
Cite as: Ian Johnson. Pipeline Creation Language for Machine Translation. The Prague Bulletin of Mathematical Linguistics No. 101, 2014, pp. 55–69. doi: 10.2478/pralin-2014-0004.

- Be too coupled: A subsequent component may need to alter the data on its input from another component. The logic needed to do this is usually placed in the subsequent component. This “glue” logic prevents the component being easily re-used in the same or another pipeline.

In this case the user of a component does not know whether it is compatible with others. Is the data, or the format of the data being output compatible with another component a developer wishes to use? Plus, these problems can prevent components being shared by a community and so limiting collaboration.

The development and maintenance of pipelines can be difficult and tedious. Typical pipeline implementations, with many components, can bear little resemblance to their conceptual view. Moreover, the implementations tend to be monolithic which can lead to developers becoming lost in a mass of code. Due to these difficulties coding errors can occur that are missed.

Pipeline Creation Language (PCL) provides a specialised language that allows the construction of non-recurrent software pipelines that provides:

- **Compatibility checking**: Components define an interface that specifies named data flowing to and from input and output ports. The ports are checked so that the compatibility of two connecting components can be determined.
- **Packages and modules**: Components can be defined in a hierarchical manner. This eases the management of pipelines with many components since one component can be worked on at once.
- **Testing**: Components can be easily tested in isolation using the PCL runtime and configuration files.
- **Promotes component re-use**: Components should be developed in isolation from any other component such that they could be reused. Also, libraries of components can be constructed using packages. If “glue” logic is needed to attach two components then the “knowledge” of how to do this resides in the recipients pipeline.
- **Representational implementation**: Pipeline implementations “look” like pipelines, unlike when implemented using a procedural approach.
- **Sharing and collaboration**: All of the above make sharing of components and collaboration easier. This is how modern programming languages create ecosystems of libraries that are sharable and used in many applications.

A PCL compiler and runtime, including documentation, is freely available, under a LGPL v3 license, from GitHub by cloning `https://github.com/ianj-als/pcl.git`.

2. PCL and Experiment Management

Experiment managers allow experimenters to run and re-run experiments using a collection of scripts or executables configured into, usually, a software pipeline and handle experiment inputs and experiment results. An experiment manager ensures that each run of an experiment has a distinct location for its results. Experiment man-

agers, such as Moses' EMS (Philipp Koehn, 2010) and Eman (Ondřej Bojar and Aleš Tamchyna, 2013), handle two concerns; the definition of a software pipeline, and collating results.

The PCL language can be used for the software pipelining aspect of an experiment manager. The PCL runtime could be used to build an experiment manager since the runtime presents a public API. For more information on the runtime API please see the PCL user manual (Ian Johnson, 2013).

3. Implementing an SMT Training Pipeline

In order to demonstrate how PCL can be useful for MT pipelines, a simplified training pipeline, shown in Figure 1, is to be constructed. The training pipeline generates a translation and language model, and then tuning is done to produce models which can be used in a decoding pipeline. A similar pipeline, along with documentation, can be found in the Moses GitHub repository, see `contrib/arrow-pipelines` in a clone of `https://github.com/moses-smt/mosesdecoder.git`.

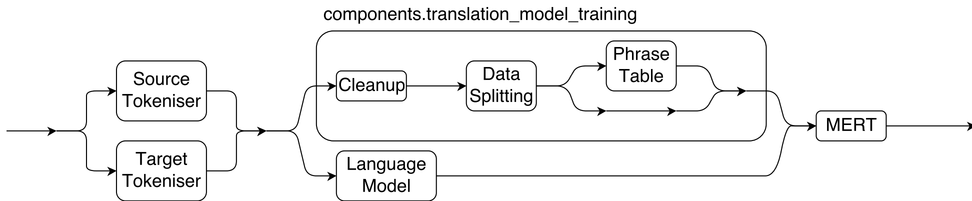

Figure 1. Simple SMT training pipeline.

3.1. Pipeline Components

In PCL a pipeline component is a reusable unit of computation that is instantiated and used in other components. Existing programs that are being used in your pipelines can be adapted for use by wrapping them in a PCL module. This PCL component can be imported, and combined with other PCL components.

PCL components can have 2, 3, or 4 *ports*, and each port should contain one or more *signals*. A port is the point, or points, at which one component attaches to another. Components may attach to each other if, and only if, they have a matching number of output and input ports. A signal is a piece of named and "duck typed" data that flows through a port. Ports can have an unlimited number of signals. *Port specifications* are used to determine the number of input and output ports and their signals. Single ports are defined as a comma separated list of signal names, e.g. `corpus.source.filename, corpus.target.filename`. Dual ports are defined using two parenthe-

sised and comma separated list of signal names, e.g. `(corpus.source.filename, corpus.source.no_sentences), (corpus.target.filename, corpus.target.no_sentences)`. This information is used to determine whether components are compatible with each other.

There are two kinds of component in PCL and use slightly different syntax, they are:

- **Computational Component**: Represents a computation that shall be evaluated when the component is executed. These components only have dependencies on the PCL runtime and are "leaves" in the component dependency tree.
- **Combinator Component**: These components combine other components, both computational and combinator kinds, to create new components.

Figure 2 shows the component model used in PCL. This is a structural design pattern called the *composite* pattern (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, 1994).

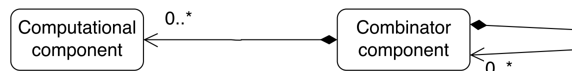

Figure 2. PCL's component model.

3.2. Computational Component

An example of how a batch tokeniser can be wrapped in PCL is shown in Figure 3. Those familiar with Haskell will see that the imperative style is similar to *do-notation* (see Miran Lipovača (2011)).

This PCL file defines the computation for a tokeniser component. It is composed of the following sections:

- **Imports** (lines 1-5): These imports are part of the runtime library and provide basic and common operations. These files are written in Python and define functions. Once imported the module alias is used with the function name to generate a function call, e.g., `path.join(...)`. The search for runtime library imports is determined by the `PCL_IMPORT_PATH` environment variable, and allows users to define their own runtime libraries.
- **Name** (line 7): A meaningful name for the component. Components should be uniquely named in a package. The file name of the component must be the same as the component's name, e.g., a file called `tokeniser.pcl` must contain a component called `tokeniser`.
- **Ports** (lines 8 & 9): The input and output port specifications of the component. Computational components can *only* define one input and one output port.

```
1  import pcl.io.file as file
2  import pcl.os.path as path
3  import pcl.system.process as process
4  import pcl.util.list as list
5  import pcl.util.string as string
6
7  component tokeniser
8    input corpus.filename
9    output corpus.tokenised.filename
10   configuration corpus.language, working.directory.root, moses.installation
11   do
12     language <- string.lower(@corpus.language)
13
14     corpus.file.basename <- path.basename(corpus.filename)
15     corpus.file.basename.bits <- string.split(corpus.file.basename, ".")
16     list.insert(corpus.file.basename.bits, -1, "tok")
17     result.basename <- string.join(corpus.file.basename.bits, ".")
18     result.pathname <- path.join(@working.directory.root, result.basename)
19
20     working.exists <- path.exists(@working.directory.root)
21     if working.exists == False then
22       path.makedirs(@working.directory.root)
23       return ()
24     else
25       return ()
26     endif
27
28     tokeniser.cmd <- path.join(@moses.installation, "scripts",
29                                "tokenizer", "tokenizer.perl")
30     tokeniser.cmd.line <- list.cons(tokeniser.cmd, "-l", language, "-q")
31
32     corpus.file <- file.openFile(corpus.filename, "r")
33     result.file <- file.openFile(result.pathname, "w")
34     process.callAndCheck(tokeniser.cmd.line, corpus.file, result.file)
35     file.closeFile(result.file)
36     file.closeFile(corpus.file)
37
38     return corpus.tokenised.filename <- result.pathname
```

Figure 3. `tokeniser.pcl`: An example of how an existing tokenisation script can be used in PCL.

- **Configuration** (line 10): The configuration is static data which is used to instantiate the component. This data is optional since components may not require any parameters to construct them. Prefixing an identifier with `@` references a configuration value, e.g., `@corpus.language`.
- **Computation** (lines 11-38): The component must define the computation that maps input signals to output signals. This section begins with the `do` keyword. Execution flows from top to bottom and each line yields a value which can be assigned to a "write once" identifier. This section *must* end with a return statement which assigns values to all output signals.

Once all of the computational components have been defined they can be combined, using PCL, to create more complex components. The next section describes how PCL combines components into, in this instance, the training pipeline.

3.3. Pipeline Implementation

A combinator component representing the training pipeline is shown in Figure 12. A PCL file that defines a combinator component is composed of the following sections:

- **Imports** (lines 1-4): Imports can be optionally specified. Importing, as in other languages, makes available other components to the PCL component being written. PCL components can be defined in namespaces. PCL components in namespaces must be fully qualified by using dot separated names. The search for PCL components is determined by the `PCL_IMPORT_PATH` environment variable. This is a colon separated list of directories where PCL namespaces are to be found. Only one PCL component can be imported with one import statement and each imported component must be given an alias. The component shall be known by its alias.
- **Component** (line 6): This starts the component definition and provides its name. The file name of the component must be the same as the component's name. For example, a component defined in `training_pipeline.pcl` must be called `training_pipeline`.
- **Inputs and Outputs** (lines 7 and 8): Defines the input and output port specifications of the component. This information is used to determine if a component is compatible with another.
- **Configuration** (lines 9-13): This is an optional section that defines static data which shall be used to construct this component.
- **Declarations** (lines 14-36): This optional section is used to construct components which will have been imported. The imported component's alias is used to build, possibly, more than one instance of the component. If an imported component requires configuration to be constructed the importing component's configuration must be mapped using a declaration's *with* clause. The with clause defines which configuration, possibly renamed, shall be passed to the component's constructor.
- **Definition** (lines 37-58): Beginning with the `as` keyword, this section defines a single expression which represents the pipeline. Constructed components, predefined components (see Section 3.5) and combinator operators (see Section 3.4) can all be used to create the component's definition.

3.4. Component Combinator Operators

Behind the scenes, combinator components are implemented as *arrows* (John Hughes (2000), Paterson (2001), Paterson (2003), John Hughes (2005), and Conor McBride

(2011)). Arrows are abstractions of computation that define a set of combinator operators which construct, as a result, another arrow. Arrows can be combined indeterminately and this allows any arbitrarily complex PCL component to be used in another component. There are five component combinator operators defined in PCL, they are:

- **Composition**: Join one component's output to the input of another component, e.g., the PCL expression composes components $f : b \rightarrow c$ and $g : c \rightarrow d$ with $f >>> g$. The PCL compiler shall compatibility check the two components when using the composition combinator. The input and output port specifications are b and d respectively.

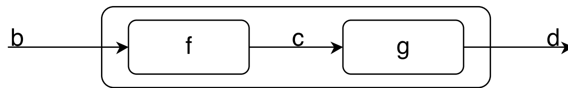

Figure 4. >>>: Component composition.

- **First**: A component that takes a component, e.g. $f : b \rightarrow c$, which will apply the first element of the input pair to f. The second element of the pair passes through unchanged. In PCL, `first f`. The input and output port specifications are $(b), (d)$ and $(c), (d)$ respectively.

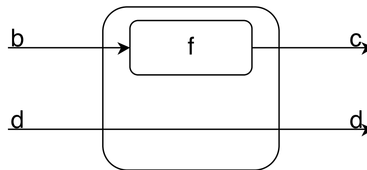

Figure 5. `first`: First element of input pair applied to component f.

- **Second**: Similar to first only the second element of the input pair is applied to the component, and the first element passes through unchanged. In PCL, `second f`. The input and output port specifications are $(d), (b)$ and $(d), (c)$ respectively.
- **Parallel**: The components $f : b \rightarrow c$ and $g : d \rightarrow e$ can be executed in parallel by using $***$ combinator. In PCL, $f *** g$. The input and output port specifications are $(b), (d)$ and $(c), (e)$ respectively.
- **Fanout**: The components $f : b \rightarrow c$ and $g : b \rightarrow d$ receive the same input and are executed in parallel. In PCL, $f \;\&\&\&\; g$. The input and output port specifications are b and $(c), (d)$ respectively.

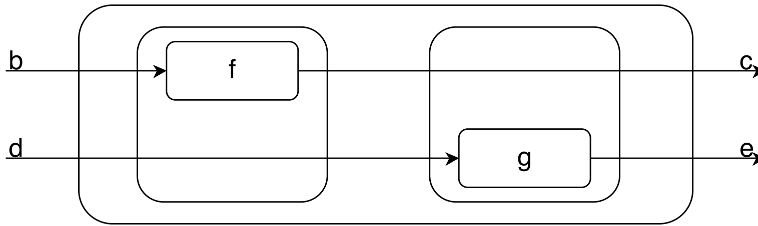

Figure 6. $***$*: Parallel components.*

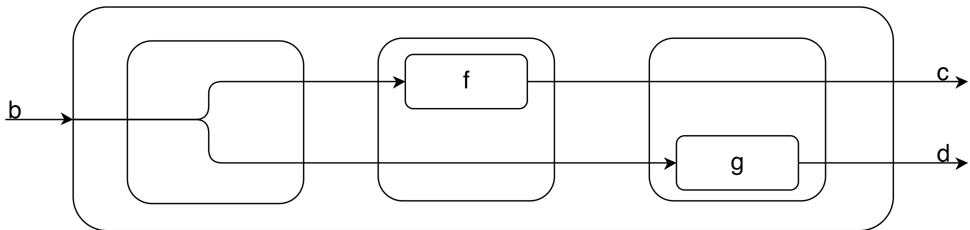

Figure 7. &&&*: Input applied to parallel components.*

3.5. Pre-defined Components

There are four pre-defined components available for use in PCL which split, merge and convert signals. Also, there is an *If* component which allows components to be conditionally executed.

3.5.1. Split

The `split` component is a component with a single port and two output ports. This component takes the signal(s) on the input port and copies them to each of the output ports.

3.5.2. Merge

The `merge` component has a dual input port and a single output port. This component merges signals from the input pair to unique signals in the output port. Signals in the input ports are used as indices to the keywords `top` and `bottom` referring the the top and bottom input port. Also, literal values can be "injected" in a merge component or signals can be dropped. All top and bottom signals must be specified in a merge. The example merge component in Figure 8 shows the top port's signals

`eval.data.filename`, and `eval.data.size` begin mapped to the output signal `evaluation_filename` and dropped respectively. Also, the `language.model.filename` sig-

```
1   merge top[eval.data.filename] -> evaluation_filename,
2         top[eval.data.size] -> _,
3         bottom[language.model.filename] -> language_model,
4         9 -> language_model_type
```

Figure 8. An example merge component.

nal from the bottom port is mapped to output signal `language_model`, and the literal value `9` is mapped to an output signal called `language_model_type`. This merge component has the following input and output port specifications

`(eval.data.filename, eval.data.size), (language.model.filename)`

and

`evaluation_filename, language_model, language_model_type`

respectively.

3.5.3. Wire

A wire component renames signals so that components can be used together. Wires can have 2 ports, one input and output port, or 4 ports, two input and output ports. Two port wires contain one signal mapping such as shown in Figure 9, and a four port wire is shown in Figure 10. As with merge components, literal values can be injected and signals can be dropped in wires, but all input signals must be specified.

```
1   wire src_filename -> filename,
2        src_language -> language,
3        src_no_sentences -> _,
4        True -> is_file_clean
```

Figure 9. An example two port wire component.

```
1   wire (tokenised_filename -> tokenised_src_filename,
2         tokenised_filename_size -> _,
3         "de" -> tokenised_src_language),
4        (tokenised_filename -> tokenised_trg_filename,
5         tokenised_filename_size -> _,
6         "en" -> tokenised_trg_language)
```

Figure 10. An example four port wire component.

3.5.4. Conditional Execution with *If*

Components can be conditionally executed using the *If* component. The *If* component takes three arguments:

- **Condition expression**: when evaluated if this is a "truthy" value the *then* component is executed, otherwise the *else* component is executed.
- **Then component**: a component which is executed on the condition being of a "truthy" value, and
- **Else component**: a component which is executed on the condition *not* being of a "truthy" value.

Both the *then* and *else* components must have identical input and output port specifications. The condition can contain the usual comparison operators (==, !=, >, <, >=, and <=), logically operators (`or`, `and` and `xor`), input port signal names and literal values. For example, `src_language == "en" and trg_language != "th"`.

4. PCL Compiler

The PCL compiler is located in `src/pclc` of your Git clone, and is called `pclc.py`. The compiler has a number of command line options that are shown in Table 1 along with their meanings.

Option	Meaning
`-i, --instrument`	The object code is instrumented with logging messages. These messages will appear on `stderr` when the pipeline is executed.
`-l LOGLEVEL, --loglevel=LOGLEVEL`	Logging level of the compiler during compilation. This affects the content of the `pclc.log` file. Valid values of `LOGLEVEL` are: `CRITICAL`, `ERROR`, `WARNING`, `WARN`, `INFO`, `DEBUG`.
`-v, --version`	Show the compiler's version and exits.
`-h, --help`	Shows the compiler's help information and exits.

Table 1. PCL compiler command line options.

To compile a PCL component called `tokeniser.pcl` use the command:

```
pclc.py tokeniser.pcl
```

This shall generate three files: the object file `tokeniser.py`, a `__init__.py` file, and a compilation log file called `pclc.log`.

5. PCL Runtime

The PCL runtime can be found in the `src/pcl-run` directory of your Git clone. The PCL runtime has some command line options that are shown in Table 2 along with their meanings.

Option	Meaning
`-n NO_WORKERS,` `--noworkers=NO_WORKERS`	Number of pipeline evaluation threads and defaults to 5 threads. The runtime executes the pipeline in a thread pool whose size is governed with this option. The performance of a pipeline may be dependent on the value used.
`-v, --version`	Show the compiler's version and exits.
`-h, --help`	Shows the compiler's help information and exits.

Table 2. PCL runtime command line options.

The runtime requires a configuration file, specified on the command line, to run the component of the same name. To execute a component called `tokeniser` invoke the runtime using:

```
pcl-run.py tokeniser.cfg
```

On `stdout`, providing all goes well, the output of the pipeline shall be displayed. The `PCL_IMPORT_PATH` environment variable is used to search for runtime libraries and compiled components.

5.1. Runtime Configuration File

The pipeline configuration file contains the static configuration and the pipeline's inputs. The configuration filename must be the same as the component you wish to run with a `.cfg` extension, e.g. the `tokeniser` configuration file must be called `tokeniser.cfg` and must be in the same directory. The configuration file contains two sections `[Configuration]`, for configuration values, and `[Inputs]`, for pipeline inputs. Each section contains key value pairs, e.g. the `tokeniser` configuration might look like the example in Figure 11. Environment variables can be used in configuration files with `$(VAR_NAME)`. The environment variable, if it exists, shall be substituted and used in the pipeline. The ability to execute any pipeline component facilitates testing during pipeline development. It is recommended that a configuration file and test data be developed alongside a component.

```
1 [Configuration]
2 corpus.language = en
3 working.directory.root = tokenisation
4 moses.installation = /opt/moses
5 [Inputs]
6 corpus.filename = my_corpus.src
```

Figure 11. An example PCL configuration file.

6. PCL Performance

The pipeline implemented here is shown in Figure 1. It is used to show the runtime overhead using PCL compared with a Bash script implementation. The PCL implementation is shown in Figure 12 and is the *top* level component definition which defines the entire pipeline. The Bash script, however, implements each of the components to emulate the PCL pipeline, and executes each "component" sequentially. The entire PCL implementation and the Bash script can be found in the `contrib/arrow-pipelines` directory of the Moses GitHub repository.[1]

The parallel corpus used is 50,000 sentences extracted from the English to Dutch Europarl corpus. Each training pipeline implementation and thread count combination was executed three times. The PCL implementation executed with 1 thread is equivalent to the Bash script, when executed, since components in both implementations are forced to execute sequentially.

The execution times for each performance experiment is shown in Table 3. The times shown represent the median, minimum and maximum execution times, in seconds, for each performance experiment.

Implementation	Thread Count	Real execution times (s)		
		Median	Min	Max
PCL pipeline	5	276	275	278
PCL pipeline	1	279	277	281
Bash script	1	650	643	654

Table 3. Performance results for the implemented training pipeline.

The PCL implementation completes in under half the time of the Bash script implementation. This is due a slow running implementation of the *Cleanup* component in the Bash implementation of the pipeline. The Bash implementation takes around 390s to execute, compared to around 1s in Python. The Bash implementation of the *Cleanup* component uses the command `wc` to count the number of words in each source and

[1] `https://github.com/moses-smt/mosesdecoder.git`

target sentence in the corpus. The *wc* command is, therefore, executed 100,000 times for this corpus. This means that at least 100,000 processes, but is probably as many as 500,000 processes, need to be spawned in order to process this corpus. The *wc* command, once cached, will execute in around 1ms; including the other processes required in this component it is conceivable that the elapsed time, on the test hardware, could be as much as 500s.

Increasing the number of threads for this pipeline only produces a marginal performance increase since the number of active branches in this pipeline is low. Moreover, the *Mert* component can only execute once *all* other components have completed and generated their output signals, and any files that need to be written to disk. It, also, takes up much of the execution time, around 180s in both implementations.

Discounting the *Cleanup* component's performance in Bash, the PCL implementations take around 7% longer to execute than the Bash script using this parallel corpus. The PCL runtime, therefore, introduces a minimal execution time overhead, and the pipeline developer gets all the advantages of the PCL language and component validations by the compiler.

7. Summary

Pipeline creation language (PCL) is a specialised, and modular language for building non-recurrent software pipelines. This paper briefly describes how PCL can be used to construct an, albeit simplified, SMT training pipeline. The PCL language, and how to integrate existing programs with PCL was described along with a performance test to show the minimal impact on execution time. It is recommended that the PCL user manual (Ian Johnson, 2013) be consulted for further details.

Acknowledgements

This work was done as part of the MosesCore project sponsored by the European Commission's Seventh Framework Programme (Grant Number 288487).

```
1  import components.translation_model_training as model_training
2  import components.wrappers.irstlm_build.irstlm_build as lang_model
3  import components.wrappers.mert.mert as mert
4  import components.wrappers.tokeniser.tokeniser as tokeniser
5
6  component training_pipeline
7    inputs src_filename, trg_filename
8    output moses_ini_filename
9    configuration source_language, target_language, max_segment_length, corpus_development_size,
10                 corpus_evaluation_size, alignment_method, reordering_method, smoothing_method,
11                 tokenisation_directory, translation_model_directory, language_model_directory,
12                 mert_directory, moses_installation_directory, giza_installation_directory,
13                 irstlm_installation_directory
14   declare
15     src_tokeniser := new tokeniser with source_language -> corpus.language,
16                                         tokenisation_directory -> working.directory.root,
17                                         moses_installation_directory -> moses.installation
18     trg_tokeniser := new tokeniser with target_language -> corpus.language,
19                                         tokenisation_directory -> working.directory.root,
20                                         moses_installation_directory -> moses.installation
21     model_training := new model_training with max_segment_length -> model_training.max_segment_length,
22                           corpus_development_size -> model_training.corpus.development_size,
23                           corpus_evaluation_size -> model_training.corpus.evaluation_size,
24                           translation_model_directory -> model_training.translation_model.dir,
25                           alignment_method -> model_training.method.alignment,
26                           reordering_method -> model_training.method.reordering,
27                           source_language -> model_training.src.language,
28                           moses_installation_directory -> model_training.moses.installation,
29                           giza_installation_directory -> model_training.giza.installation,
30                           target_language -> model_training.trg.language
31     irstlm := new lang_model with irstlm_installation_directory -> irstlm_installation_dir,
32                                   smoothing_method -> irstlm_smoothing_method,
33                                   language_model_directory -> language_model_directory
34     mert := new mert with source_language -> source_language, target_language -> target_language,
35                           moses_installation_directory -> moses_installation_dir,
36                           mert_directory -> mert_working_directory
37   as
38     (wire src_filename -> src_filename, trg_filename -> _ &&&
39      wire trg_filename -> trg_filename, src_filename -> _) >>>
40
41     (wire (src_filename -> corpus.filename), (trg_filename -> corpus.filename) >>>
42      (src_tokeniser *** trg_tokeniser) >>>
43      wire (corpus.tokenised.filename -> tokenised_src_filename),
44           (corpus.tokenised.filename -> tokenised_trg_filename)) >>>
45
46     merge top[tokenised_src_filename] -> tokenised_src_filename,
47           bottom[tokenised_trg_filename] -> tokenised_trg_filename >>>
48
49     ((wire tokenised_src_filename -> src_filename, tokenised_trg_filename -> trg_filename >>>
50       model_training) &&&
51      (wire tokenised_trg_filename -> input_filename, tokenised_src_filename -> _ >>> irstlm)) >>>
52
53     merge top[moses_ini_filename] -> moses_ini_filename,
54           top[evaluation_data_filename] -> evaluation_data_filename,
55           bottom[compiled_lm_filename] -> trg_language_model_filename,
56           bottom[add_start_end_filename] -> _, bottom[lm_filename] -> _,
57           3 -> trg_language_model_order, 9 -> trg_language_model_type >>>
58     mert
```

Figure 12. PCL implementation of the simple training pipeline.

Bibliography

Conor McBride. Kleisli arrows of outrageous fortune. (unpublished), 2011.

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, chapter Structural Patterns. Addison-Wesley, 1994. ISBN 0201633612.

Ian Johnson. *Pipeline Creation Language (PCL): User Manual*. Capita Translation and Interpreting, 2013. URL `https://github.com/ianj-als/pcl/blob/master/documentation/pcl-manual.latest.pdf`.

John Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37:67–111, May 2000.

John Hughes. Programming with Arrows. In *Advanced Functional Programming*, pages 73–129. Springer Berlin Heidelberg, 2005.

Miran Lipovača. *Learn You a Haskell for Great Good!*, chapter A Fistful of Monads. No Starch Press, 2011. ISBN 1593272839.

Ondřej Bojar and Aleš Tamchyna. The Design of Eman, an Experiment Manager. *The Prague Bulletin of Mathematical Linguistics*, 99:39–58, September 2013.

Paterson, Ross. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, Sept. 2001.

Paterson, Ross. Arrows and computation. In *The Fun of Programming*, pages 201–222. Palgrave, 2003.

Philipp Koehn. An Experimental Management System. *The Prague Bulletin of Mathematical Linguistics*, 94:87–96, September 2010.

Address for correspondence:
Ian Johnson
`ian.johnson@capita-ti.com`
Capita Translation and Interpreting,
Riverside Court, Huddersfield Road,
Delph, Lancashire,
OL3 5FZ, United Kingdom.