Now you C it. Now you don't!

Robert Ennals
Intel Research Cambridge
robert.ennals@intel.com

Abstract

Currently, much of the software that we would like to run fast on multi-core processors is written in C — a language in which expressing concurrency is difficult and error prone. Unfortunately, moving such programs to a new language is very difficult, since the programmers only know C, the tools only know C, and developers are unkeen to trust their projects to a language that might not be supported in a few years time. Our solution to this problem is Jekyll — a high level language that is losslessly inter-translatable with readable, editable C. This allows Jekyll to take advantage of C programmers and C tools, since they can use the C version of a Jekyll file, and means that if Jekyll ever ceases to be maintained, developers still have a usable C code-base.

1. Introduction

Since C was introduced thirty year ago, the programming language community has produced many programming languages that make it easier to write concurrent software. However, despite this, a large proportion of the programs for which performance is most important continue to be written in C.

Prior work suggests that primary reason developers continue to use C is that it has built up such a strong ecosystem that the switching costs associated with moving to a new language are too great [14, 6]. In particular:

- Their software is already written in C
- Their libraries are written in C
- Their programmers only understand C
- Their tools only understand C
- They don't want to trust a language that might not be supported in a few years time

1.1 Lossless Translation

Our language, Jekyll, reduces these switching costs by being losslessly inter-translatable with C. Every file in a Jekyll/C project corresponds directly to a C file, with both the Jekyll and C versions of a file being human-readable and fully editable. If either file is modified then the changes can be automatically reflected into the other version of the file (Figure 1).

Lossless translation removes the need for all programmers and tools to understand Jekyll, since programmers and tools that do not understand Jekyll can simply use the C versions of Jekyll files. Similarly, if Jekyll ceases to be supported, then one can simply continue development with the C version of the program.

[copyright notice will appear here]

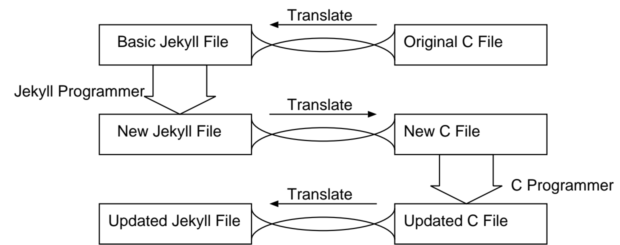

Figure 1. A Jekyll programmer and a C programmer can work on the same file

When we say that Jekyll can be translated losslessly to and from C, we mean that if one translates a Jekyll file to C, and then translates that file back to C, that file is guaranteed to be bit-for-bit identical to the original C program, preserving layout, comments, and everything else. Moreover, if a file is translated to C, edited in C, and then translated back to Jekyll, it is guaranteed that the only changes in the Jekyll file will be those corresponding to the changes made by the C programmer.

1.2 The Jekyll Language

If Jekyll was a simple C-like language with C-like features then such inter-translation might be quite simple; however Jekyll goes considerably beyond the feature set of C, offering many of the features found in modern functional languages such as Haskell [11]. Amongst other things, Jekyll currently supports lambda expressions, type classes [4], parametric polymorphism, and full type safety. Support for effect typing, ownership types, atomic sections, futures, and loop parallelism (using OpenMP [9]) is in the works.

Jekyll's lossless translation is not merely a matter of expanding Jekyll expressions to larger C expressions. Many Jekyll features do not map simply into C, and the C code will often place expressions in a very different order to the Jekyll code. Lossless translation thus requires fairly sophisticated techniques, which we describe in Section 2.

2. Lossless Translation

To achieve lossless translation, it is necessary that all whitespace (including comments) present in the Jekyll file be encoded in the C file in such a way that it can be retrieved when the file is translated back to Jekyll.

We consider all whitespace to be attached to the token that immediately follows. For example, in the following example the whitespace for "`x`" is "   " and the whitespace for `3` is " `/* hello */ `":

```
int  x = /* hello */ 3;
```

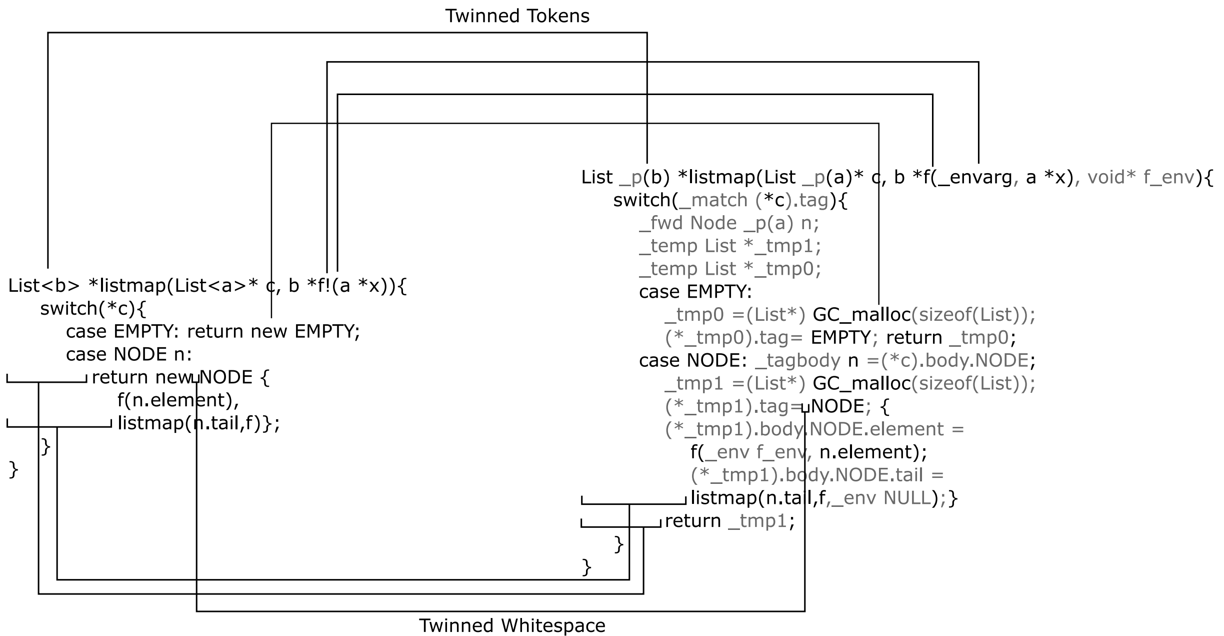

Figure 2. Jekyll and C versions of the same function — Jekyll features are encoded in Jekyll using special macros

2.1 Token Twinning

To allow whitespace to be preserved, we assign every Jekyll token a C token that is its *twin*. Every C token is either twinned with a Jekyll token, or is *untwinned*.

When a file is translated from Jekyll to C, the Jekyll translator preserves the whitespace for each Jekyll token by attaching it to the token's C twin. When the C file is translated back to Jekyll, the whitespace for each Jekyll token is retreived from the twin. Twinning thus allows us to preserve whitespace during translation.

An example of twinning is illustrated in Figure 2 which shows example token twins in Jekyll and C versions of a simple `map` function. Grayed out C tokens are those that have no Jekyll twin. We can observe the following:

- Jekyll features are encoded in C using special macros that are understood by the Jekyll translator but ignored by a C compiler.
- The whitespace preceding a token is always the same as for its twin.
- A token's twin may be a different string. For example "new" is twinned with "GC_malloc".
- The C and Jekyll files may place tokens in a different order.
- The translator choses the whitespace for untwinned C tokens to fit the surrounding code. For example the third line has been indented to match the sixth line.

2.2 Twinned Pretty Printing

So, how does the translator know which input token is twinned with which output token?

Our approach is to have a single pretty-printing function that generates token lists for Jekyll and C simultaneously, while recording which tokens are twinned together. To translate a Jekyll file to C, we match the generated Jekyll tokens against the input Jekyll tokens. We then use the twinning information to work out which input Jekyll token is twinned with which output C token, and thus deduce the whitespace for the output. This is illustrated in Figure 3.

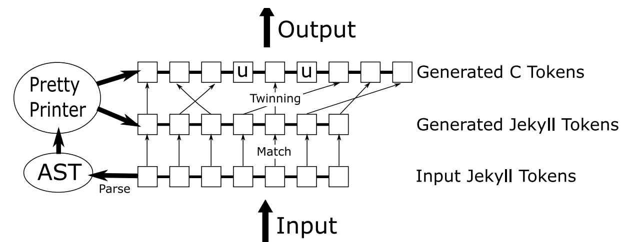

Figure 3. Whitespace from from Jekyll to C (u = untwinned)

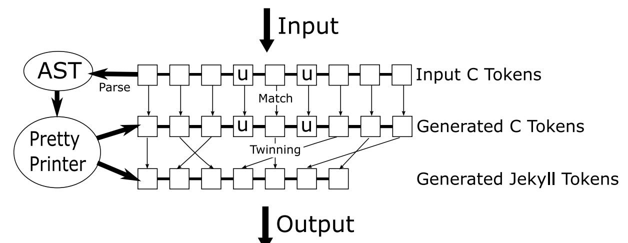

Figure 4. Whitespace flow from C to Jekyll (u = untwinned)

The procedure for converting C to Jekyll is essentially the same (Figure 4). We match the generated C code to the parsed C code to obtain the whitespace for the pretty printed Jekyll code. When translating from C to Jekyll, the translator will warn if the programmer has modified the whitespace for any untwinned C token — since such whitespace will be lost.

2.3 Malformed C files

Not all C files can be translated to valid Jekyll files. If a C file makes use of Jekyll macros in an invalid way then it will not be possible to translate it to a Jekyll file. In particular, a C file may contain sections of boilerplate code . If this boilerplate is modified incorrectly then it will not correspond to a valid Jekyll construct.

The translation technique described in the previous sections allows us to handle such malformed C files easily. Our parser ignores generated boilerplate code completely. It is the role of the matching stage (Section 2.2) to check that the parsed C tokens are the same as those that the pretty printer would pretty print for the parsed abstract syntax tree.

If the matching stage detects that the input tokens are malformed, it will give a malformed input message, such as the following:

```
File examples/demo.c : 52 characters 12 - 15
Malformed input file: expected fromInt but given fInt

    int * (*fInt)(_dictenv(Num, int), int x);
            ^^^^
```

3. Related Work

Many of the ideas in Jekyll are similar to ideas that have appeared in previous work.

All the language features present in Jekyll have appeared previously in other languages; indeed most of Jekyll's features can be found in Haskell [11], and other functional languages. These languages tend to be more elegant than Jekyll, since they do not need to be inter-translatable with elegant C, but they are not able to take advantage of C's ecosystem.

Many previous languages have extended C with new features, including C++ [13], and Cyclone [5]. Such languages tend to be more elegant than Jekyll, but any programs that use the new features will no longer be compatible with C.

Several systems have used macros and libraries to embed extra features in C. For example OpenMP [9] adds parallel features, CCured [8] adds safety annotations, and FC++ [7] supports functional programming idioms. Such languages retain full compatibility with C/C++, but are restricted by the need to stay within C syntax. Since Jekyll is inter-translatable with C, it is possible to use such systems together with Jekyll.

Many compilers for high-level languages translate to C as part of their compilation process. Examples of this include GHC [10] and CFront [12]. Unlike Jekyll, the generated C is not intended to be human readable, and the translation is not intended to be reversible.

Many people have implemented language translators that translate one language into another. For example FOR_C [1] translates FORTRAN to C, and p2c [3] translates Pascal to C. While the result of such a transformation is readable, the translation is not intended to be reversible.

Like Jekyll, refactoring editors make changes to programs while preserving layout and comments [2]. Unlike Jekyll, such editors do not map one language to another, and do not have need for the twinned-token techniques described in Section 2.

As far as we are aware, no previous work has produced a high-level language that is losslessly inter-translatable with C.

4. Conclusions

Jekyll is not perfect. The language makes considerable elegancy sacrifices in order to inter-translate with C, the current C encoding of Jekyll features can be confusing for C programmers who are not accustomed to it, many useful features are currently missing, and type-safety requires that all code be ported to Jekyll and all uses of "`unsafe`" be eliminated.

We do however believe that Jekyll is a step in a sensible direction. By being losslessly inter-translatable with C, Jekyll is able to make use of C programmers, C tools, and C libraries in a way that would not be practical otherwise. Similarly, by being losslessly inter-translatable with C, Jekyll is able to alleviate one of the software developer's worst nightmares — that their code will be trapped in a dead language for which no tools or programmers are available.

We believe the approach taken by Jekyll is more generally applicable. Indeed, we are exploring the possibility of designing languages that are losslessly translatable with Verilog and Java.

Availability

All features described in this paper have been implemented in our Jekyll translator, which is available on SourceForge at: `http://sourceforge.net/projects/jekyllc`

We encourage readers to download Jekyll and try it out.

Acknowledgments

We would like to thank Michael Dales, Simon Peyton Jones, Greg Morrisett, Alan Mycroft, Matthew Parkinson, and Richard Sharp for providing useful suggestions.

References

[1] FOR_C: Converts FORTRAN into readable, maintainable c code. `http://www.cobalt-blue.com`.

[2] FOWLER, M. *Refactoring: Improving the design of existing code*. Object Technology Series. Addison-Wesley, 2000.

[3] GILLESPIE, D. p2c – a Pascal to C translator.

[4] HALL, C., HAMMOND, K., JONES, S. P., AND WADLER, P. Type classes in haskell. In *Proceedings of the European Symposium on Programming (ESOP'94)* (Apr. 1994).

[5] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of c. In *Proceedings of the USENIX annual technical conference* (2002).

[6] MASHEY, J. R. Languages, levels, libraries, and longevity. *ACM Queue 2*, 9 (Dec. 2004).

[7] MCNAMARA, B., AND SMARAGDAKIS, Y. Functional programming in C++. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'00)* (Sept. 2000).

[8] NECULAR, G. C., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. CCured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems* (2004).

[9] OpenMP application program interface. `www.openmp.org`, May 2005.

[10] PEYTON JONES, S., HALL, C., HAMMOND, K., PARTAIN, W., AND WADLER, P. The Glasgow Haskell Compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference, Keele* (Mar. 1993), DTI/SERC, pp. 249–257.

[11] PEYTON JONES, S., HUGHES, R., AUGUSTSSON, L., BARTON, D., BOUTEL, B., BURTON, W., FASEL, J., HAMMOND, K., HINZE, R., HUDAK, P., JOHNSSON, T., JONES, M., LAUNCHBURY, J., MEIJER, E., PETERSON, J., REID, A., RUNCIMAN, C., AND WADLER, P. Report on the programming language Haskell 98. `http://haskell.org`, Feb. 1999.

[12] STROUSTRUP, B. *The Design and Evolution of C++*. Addison Wesley, 1994.

[13] STROUSTRUP, B. *The C++ Programming Language*. Addison Wesley, 1997.

[14] WADLER, P. Why no-one uses functional languages. *SIGPLAN Notices 33* (Aug. 1998).