The SuperC Technical Manual

March 25, 2012

Contents

1 Usage

1.1 Environment

SuperC requires three environment variables to run properly: (1) `JAVA_DEV_ROOT` should be set to the root of the `xtc` source tree. (2) the `PATH` should include `$JAVA_DEV_ROOT/xtc/lang/cpp/scripts/`; this directory contains scripts to run and test SuperC. (3) `JAVA_ARGS` should be set to increase the heap and stack sizes. The scripts directory contains a script `env.sh` that sets these and `xtc`'s `CLASSPATH`.

1.2 Executables

`java xtc.lang.cpp.SuperC` - The configuration-preserving C parser, preprocessor, and lexer.

`java xtc.lang.cpp.cdiff` - C-token diff utility that uses SuperC's lexer and directiveParser.

`java xtc.lang.cpp.FilenameService` - provides a centralized distributor of filenames for distributed processing by `superc_linux.sh`.

1.3 Core Scripts

`data.sh` - has the commented commands for all summary data used in the SuperC paper. It is not meant to be executed.

`superc_linux.sh` - runs SuperC on linux and provides other facilities related to testing SuperC on linux, e.g., extracting header paths and comparing output to the Linux allyesconfig.

`regression.sh` - runs regression tests. It is used by `xtc`'s `make check-cpp` target to regression test SuperC.

`typechef_test.sh` **and** `typechef_run_file.sh` - runs SuperC and TypeChef on TypeChef's constrained kernel.

`dynamic_analysis.sh` - summarizes the output of `-preprocessorStatistics` and `-languageStatistics`.

All of SuperC's scripts can be found in `$JAVA_DEV_ROOT/src/xtc/lang/cpp/scripts/`.

2 Instrumentation

SuperC and its scripts can report detailed statistics and analyses of compilation units. All instrumentation reports data as rows of space-delimited columns. The first column in each row is a keyword identifying the type of data, e.g., `define` for a macro definition or `max_subparsers` for the maximum subparsers used. The format of each type of data is list in the following subsections.

2.1 The `-preprocessorStatistics` Flag

Preprocessor statistics are output when the `-preprocessorStatistics` flag is turned on. Below is the format of the data output.

`define name fun|var location ndefs` - will be output every time a macro definition is encountered. `ndefs` is the number of macros definitions in the conditional symbol table *after* evaluating the definition.

`undef name location ndefs` - for every `#undef`. `ndefs` is the number of definitions *after* evaluating the undef.

`object name location depth ndefs nused` - for every object-like macro invocation. `depth` is the macro invocation nesting level. `ndefs` is the number of definitions, but `nused` is the actual number of definitions expanded. `ndefs` is always greater than or equal to `nused`. `nused` is less when some macro definitions are trimmed.

`function nargs name location depth ndefs nused` - for every function-like macro invocation. `nargs` is the number of arguments passed to this function-like macro invocation. `depth`, `ndefs`, and `nused` are the same as for object-like macros.

`hoist_function name location depth nhoisted` - every function-like macro is hoisted before invoking, even if their is only one resulting function. `nhoisted` shows how many function-like macro invocations resulted from hoisting.

`paste token|conditional token|conditional location nhoisted` - a token-pasting. Each argument's type, token or conditional, is indicated. `nhoisted` is the number of token-paste operations resulting from the hoisting.

`stringify token|conditional location nhoisted` - a token-pasting. The argument's type, token or conditional, is indicated. `nhoisted` is the number of stringification operations resulting from the hoisting.

`include name resolved location (un)guarded system|user normal|next single|computed depth` - a header include. Indicates given header name, header name resolved by the preprocessor, location of the directive, whether it was guarded, whether it was a system (`<...>`) or user header (`"..."`), whether it was a gcc-specific `#include_next` directive, whether it was from a computed header or not, and how many headers deep we are after entering this header.

`computed location normal|next depth nhoisted` - a computed include, indicating how many headers deep after performing the include, and the number of hoisted include directives. Not all hoisted headers must be valid, so `end_computed` will summarize how many actual headers were included.

`end_computed location nvalid` - after a computed include has finished, this reports how many valid headers were hoisted.

`conditional if|ifdef|ifndef|elif|else location boolean|nonboolean depth nhoisted` - a start or next conditional directive. `nonboolean` is emitted when the expression contains at least one non-boolean subexpression `nhoisted` is the number of expressions resulting from multiply-defined macros in its expression.

`endif location breadth` - an endif, its location, and that number of branches in the conditional it is ending.

`line|error_directive|warning|pragma location` - a line, error, warning, or pragma directive and its location.

In all cases, `location` has the following format:

```
file:line:col[:macro]
```

where `macro` is present only when the preprocessor usage occurs inside of a macro expansion, e.g., nested macro expansion.

2.2 The `-parserStatistics` Flag

Parser statistics are output when the `-parserStatistics` flag is turned on. Below is the format of the data output.

`iterations n` - the number of iterations of the main parsing loop.

`subparsers size iterations` - this shows how many `iterations` of the FMLR parsing loop had `size` subparsers.

`max_subparsers size` - this shows the maximum number of subparsers.

`killswitch_subparsers size` - when -killswitch is used and the naive parser reaches the limit, this reports how many subparsers there were before killing the parser.

`follow n` - the number of times `follow()` was called.

`dag_nodes n` - the number of DAG nodes from the parser. This represents the actual spaced used by the parser output in terms of nodes in memory.

`dag_nodes_shared n` - the number of DAG nodes that have more than one edge leading to it.

`empty_conditionals total with_empty`

`lazy_forks total with_lazy`

2.3 The `-languageStatistics` Flag

Language statistics are output when the `-languageStatistics` flag is turned on. Below is the format of the data output.

`c_construct name location` - emitted every time a C statement or declaration is reduced.

`typedef name location` - emitted every time there a typedef name is bound.

`typedef_ambiguity name location` - emitted every time a subparser encounters a name that is both a typedef name and a variable name in the same parsing context.

*`conditional_inside name location` - emitted every time a C statement or declaration contains a conditional when it is reduced. *Requires patching the parser (see Section 2.3.1).

2.3.1 Patching the Parser to Report Embedded Conditionals

SuperC can also report conditionals that are embedded in declarations and statements when using `-languageStatistics`. SuperC is shipped with a patch that add extra fields to stack frames and modifies the algorithm to track the conditionals and generate output. The patch can be applied in the SuperC source directory `$JAVA_DEV_ROOT/src/xtc/lang/cpp/`:

```
patch < instrument_embedded_conditionals.patch
```

The patch can be reversed like this:

```
patch -R < instrument_embedded_conditionals.patch
```

2.4 Other SuperC Flags

2.4.1 `-configurationVariables`

`config_var name` - reports a configuration variable name

2.4.2 `-headGuards`

`header_guard name` - reports a header guard name

2.4.3 `-size`

`size bytes` - reports the total compilation unit size in bytes

2.4.4 `-time`

`performance_breakdown file parsing preprocessing lexing` - reports the latencies in seconds of the file broken down by lexing alone, preprocessing plus lexing, and parsing plus preprocessing and lexing.

2.4.5 `-killswitch`

`killswitch subparsers` - reports the number of subparsers reached if the killswitch is triggered

2.5 The `superc_linux.sh` Script

`configs_superc gcc-args` - with `-e`, the gcc arguments for header include paths and the `KBUILD_NAME` macros.

`configs_all gcc-args` - with `-k`, the gcc arguments for header include paths and all configuration variables.

`grammar_succeeded file` - with `-P`, for when the preprocessed version of the file was parsed successfully.

`grammar_failed file` - with `-P`, for when the preprocessed version of the file was not parsed successfully.

`performance file time` - with `-r`, the running time in seconds of processing the file.

`performance_raw file before after` - with `-r`, the start and end times in seconds of processing the file. Output when the program `bc` is not available to calculate the running time.

`comparison_succeeded file` - with `-c`, for when the output of SuperC, preprocessed, matches the tokens of the preprocessed original file.

`comparison_succeeded file` - with `-c`, for when the output of SuperC, preprocessed, does not match the tokens of the preprocessed original file.

`comparison_error file` - with `-c`, when cdiff returned another error, meaning the test needs to be run again.

`excluded file msg` - When a file was skipped either because its configuration information was not found in the currently configured kernel or, with `-w`, its configuration was already written out to disk.

2.6 The `typechef_test.sh` Script

`performance file seconds` - reports the latency in seconds of the file, including JVM startup.

2.7 The `dynamic_analysis.sh` Script

`summary_definitions n in_conditional nobject nfunction` - the number of macro definitions, the number inside of conditionals, and the number that are object-like and function-like.

`summary_redefinitions n` - the number of define directives that overrode the configuration of an existing definitions.

`summary_invocations n trimmed nobject nfunction` - The number of macro invocations. trimmed is the number of invocations where at least one definition was trimmed.

`summary_hoisted_functions n` - the number of function-like invocations that required conditionals to be hoisted around them.

`summary_nested_invocations n` - the number of macro invocations nested in other macro invocations.

`summary_paste n hoisted conditional_arg` - the number of token-asting operations. hoisted is the number of paste operations with conditionals requiring hoisting. conditional_arg is the number of operations with a conditional argument (which should be the same as hoisted unless the conditional has only one branch and it's the same as the operation's presence condition).

`summary_stringify n hoisted conditional_arg` - the number of stringification operations.

`summary_include n computed hoisted valid_hoisted` - the number of file includes. computed is the number of computed includes. hoisted is the number of computed includes requiring hoisted. valid_hoisted is the number of computed includes with more than one valid resulting header file.

`summary_reinclude n` - the number of includes that reinclude a previously-included header.

`summary_conditionals n nonboolean` - the number of static conditionals (including all branches and endif). nonboolean is the number of if and elif conditional directives with nonboolean subpexpressions in their expression. maxdepth is the maximum nesting depth of conditionals, considering file inclusion. hoisted is the number of if and elif conditionals directives whose expressions required hoisting of conditionals due to multiply-defined macros.

`summary_error_directives n` - the number of error directives.

`summary_c_statement_and_declarations n with_conditionals` - the number of C statements and declarations, including external declarations. with_conditionals is the number of those constructs having embedded conditionals.

`summary_typedefs n` - the number of typedef declarations.

`summary_typedef_ambiguities n` - the number of typedef names ambiguously-defined as an identifier.

3 Implementing a New Language

The preprocessor and the parser are language-independent. To use them for a specific language, they need the following:

1. A token specification, e.g., `c.l` for SuperC. This specifies the token name and regular expression for each token. It also provides information to the preprocessor, specifically, which tokens are the comma, parentheses, hash mark, double hash mark, and ellipsis, and also which tokens can be used as macro names. The header file `token.h` provides preprocessor macros that make specifying tokens easy.

2. A grammar specification, e.g., `c.y` for SuperC. This is a Bison grammar file annotated with AST-building instructions. `xtc.lang.cpp.ActionsGenerator` will automatically generate an implementation of `Actions.java` which provides the parser with AST-building information, e.g., `CActionsBase.java`. `getValue()` must be implemented in a subclass of the generated class to tell the parser how to create semantic values, e.g., `CActions.java`.

3. A token creator, e.g., `CTokenCreator.java` for SuperC. This provides the preprocessor with language-dependent methods to create new tokens and paste tokens. `TokenCreator.java` defines the interface.

4. (optional) An implementation of any semantic actions defined with the `action` annotation in the grammar specification. `xtc.lang.cpp.ActionsGenerator` will generate an abstract method for each action, e.g., in `CActionsBase.java`, that should be implemented in a subclass, e.g., `CActions.java`.

5. (optional) An implementation of the parsing context and its reclassification and merge methods. The parsing context needs to implement the `Actions.Context` interface, e.g., `CParsingContext.java`