Shuffle: manipulating source fragments

Atze Dijkstra

September 28, 2012

Contents

1 Introduction and tutorial

Shuffle takes files with chunks of text defined in shuffle notation, and filters these chunks according to a requested selection of variant and aspects. This allows for the following typical use:

- Transform a chunked source file to a compilation unit (for ghc, C, etc). One file then is transformed one-to-one to the corresponding output without inclusion of chunks from other files. All variants and aspects are defined in the same file. The convention is to let a chunked file producing a file with suffix .X have the suffix .cX
- Transform a set of chunked source files to a single file for further processing. This is typically done for LaTeX files where various topics are spread over multiple files. On the commandline the set of files is specified, the first file being the root, other chunks are referred to by a naming mechanism.

1.1 Basics

A minimal input to shuffle looks like the following:

```
%%[1
some text
%%]
```

The two percent symbols `%%` at the beginning of a line start shuffle commands. `%%[1 .. %%]` delimits a chunk for variant 1. With this content placed in file `doc-ex1`, the following invocation of shuffle produces the content of the chunk:

```
shuffle --gen-reqm=1 --variant-order=1 --plain --lhs2tex=no doc-ex1
```

With `--gen-reqm` the required variant is specified, where variants are ordered according to the ordering passed via `--variant-order` (here just a single variant), `--plain` means that no further interpretation of chunks should be done, and `--lhs2tex=no` that no chunk delimiting for lhs2tex should be done. Although shuffle is invoked by `shuffle` in the above, shuffle is not installed globally as part of the ehc install. It can be found in `EHC/bin`, where `EHC` is the ehc directory. Text outside chunks is considered shuffle comment and not copied to output. If there are no lines between chunks then no lines are generated; if there are ¿= 1 lines between chunks one line of space is printed between generated individual chunks. The original order of chunks as it appears in the chunked source file is maintained. None of the commandline parameters currently can be omitted; in future versions of shuffle it is likely that default values will be assumed.

A variant order allows for a hierarchical ordering of variants, where later variants build on top of earlier variants, possibly overriding earlier definitions:

```
%%[1
some text
%%]

%%[2
more text
%%]
```

With the following commandline we get both text fragments, where variant 2 builds on top of variant 1.

```
shuffle --gen-reqm=2 --variant-order="1 < 2" --plain --lhs2tex=no doc-ex1
```

Variant 1 can still be extracted by `--gen-reqm=1`.

Chunks can be explicitly overridden, say a new variant 3 wants to replace the text for variant 1. We then have to give a name to the overridden chunk so we can refer to it when overriding the chunk:

```
%%[1.sometext
some text
%%]

%%[2
more text
%%]

%%[3 -1.sometext
other text
%%]
```

With the following commandline:

```
shuffle --gen-reqm=3 --variant-order="1 < 2 < 3" --plain --lhs2tex=no doc-ex1
```

this will give:

```
more text

other text
```

Again, previous variants can still be retrieved via the `--gen-reqm` option.

1.2 Output specific configuration

The output of shuffle can be tailored to more specific targets, for example haskell source code. Chunks then contain haskell source text with optional meta information about name of the module, the imports, and the exports:

```
%%[1 module Some import(SomeImport)
%%]

%%[1 export(someFunction)
someFunction :: Int -> Int
someFunction x = x
%%]
```

On the commandline we then need to specify that we want Haskell source text to be generated; the `--hs` option is used for that purpose:

```
shuffle --gen-reqm=1 --variant-order="1" --hs --lhs2tex=no doc-ex1
```

The corresponding output then is:

```
module Some
( someFunction )
where
import SomeImport

someFunction :: Int -> Int
someFunction x = x
```

Similarly we can specifically can generate for the AG system. The input then looks like the following:

```
%%[1 ag module Some import(SomeAGImport)
%%]

%%[1 hs import(SomeHaskellImport)
%%]

%%[1 ag
DATA SomeData
  | SomeAlt1

ATTR SomeData [ | | someAttr: Int ]

SEM SomeData
  | SomeAlt1  lhs.someAttr = 1
%%]

%%[1 hs export(someFunction)
someFunction :: Int -> Int
someFunction x = x
%%]
```

Because the AG system process both AG notation and escaped+copied Haskell code, it is necessary to tag each chunk with its source code type. It then ends up in the appropriate part. We now get wiith the following commandline:

```
shuffle --gen-reqm=1 --variant-order="1" --ag --lhs2tex=no doc-ex1
```

we now get:

```
{
module doc-ex1
( someFunction )
where
import SomeHaskellImport

}
INCLUDE "SomeAGImport.ag"
{
}
DATA SomeData
  | SomeAlt1

ATTR SomeData [ | | someAttr: Int ]

SEM SomeData
  | SomeAlt1  lhs.someAttr = 1

{
someFunction :: Int -> Int
someFunction x = x
}
```

1.3 Reusing and including chunks in other files

Chunks can be included at arbitrary places by referring to them by name. For example, given the content of the files `doc1` and `doc2`. In `doc2` the content of the chunk is defined.

```
%%[1.someText2
some text2
%%]
```

In `doc1` we refer to the chunk in `doc2` by `%%@doc2.1.someText2`:

```
%%[1
some text
%%@doc2.1.someText2
%%]
```

The reference must start at the beginning of the line and follows the syntax `file.variant.name`. The corresponding output then is:

```
some text
some text2
```

when shuffle is invoked with:

```
shuffle --gen-reqm=1 --variant-order=1 --plain --lhs2tex=no doc1 doc2
```

The first file given to `shuffle` acts as the root file. The remaining files (and the root file) act as a repository of chunks, for which output is only generated when included within the root file.

2 Variants and Aspects

A chunk contains content to be used for specific variants and aspects. A variant is denoted by a number, an aspect by an alphanumeric identifier. Variants and aspects are used and specified independently and represent the two dimensions over which variation can be specified. Variants and aspects though are used in a different way and with a different purpose. Variants are used to describe stepwise increments thus forming a hierarchy. Aspects are used to isolate such groups of variants. A variant is to be used for adding language features; an aspect is to be used for a global property valid for all language features. For example, in EHC, codegeneration and the type system are considered seperate aspects.

Variants and aspects are offered by chunks. When shuffle is invoked a specific combination of variants and aspects is required/requested. For example, the following respectively specifies a general chunk, 2 chunks for 2 different aspects, and a chunk to be used when both aspects are requested for:

```
%%[1
some general text
%%]

%%[(2 asp1)
some asp1 text
%%]

%%[(1 asp2)
some asp2 text
%%]

%%[(1 asp1 asp2)
some asp1 && asp2 text
%%]
```

2.1 Specifying required chunks

When we ask for all aspects with:

```
shuffle --gen-reqm="(2 asp1 asp2)" --variant-order="1 < 2" --plain --lhs2tex=no doc1
```

we indeed get the content for all aspects:

```
some general text
some asp1 text
some asp2 text
some asp1 && asp2 text
```

Via option `--gen-reqm="(2 asp1 asp2)"` we specified the required variant and aspects. The syntax of the required variant and aspects is `variant | '(' variant aspect* ')'`, where the absence of aspects means all aspects. Variant and aspects are defined independently and have the effect that only chunks offering both variant and all required aspects are generated on output. For example, the following shuffle invocation only produces the general text:

```
shuffle --gen-reqm="(1 asp1)" --variant-order="1 < 2" --plain --lhs2tex=no doc1
```

The invocation

```
shuffle --gen-reqm="(2 asp2)" --variant-order="1 < 2" --plain --lhs2tex=no doc1
```

produces both the general text and asp2 only text:

```
some general text
some asp2 text
```

2.2 Specifying offered chunks

The specification of chunk offerings follows a richer language for specifying aspects, including conjunction `&&` and disjunction `||`. The specification `(1 asp1 asp2)` of the example is a shorthand notation for `(1 asp1 && asp2)`, which means that the chunk only is generated when both aspects are asked for simultaneously. When disjunction `||` is used, as in `(1 asp1 || asp2)`, only at least one of the aspects must be asked for. The syntax of such an 'offering' specification then is `variant | '(' variant aspectexpr? ')'`, where the absence of the aspect expression means 'any aspect'. An aspect expression follows the usual notation and priority rules as used in (say) Haskell.

3 Chunks

3.1 Overriding and alternatives

Chunks can be introduced, overridden completely or partially. Chunk introduction and overriding has been discussed, for example a chunk is introduced by:

```
%%[1.sometext
some text
%%]
```

and is overridden completely in variant 2 by:

```
%%[2 -1.sometext
some text
%%]
```

Completely overriding a chunk can be too much if only a part of a chunk has been modified. There are three ways to deal with this, using already introduced notation and notation for conditional nested chunks (for the third way). First, a large chunk can be split into smaller, so we have finer grain control for overriding:

```
%%[1
some part 1
%%]
%%[1.somepart2
some part 2
%%]
%%[1
some part 3
%%]
```

If we want to override `some part 2` we can now do this as usual:

```
%%[2 -1.somepart2
some part 2 new
%%]
```

However, the partitioned chunk now has a lot of chunk related clutter. A second solution is to name all parts, include those in a new overall chunk and override the latter:

```
%%[somepart1
some part 1
%%]
%%[somepart2
some part 2
%%]
%%[somepart3
```

```
some part 3
%%]

%%[1.all
%%@somepart1
%%@somepart2
%%@somepart3
%%]
```

Chunks without a variant number are never included automatically but only when referred to by name from another chunk which is included. We override by:

```
%%[somepart2new
some part 2 new
%%]

%%[2 -1.all
%%@somepart1
%%@somepart2new
%%@somepart3
%%]
```

Again, a lot of clutter, so usually the best solution is to inline the choice using conditional nested groups of chunks. The original text structure and ordering is then maintained, although we still require shuffle notation:

```
%%[1
some part 1
%%[[1
some part 2
%%][2
some part 2 new
%%]]
some part 3
%%]
```

A nested chunk is denoted by `%%[[ ... %%]]` instead of `%%[ ... %%]`. Conditional nested groups of chunks are denoted by `%%[[ ... %%][ ... %%]]` A nested chunk always is for a particular variant; a name without a variant is not allowed.

We can also use this mechanism to specify defaults for aspects:

```
%%[1
some part 1
%%[[1
some part 2
%%][(2 asp)
some part 2 new asp
%%][2
some part 2 new
%%]]
some part 3
%%]
```

If for variant 2 aspect `asp` is requested the text `some part 2 new asp` is generated, otherwise the default `some part 2 new` for variant 2. If more than one nested chunk can be chosen from, `shuffle` chooses arbitrarily.

3.2 Metadata: type, wrapping, module + import/export, ..

A chunk may have data associated other than textual content. The use of this information depends on the context it is used in. For example, when asking for AG output (option `--ag`), the type of a chunk influences how its content ends up in the generated output:

```
%%[1 hs
haskell
%%]

%%[1 ag
AG
%%]
```

The type `ag` specifies that the chunk content is AG text, `hs` specifies it is Haskell. However, when asking for Haskell output (option `--hs`) this difference is ignored. The metadata defined for a chunk textually starts right after the `%%[` and ends at the end of the line. See the syntax diagram for chunks (page 8) below for the exact placement and values of metadata:

- `chunktype`, as discussed above. Additionally haddock comment (wrapped in `{-| ... -}` can be included. However, when AG is generated its position in the output is not deterministic as the AG compiler also shuffles around with Haskell code.
- `chunkref`, as discussed earlier, used to override a particular chunk or include it.
- `chunkwrap`, additional wrapping of a chunk, currently only used for output latex. `code` wraps as a lhs2tex code block, `safecode` puts this in a parbox as well (used within [[http://latex-beamer.sourceforge.net/][beamer]]), `verbatim` wraps inside a verbatim environment, normal size or small, `tt` is similar to `verbatim` but will become obsolete.
- `module`, adds a Haskell for type `hs` module definition when Haskell or AG is generated.
- `import` and `export`, take import/export lists and generate according to Haskell or AG and type `hs` or `ag`.

Inside and entity or module name special syntax allows the use of key/value pairs. Key/value pairs are defined on the commandline by option `--def=key:value` and the value is used by referring to the key by `%{key}`. In EHC this is used for externally specifying the toplevel module name of the library constructed for a particular variant. Overlapping Haskell namespaces are thus avoided and libraries for different variants can be used simultaneously. Also, grouping of name elements is done by `{ ... }`, guaranteeing whitespace free output generation for its content.

Syntax for chunk definitions:

```
<chunk>             ::= '%%[' (<chunkvariantdef> | <chunknameddef>) <chunkcontent> '%%]'
<chunkvariantdef>   ::= <variantoffer>
                        ('.' <nm>)?
                        ('-' (<chunkref> | '(' <chunkref>* ')'))?
                        <chunkoptions>
                        <module>? <imports>? <exports>?
<chunknameddef>     ::= <nm>
<chunkcontent>      ::= <textline>*
<variantoffer>      ::= <variantnr> | '(' <variantnr> <aspectexpr>? ')'
<variantnr>         ::= <int>
<aspectexpr>        ::= <aspectand1> ('||' <aspectand1>)*
<aspectand1>        ::= <aspectand2> ('&&' <aspectand2>)*
<aspectand2>        ::= <aspect>+
<aspect>            ::= <ident>
<chunkref>          ::= (<fileref> '.')? (<variantnr> '.')? <nm>
<nm>                ::= <ident> ('.' <ident>)*
<chunkoptions>      ::= (<chunkwrap> | <chunktype>)*
<chunkwrap>         ::= 'wrap' '=' ( 'code' | 'safecode'
                                   | 'tt' | 'tttiny'
                                   | 'verbatim' | 'verbatimsmall'
                                   | 'beamerblockcode' <str1>
                                   | 'boxcode' ('{' <frac> '}')?
                                   )
<chunktype>         ::= 'hs' | 'ag' | 'haddock' | 'plain'
```

```
<module>            ::= 'module' <str2>
<imports>           ::= 'import' '(' <entities> ')'
<exports>           ::= 'export' '(' <entities> ')'
<entities>          ::= <entity> (',' <entity>)*
<entity>            ::= <str2>
<str2>              ::= <str1> | <ident> | <int> | '{' <str2>+ '}' | '%' '{' <ident> '}"
<str1>              ::= '"' <non-">* '"'
<frac>              ::= <int> ('.' <int>)?
```

3.3 Content

Content of a chunk consists of a mixture of plain text and shuffle interpreted parts. See the syntax diagram for chunk content (page 9) below. Each piece of content either is a line not starting with `%%@` or `%%[[` or a reference to another chunk or a nested conditional group. A plain line consists of characters. The only escape to shuffle not starting at the beginning of a line consists of `%%@{ ... %%}` and `%%@[ ... %%]`:

- The `%%@{` form allows the value of key/value pairs to be substituted.
- The `%%@[` form allows inlining of file content and output of shell commandline invocation. If the type of inlining is `file` the full text between `%%@[` end `%%]` is opened as an url, and its content inlined. For inlining type `exec` the remainder after the colon is executed as a shell command and its output is inlined. For example, the commandline invocation of `shuffle` itself is inlined in this document by means of

  ```
  %%@[exec:bin/shuffle --help%%]
  ```

Syntax for chunk content:

```
<chunkcontent>      ::= <content>*
<content>           ::= <textcontent>
                      |  '%%@' <chunkref> ('@' <variantreqm>)? <chunkoptions>
                      |  '%%[[' <variantoffer> <chunkoptions> <chunkcontent>
                         ('%%[' <variantoffer> <chunkoptions> <chunkcontent>)*
                         '%%]]'
<textcontent>       ::= <char>*
                      |  '%%@{' <str2> '%%}'
                      |  '%%@[' <inlinetype> ':' <char>* '%%]'
<inlinetype >       ::= 'file' | 'exec'
<variantreqm>       ::= <variantnr> | '(' <variantnr> <aspect>? ')'
```

3.4 References

4 Output

4.1 Haskell

4.2 AG

4.3 LaTeX / Lhs2TeX

4.4 Plain

5 Makefile generation

Dependencies between AG files are explicitly encoded in makefiles. These dependencies can be generated from the AG imports in a chunked AG file using `shuffle`. We demonstrate this by means of an example from the `ruler` project.

There are typically two ways to compile an AG file: as a module with semantic functions, or as a module with data type definitions. In order to compile the AG files, the makefile infrastructure requires the following lists:

`RULER3_AG_D_MAIN_SRC_AG`	AG modules with data type definitions
`RULER3_AG_D_DPDS_SRC_AG`	Dependencies of the above AG modules
`RULER3_AG_S_MAIN_SRC_AG`	AG modules with semantic functions
`RULER3_AG_S_DPDS_SRC_AG`	Dependencies of the above AG modules

Aside from these lists, there needs to be a rule for each AG source file and its dependencies, to the derived AG module:

```
RULER3_EXPR_EXPR_MAIN_SRC_AG := $(patsubst %, \
  $(SRC_RULER3_PREFIX)%.cag, Expr/Expr)
RULER3_EXPR_EXPR_DPDS_SRC_AG := $(patsubst %, \
  $(RULER3_BLD_PREFIX)%.ag, Expr/AbsSynAG)
$(patsubst $(SRC_RULER3_PREFIX)%.ag,$(RULER3_BLD_PREFIX)%.hs, \
  $(RULER3_EXPR_EXPR_MAIN_SRC_AG)) : $(RULER3_EXPR_EXPR_DPDS_SRC_AG)
```

These lists and rules are generated by `shuffle` on a file with paths to AG modules, using the `--dep` option and several parameters to choose names for these makefile variables. For example, the file `files-ag-d.dep` contains the following lines:

```
Expr/Expr.cag
Ty/Ty.cag
AbsSyn/AbsSyn1.cag
```

The invocation of `shuffle`:

```
shuffle files-ag-d.dep --dep \
  --depnameprefix=RULER3_ \
  --depsrcvar=SRC_RULER3_PREFIX \
  --depdstvar=RULER3_BLD_PREFIX \
  --depmainvar=RULER3_AG_D_MAIN_SRC_AG \
  --depdpdsvar=RULER3_AG_D_DPDS_SRC_AG \
  > files-ag-d-dep.mk
```

results in a makefile containing the required rules and the lists `RULER3_AG_D_MAIN_SRC_AG` and `RULER3_AG_D_DPDS_SRC_AG`.

Note that `shuffle --dep` takes only and all chunk-imports tagged with an `ag`-kind into consideration.

6 Shuffle commandline invocation

6.1 Commandline usage

As printed by `shuffle --help`:

```
Usage shuffle [options] [file ([alias=]file)*|-]

options:
  -a                          --ag                                                                  generate code
  -h                          --hs                                                                  generate code
  -l                          --latex                                                               generate code
                              --preamble[=yes|no]                                                   include preamb
                              --line[=yes|no]                                                       insert #LINE p
  -p                          --plain                                                               generate plain
                              --text2text                                                           generate code
                              --index                                                               combined with
                              --gen=all|<nr>|(<nr> <aspect>*) (to be obsolete, renamed to --gen-reqm)  generate for v
```

```
-g all|<nr>|(<nr> <aspect>*)  --gen-reqm=all|<nr>|(<nr> <aspect>*)                                  generate for v
                              --compiler=<compiler version>                                        Version of the
                              --hidedest=here|appx=<file>                                          destination of
                              --order=<order-spec> (to be obsolete, renamed to --variant-order)    variant order
                              --variant-order=<order-spec>                                         variant order
-b <name>                     --base=<name>                                                        base name, def
                              --xref-except=<filename>                                             file with list
                              --help                                                               output this he
                              --dep                                                                output dependen
                              --depnameprefix[=<name>]                                             Prefix of gene
                              --depsrcvar[=<name>]                                                 Source base-di
                              --depdstvar[=<name>]                                                 Destination ba
                              --depmainvar[=<name>]                                                Varname for th
                              --depdpdsvar[=<name>]                                                Varname for th
                              --deporigdpdsvar[=<name>]                                            Varname for th
                              --depderivdpdsvar[=<name>]                                           Varname for th
                              --depbase[=<dir>]                                                    Root directory
                              --depign[=(<file> )*]                                                Totally ignore
                              --depterm[=(<file> => <dep>+ ,)*]                                    Dependency ign
                              --lhs2tex[=yes|no]                                                   wrap chunks in
                              --agmodheader[=yes|no]                                               generate AG MO
                              --def=key:value                                                      define key/val
```

6.2 Commandline options

- `--gen-reqm=variant | '(' variant aspect* ')'`
- `--plain`
- `--ag`
- `--hs`
- `--lhs2tex=no|yes`