Multi-Language Synchronization

Robert Ennals David Gay

Intel Research Berkeley,
2150 Shattuck Avenue, Berkeley,
CA 94704, USA
robert.ennals@intel.com
david.e.gay@intel.com

Abstract. We propose *multi-language synchronization*, a novel approach to the problem of migrating code from a legacy language (such as C) to a new language. We maintain two parallel versions of every source file, one in the legacy language, and one in the new language. Both of these files are fully editable, and the two files are kept automatically in sync so that they have the same semantic meaning and, where possible, have the same comments and layout.

We propose *non-deterministic language translation* as a means to implement multi-language synchronization. If a file is modified in language A, we produce a new version in language B by translating the file into a non-deterministic description of many ways that it could be encoded in language B and then choosing the version that is closest to the old file in language B.

To demonstrate the feasibility of this approach, we have implemented a translator that can synchronize files written in a straw-man language, Jekyll, with files written in C. Jekyll is a high level functional programming language that has many of the features found in modern programming languages.

1 Introduction

The programming language community has produced many programming languages that improve on legacy languages such as C in useful ways. They have produced languages that are easier to use, easier to understand, safer, more portable, more reusable, etc. But, despite all these advantages, a large proportion of important software projects continue to use legacy languages.

Why is this? Prior work suggests that one of the principle reasons why programmers continue to use legacy languages is that they have built up such a strong ecosystem around them that the switching costs associated with moving to a new language are prohibitive [28,17]. In particular:

- Much software is already written in legacy languages
- Many libraries are written in legacy languages
- Many programmers only understand legacy languages
- Many tools only understand legacy languages

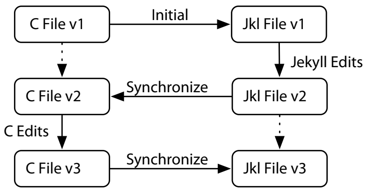

Fig. 1. *JT* keeps the Jekyll and C versions of a file synchronized

- Developers are wary of trusting a language that might not be maintained in 10 years time
- For existing projects, developers are unwilling to port large code bases to new languages, both because of the effort involved and the risk of introducing new bugs

Historically, new languages that have achieved success have overcome these issues using a combination of three techniques:

- Attack a niche in which no other language has built up a strong ecosystem (e.g., Perl [29] for text processing or SQL [5] for database queries).
- Build up a new ecosystem from scratch (e.g., Java [15] and C# [4], using the muscle of large global companies, or Ada [25] under government mandate).
- Exploit the ecosystem of an existing language by having some degree of compatibility with that language (e.g., C++ [27] and Objective C [23], which are supersets of C).

In this paper, we propose a novel way for a new language to exploit the ecosystem of legacy languages such as C. Our approach is to maintain two parallel versions of each source file, one in the legacy language and one in the new language. Both of these files are human readable, un-annotated, and fully editable. A synchronizer program propagates updates between the two files, ensuring that they remain semantically equivalent, and, as much as possible, have the same comments and layout (Figure 1). We call this technique *multi-language synchronization.*

The hope is that, by providing an editable version of a file in the legacy language, it becomes easier for a project to adopt a new language, since greater use can be made of the ecosystem of the legacy language. In particular:

- Programmers who do not know the new language can edit the legacy file
- Legacy language tools can be applied to the legacy version of the file
- Tools that generate code in the legacy language can be used with projects in the new language
- If the new language ceases to be developed, one can continue development using the legacy version of the file
- Legacy programs can begin to transition to a new language, without having to commit to abandoning the legacy language

– When legacy programmers and new-language programmers work on the same program, edits made by one group can be seen as minimal edits by the other group — preserving language-specific structure and layout

While the task of translating between two languages without losing language-specific information might seem daunting, we show that it can be done using *non-deterministic language translation*. To translate a file from language A to language B, we produce a description of many encodings of the file in language B, and then select the version that is closest to the old file in language B (Figure 2).

To demonstrate the feasibility of multi-language synchronization, we have implemented a translator, *JT*, which can synchronize files written in C with files written in a new language, called Jekyll. The design of Jekyll is not a goal in itself; rather it is intended to show that multi-language translation is possible between two fairly different languages: Jekyll is a modern functional programming language which has many of the features present in languages such as Haskell [24], ML [21], and Cyclone [16], including generic types, lambda expressions, pattern matching, algebraic datatypes, and type classes. Jekyll also has all of the features of C, although potentially unsafe features such as pointer arithmetic require use of a explicit **unsafe** keyword in order to avoid a warning (in common with C# [4]). A more complete description of Jekyll can be found in a companion tech report [8]; JT is available on SourceForge at: `http://sourceforge.net/projects/jekyllc`.

The main contributions of this paper are the concept of multi-language synchronization, presented in more detail in Section 2, and the algorithms and techniques that make multi-language synchronization possible (Section 3). In Section 4 we present a preliminary evaluation of multi-language synchronization based on our experiences with JT. This evaluation shows that multi-language synchronization does work in practice, at least on a small scale. In the future, we hope to conduct a full evaluation based on a realistic successor to C used on a large-scale software project, as part of the Ivy project [1]. We discuss related work in Section 5 and conclude in Section 6.

2 Multi-Language Synchronization

We start by outlining the basic model for, and usability requirements on, multi-language synchronization (Section 2.1), followed by a discussion of the requirements on the languages being translated (Section 2.2), For concreteness, in this section and the rest of the paper, we discuss multi-language synchronization in terms of C, Jekyll and JT. However, except when referring to language-specific features, our comments apply to multi-language synchronization in general.

2.1 Model and Usability Requirements

Our basic model for multi-language synchronization, shown earlier in Figure 1, is that at all times each source file S exists in C (S_{C}) and Jekyll forms (S_{J}). After a programmer edits the C file X_C, the system regenerates ("synchronizes") the corresponding Jekyll file X_J, based on the **new** contents of the C file and the **old** contents of the Jekyll file; edits to Jekyll files are handled in an analogous fashion. This regeneration is expected to happen frequently (e.g., after every successful build or before every commit to a source-code control system).

It is of course also possible to translate a C file to Jekyll without any previous Jekyll version (e.g., when importing an existing project). However, the presence of a previous version allows for a better translation preserving the use of Jekyll-specific features not explicitly present in the C version of the source code, as discussed in Section 3 and shown in the examples of Section 4.

Multi-language synchronization is an inexact science. A C file generated from a Jekyll file is typically not as readable as a C file written by a C programmer, and there are limits on the degree to which a C programmer can edit C code that represents a higher-level Jekyll feature before JT is unable to produce a good corresponding update to the Jekyll file.

The goal however is not to be perfect, but to be good enough to be useful. In particular, the translation should be good enough that a C programmer unfamiliar with Jekyll would find it easier to edit the C file than to edit the Jekyll file, and a developer would find it easier to use an existing C tool on the C file than to work without that tool using the Jekyll file. More generally, the translation has the following goals:

- **Semantics are preserved:** C code translated into Jekyll has unchanged behaviour, and vice-versa.
- **Edits are translated naturally:** The result of making a change to a C file and then translating it to Jekyll is close to the results of translating the original C file to Jekyll and logically performing the same change, and vice-versa.
- **C programmers can understand C code produced by JT:** Generated C code is readable, fully commented, and does not contain additional annotations.
- **JT can understand code produced by C programmers:** JT is sufficiently tolerant of edits to C code encoding Jekyll features that it can produce reasonable Jekyll updates for a large proportion of C updates.
- **No special infrastructure needed:** JT works from the text files containing the C and Jekyll source code. It does not require, for example, that all code modifications be performed by a special editor. We do however use, as outlined above, the previous version of the **target** of the translation.

Note that some of these goals may be in conflict: for instance, as we discuss in more detail in Section 3.3, the desire to produce a translation from C which preserves the use of some Jekyll feature — in support of the natural edit translation goal — may lead JT to change the semantics during translation. Such behaviour is acceptable in a translator as long as it always warns the programmer in an appropriate way, and only does it in well-justified cases (e.g., JT believes the code it was translating was buggy).

2.2 Language Requirements

We do not believe that multi-language synchronization between arbitrary pairs of languages is practical. We do believe the following properties of Jekyll and C (especially the first two) are what makes JT practical, and suggest that these should serve as guidelines in the design of other multi-language translation systems:

– **All C features can be translated reasonably easily into Jekyll.** In particular, Jekyll supports all unsafe features of C (although their use is discouraged, and warnings are produced unless the **unsafe** keyword is used).
– **All Jekyll features can be translated into reasonably readable C.** In particular, Jekyll does not support lazy evaluation or tail recursion elimination, and several features (e.g., the implementation of closures) are designed with a C encoding in mind.
– **Jekyll uses the same data-layout as C.** This is particularly important in a language such as C where low-level features allow data layout to be exposed.

3 Non-Deterministic Language Translation

One approach to maintaining two consistent versions of the same file in different languages would be to apply the actions performed on one file (e.g., rename this function, insert this code) to the other, in a fashion similar to database view updates [14,6]. However, this approach is not practical as editors do not record such information, and programmers do not edit files purely in terms of structural operations.

Instead, our approach to implementing multi-language synchronization is *non-deterministic language translation*. A modified C file can be encoded into Jekyll in many different ways. Rather than picking one of these encodings, JT translates a C file into a non-deterministic description of many of the ways that the file might be encoded as Jekyll. JT then resolves this non-determinism by attempting to choose the Jekyll file that is the closest textual match to the previous Jekyll version of the file (Figure 2). Similarly, there are many different ways that a Jekyll file might be translated to C. JT attempts to choose the decoding that most closely resembles the previous C file.

This non-deterministic approach allows JT to translate Jekyll code into completely unannotated C. It is not necessary to add information to C files since JT attempts to deduce this information from the previous Jekyll version of the file (Section 3) and preserve that encoding.

Similarly, the non-deterministic approach also allows JT to be reasonably tolerant of edits to C code while still ensuring that translation is lossless. JT allows a Jekyll feature to be encoded into C in many different ways, increasing the chances that when a C programmer edits such C code JT will still recognize it as the encoding of a Jekyll feature.

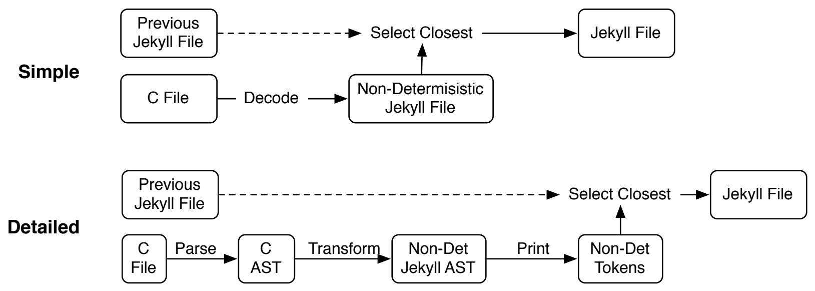

Fig. 2. The structure of the JT translation system

Perhaps most crucially, the non-deterministic approach allows the implementation to be simple and elegant. The translator need merely describe the various ways in which C and Jekyll can be encoded into each other, and the details of how to choose the correct encoding are left to a generic matching algorithm. There is no need for special-purpose code to recognize particular kinds of updates or preserve particular kinds of information, and new encodings and new language features can be added very easily.

Figure 2 illustrates the structure of the translation system used by JT. In the following sections, we will discuss this translation process in more detail.

3.1 Non-Deterministic Abstract Syntax Trees (ASTs)

When an AST is translated from one language into another, some of the nodes in the target syntax tree may be special "?" nodes that represent a non-deterministic choice of encoding/decoding. The "?" node takes three arguments, which have the following meanings:

- The **decision variable**, v, is a logical decision variable that can be either *true* or *false*.
- The **choices**, a_t and a_f are the different nodes that the non-deterministic node can resolve to. If v is *true* then the node resolves to a_t, otherwise it resolves to a_f. Although only two choices are specified, one can encode an arbitrarily long list of choices by using several nested "?" constructors.
  These choices will often have substantial similarities. To avoid exponential blow-up in the size of our AST, we allow different choices to share sub-nodes.

Decision variables allow one to specify dependencies between the choices made in different parts of the tree. This is useful since a single encoding/decoding decision may have effects in a number of places throughout the file. For example a C function that is never called directly and has its address taken once could be decoded either as a Jekyll function or as a lambda expression. Since a decision needs to be made, a decision variable is allocated. This variable will be *true* if the function is a lambda expression and *false* if the C function is just a Jekyll function. This decision variable is then used to parameterize each point in the AST at which this decision would cause the Jekyll program to be different, including the function definition and the function use.

The a_f choice is the *default* choice, and is the choice that the *select closest* stage (Figure 2) will pick if neither of the two options is a close match to the previous file. The default choice is always the most conservative choice. For example, when decoding C as Jekyll, the default is to produce Jekyll code that is identical to the original C code. Amongst other things, the default case will typically be used when new code is added to a file, or no current version exists in the other language.

3.2 Encoding Arbitrary Elements

Sometimes, when translating C to Jekyll, it is necessary to encode something like "an arbitrary type" or "an arbitrary name". For example, when translating a C type to a Jekyll type, we allow the Jekyll type to have arbitrary additional type parameters that were not present in the C type.

Given the data type given in Section 3.1 it is not obvious how one encodes something like "an arbitrary type" or "an arbitrary expression". If we were to encode all possible types or expressions using "?" nodes then we would have to build an infinite tree, significantly complicating the design of the translation system.

To avoid this problem, the core translation system mines the previous version of the file for instances of particular syntax elements. If the translate stage wants to encode "an arbitrary type" then rather than describing all types possible in the language, it lists all the types present in the previous version[1].

At first it might seem that this technique would artificially restrict the choice of types and prevent the *select closest* stage from selecting the encoding that most closely matches the previous version. However, it turns out that this is not the case. Since the *select closest* stage aims to minimize the distance from the previous version, it will always choose types that appear in the previous version in preference to types that do not. Thus there is no need to list types that do not appear in the previous version, and no need for JT to support infinite ASTs.

3.3 Checking Correctness

Sometimes a C programmer will edit C code implementing a Jekyll feature such that it is no longer a valid implementation of that Jekyll feature. For example JT requires that if a C function is implementing a Jekyll lambda expression then the first argument of that function must be the lambda expression's environment. If a C programmer changes the argument order then the function will no longer be a correctly encoded lambda expression. While we could just translate the C

[1] The actual implementation is a little cleverer than this, leaving some of the list expansion until match time.

code to equivalent low-level Jekyll code, ignoring the Jekyll feature, it is likely that this result is not what the programmer intended.

To deal with such cases, Jekyll will attempt to decode any code as a Jekyll feature if it looks like the code *intended* to encode a Jekyll feature, even if the code does not encode that feature correctly. Once JT has translated a C file to a Jekyll file, it checks that the Jekyll file can be translated back to the original C file. If it cannot then the programmer is warned that the result of the transformation may be incorrect, and is encouraged to look at the differences between their file and the correctly encoded C file.

3.4 Non-Deterministic Token Sequences

Rather than resolving non-determinism directly at the AST level, we instead translate the AST into a *non-deterministic token sequence* and resolve the non-determinism at the token level. While this approach might seem to be throwing information away, our experience so far is that this method copes better with real code edits, which do not always follow AST nesting structure[2].

This non-determistic token sequence preserves all of the non-determinism that was present in the non-deterministic AST, but reduces the abstraction level down to a sequence of strings. Non-deterministic token sequences can be described as follows:

$$\begin{array}{ll} t \leftarrow v \ ? \ t_f : t_t & \text{non-deterministic choice} \\ \quad \mid \ t_0 \bullet t_1 \mid \text{“}s\text{”} \mid \emptyset & \text{sequence, literal, empty} \end{array}$$

The *pretty print* stage produces a non-deterministic token sequence by applying a pretty printing function to each node in the non-deterministic AST. A "?" node in the AST is translated into a "?" node in the token sequence with the same decision var and with choices that are produced by pretty printing the choices from the AST node. All other nodes in the AST are pretty printed by sequencing literal tokens together with token sequences from subtrees. As with ASTs, non-deterministic token sequences use sharing to avoid blow-up.

3.5 Distance between Two Files

The *select closest* stage resolves a non-deterministic token sequence t into a deterministic token sequence t'. In so doing it attempts to minimize the *distance* between t' and the previous tokens. (Figure 2).

The distance metric we have chosen is the number of distinct *spans* needed to construct the target file from the previous file, where a span is defined to be either a single token, or a consecutive sequence of tokens from the previous file. For example, the distance from "`int x = 3; int j`" to "`int j = 3; int z`" is 3, since the new string can be constructed from the following three spans: *(i)*

[2] It may be worth experimenting more with tree-based matching.

"`int j`", *(ii)* "`= 3; int`", *(iii)* "`z`". We believe this metric fits a programmers intuitive model of what it means for files to be similar.

Note that this metric is not the same as the edit distance. Edit distance only considers insertion, deletion, and substitution of a single character; it does not consider copyings and reorderings of large blocks of text. If the order of two functions was swapped, then the edit distance would be twice the number of characters in the smaller of the two functions, while the number of spans would be 2.

3.6 Optimal Translation is NP-Hard

Ideally, we would like the *select closest* stage to guarantee that it resolves a non-deterministic token sequence to the token sequence that is closest to the previous token sequence — we refer to this problem as *optimal matching*. Unfortunately, optimal matching turns out to be NP-hard. This result is not surprising, given a similar result is known for synchronizing database views [2].

We can demonstrate that optimal matching is NP-hard by showing that it takes only a polynomial number of steps to translate any problem in 3-SAT (known to be NP-hard) into an optimal-matching problem. The encoding $[\![A]\!]$ of a 3-SAT expression A as a non-deterministic token sequence is quite simple:

$$\begin{array}{rlrl} [\![v]\!] = & v \ ? \text{ “true” } : \text{ “false” } & [\![\neg v]\!] = & v \ ? \text{ “false” } : \text{ “true” } \\ [\![A \wedge A']\!] = & [\![A]\!] \bullet [\![A']\!] & [\![A \vee A']\!] = & x \ ? \ [\![A]\!] : [\![A']\!] \\ & & & \text{where } x \text{ is fresh} \end{array}$$

The previous file is an infinite sequence of *"true"* tokens. Provided the 3-SAT formula A has more than one disjunction[3], the formula is satisfiable if and only if the optimal matching of $[\![A]\!]$ has distance of 1 (a single span of *"true"* tokens).

Fortunately, like many NP-hard problems, we have found that it is possible to produce an approximate algorithm. Our current algorithm is a simple greedy search that walks sequentially through the token sequence, choosing variable assignments such as to maximize the length of the longest matching span[4]. While the worst case performance of this algorithm is still exponential, we have found that this algorithm runs in reasonable time and produces good results on reasonable-size source files (Section 4). This is partly an artifact of the kind of non-determistic ASTs produced by JT, in which the choices at a "?" node tend to be quite different, and partly a result of the structure of C and Jekyll programs, which tend to have fairly little textual self-similarity.

3.7 Synchronizing Comments and Whitespace

It is important that any comments present in one view of a file be also present in the other file. Similarly it is important that synchronization not make gratuitous changes to the whitespace of a file.

[3] Since a single "false" token would also have distance 1.

[4] The algorithm is omitted due to lack of space. See our tech report [8] for details.

JT divides whitespace into *common* and *private* whitespace. Common whitespace is considered to be part of the program representation and is carried across during translation. The other whitespace is considered private and is inferred non-deterministically to match the previous version of the target file.

The rules for distinguishing common and private whitespace are language-specific. The intention is that common whitespace be used in places where comments are typically placed, and private whitespace be used in cases where there is no obvious corresponding location in the other language, or where the correct whitespace is likely to be language-specific. A warning is generated if comments are found in private whitespace.

4 Evaluating JT

In this section, we present a preliminary evaluation of JT. We start by showing JT's behaviour on simple snippets of code encoding non-C features (Section 4.1), then evaluate its use on edits of source files from the GNU C Compiler [26] (Section 4.2). We conclude with a discussion of the limitations of our prototype (Section 4.3).

4.1 Feature Translation

We show here how JT handles the translation between two higher-level Jekyll features not found in C: generic types and closures. The output code is a very slightly cleaned-up version of the results of the JT tool, and is similar to real examples we encountered when modifying GCC (Section 4.2).

Jekyll has generic types similar to those found in ML [21], Haskell [24], and Cyclone [16]. Type variables are written as `%a`, rather than the more conventional `'a`, to allow Jekyll files to be easily processed using the standard C preprocessor. When generic types are translated into C, all generic type information is thrown away. When translating back to Jekyll this information is reconstructed from the previous file (Section 3.2). For example:

Jekyll

```
struct<%a> Node{
  %a *element;
  List<%a> tail;
};
%a* get_element(Node<%a>* x){
  return x->element;
}
```

C

```
struct Node{
  void *element;
  List tail;
};
void* get_element(Node* x){
  return x->element;
}
```

Jekyll supports closures and lambda expressions, as found in functional programming languages such as ML. Closures are written with syntax similar to Smalltalk [13], with arguments separated from their body by a colon. A lambda expression is translated into a function with an environment argument[5].

[5] Free variables are passed by reference since they may be modified. In this case JT could have passed z by value since `ff_dbl` does not modify it.

Jekyll

```
int dbl(int z){
  return twice(0,
    {x: ret x+z;});
}
```

C

```
struct fe_dbl{
  int *z;
};
int ff_dbl(struct fe_dbl *_env, int x){
  return x+*(_env->z);
}
int dbl(int z){
  struct fe_dbl ft = {&z};
  return twice(3,(void*)&ff_dbl,&ft);
}
```

By default, Jekyll uses `ff`, `fe` and `ft` prefixes for lambda functions, closure environment types, and closure values, however the programmer is free to change these names to whatever they prefer, since JT allows arbitrary names to be used (Section 3.2).

4.2 Edit Translation

To demonstrate the behavior of edits, we took the `hashtab.c`, `hashtab.h`, and `ssa.c` files from the SPEC2006 version of GCC (3,070 lines total), and performed a sequence of edits on them. For each edit, we note the language the edit was made in (L), what the edit was and the effect it had in the other language, and the number of lines that changed in C and Jekyll, as measured by diff [6] (DC and DJ respectively). All file versions are available in the Jekyll source distribution.

L	Description	DC	DJ
C	Remove the use of macros that Jekyll does not understand.	55	
C	Convert to Jekyll – Jekyll is a near-superset of C, so only change is `#including` "hashtab.jkh" in place of `"hashtab.h"`	0	2
J	Make the hashtable generic and make the visitor callback a closure — leaves the C file largely unchanged. Most differences due to callback arguments changing order, GCC source using `PTR` in place of `void*`, Jekyll code replacing a typedef with a literal generic type	18	40
J	Update ssa.c to use lambda expressions. Generated C file is correct.	376	358
C	Rename generated lambda functions. Jekyll unchanged.	22	0
C	Rename functions, reorder functions, and insert and delete code – all mapping into correct Jekyll updates	42	43
C	Reorder arguments to the closure type – No longer recognized as a closure. Reverts back to being a basic function	2	2

No invocation of JT took more than 2.5 seconds on a Pentium 4 desktop.

4.3 Where it works, and where it doesn't work

JT has two significant limitations. First, it does not support the C preprocessor very well. Currently, Jekyll uses an ad-hoc series of annotations that tell JT how

[6] Less accurate than our distance metric, but something people are familiar with

to treat particular macros (e.g., treat like a function of this type, or ignore). We believe that the results of the Macroscope project [18] could be used to design a better approach.

Secondly, as we saw in the last edit in Section 4.2, JT does not cope well with some kinds of edits. In particular:

– Breaking encoding rules: Some encodings of Jekyll features into C have rules that must be followed. For example closures must take their environment as their first argument and features that expand to several statements require that those statements not be re-ordered. If C edits break these rules then the translation will either revert to the raw C, or generate non-equivalent Jekyll code (Section 3.3). In some cases these rules could be relaxed (e.g., reordering non-side-effecting statements), but in other cases they are necessary in order to allow meaningful translation.
– Moving Between files: JT only looks at the current file. If code is moved between files then the translation will revert to the defaults.
– Large updates: If an update has caused many separate changes to a file then JT will find it harder to find the correct decoding, since the new version will correlate less well with the old version. Synchronizing often is a good idea.

However our limited personal experience is that many kinds of update work well. In particular, any C update that does not affect code implementing a Jekyll feature is highly likely to work correctly, since the translate stage will find few things that look like Jekyll features and the select-closest phase will be unlikely to find close matches to Jekyll features. Similarly, we have found that simple transformations such as renaming variables, reordering definitions, and adding and deleting code work reliably.

Ideally, a synchronizer would be used with an interactive tool that allowed the user to pick the correct translation in cases where the correct result is unclear.

5 Related Work

Much previous work has looked at connections between different languages: bidirectional translation between different data formats, languages that are designed to extend C, languages that are translatable to C, and tools that preserve program formatting while editing. As far as we are aware, no previous work has performed bi-directional synchronization between programming languages.

5.1 Bidirectional Translation

The concept of bidirectional translation between different data formats has appeared in many different fields.

The Harmony project [9] uses a set of tree-based combinators [10] to transform data structures between different data representations, with the aim of allowing easy synchronization of data between different programs and devices.

Like JT, Harmony uses information from the previous file during translation. Unlike JT, Harmony does all matching on local subtrees, based on the names of nodes on the tree, rather than doing a global analysis based on textual comparisons. While this approach works well for the data-synchronization domain that Harmony is designed for, it is not clear whether this approach would perform well in the domain of programming language translation, where transformations are complex and edits can move expressions to arbitrary positions in a program.

Meertens [20] applies the concept of bi-directional translation to the world of user interfaces. The idea here is that a user interface provides a view onto some underlying data, and constraints are established that ensure that the user interface remains an accurate representation of the data, even when the data or the user interface is manipulated. This approach is constraint based, and it is not clear whether it could be applied to something as complex as translating between programming languages.

In the database community, there has been a lot of work on "the view update problem", in which one tries to translate an update to a view into an appropriate update to the underlying database [14,6]. As with JT, a view update is able to see the previous version of a database when applying an update to it, and will try to minimize the extent of the change made. Unlike JT, a view update operation has the privilege of being able to see the exact update commands used, rather than simply being presented with a changed file and trying to work out what was intended.

Martin Fowler proposes the idea of a Language Workbench [11], which is an IDE in which users write programs using multiple user-defined DSLs. In some cases it may be possible to represent the same AST using different DSLs (e.g., graphical and Java representations of a GUI). As with database view updates, the IDE translates operations rather than programs.

5.2 Inter-Language Translation

Many people have implemented language translators that translate one language into another. For example FOR_C [3] translates FORTRAN to C, and p2c [12] translates Pascal to C. While the resulting program is human-readable, there is no means to keep the files in sync if they are modified. Similarly, many compilers perform one-way translations to C as part of their compilation process.

5.3 Languages that extend other languages

Many languages have extended C with new features. Cyclone [16], Vault [7], Ivy [1], C++ [27], Objective C [23] and many others all add useful new features to the core C language. While existing C code is often valid in these languages, any use of new features will prevent the program being a valid C program. In principle it should be possible for us to apply the transformation techniques used by JT to translate one of these languages to and from C.

Several authors have designed systems that use macros, templates, and naming conventions to embed extra features into C programs. CCured [22] allows a programmer to annotate their C programs with safety annotations, which are used by the CCured compiler, but ignored by a C compiler. FC++ [19] is a template library that makes it easy to express common functional programming idioms. These languages benefit from the ability to retain full C/C++ compatibility without translation, but are forced to use non-optimal syntax in order to do so — as with our encoding of Jekyll into C.

6 Conclusions

While it would be necessary to perform detailed evaluations with real programming teams to determine conclusively that multi-language synchronization works in practice, our experience so far has been very positive. Those C programmers that we have shown JT to have been impressed by its ability to cope with changes to code updates and have claimed that they would be able to edit C code generated from JT.

As part of the Ivy project [1], which aims to produce a system's programming language to replace C, we intend to apply multi-language synchronization to Ivy and C, and use it to make modifications to large legacy systems. Ultimately, we aim to convince external developers to use this system.

JT, the Jekyll Translator, is available on SourceForge at `http://sourceforge.net/projects/jekyllc`

Acknowledgements

The design of Jekyll has been influenced by discussions with many people. Particular thanks must go to Michael Dales, Minos Garofalakis, Simon Peyton Jones, Bill McCloskey, Greg Morrisett, Alan Mycroft, Matthew Parkinson, Claus Reinke, Richard Sharp, Simon Thompson, and everyone in the Kent Theory group, Cambridge Systems Research Group, and Berkeley Ivy group.

References

1. Brewer, E., Condit, J., McClosky, B., and Zhou, F. Thirty years is long enough: Getting beyond C. In *Proceedings of the USENIX workshop on Hot topics in Operating Systems* (2005).
2. Buneman, P., Khanna, S., and Tang, W. C. On propagation of deletions and annotations through views. In *PODS'02* (2002).
3. FOR_C: Converts FORTRAN into readable, maintainable C code. http://www.cobalt-blue.com.
4. *C# Language Specification*. ECMA, June 2005.
5. Date, C. J. *A Guide to the SQL Standard*. Addison-Wesley, 1986.
6. Dayal, U., and Bernstein, P. A. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems 8* (Sept. 1982).
7. DeLine, R., and Fahndrich, M. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM conference on Programming Language Design and Implementation* (2001).

8. Ennals, R. Dr Jekyll and Mr C. Tech. Rep. IR-TR-2005-104, Intel Research, 2005.
9. Foster, J. N., Greenwald, M. B., Kirkegaard, C., Pierce, B. C., and Schmitt, A. Schema-directed data synchronization. Tech. Rep. MS-CIS-05-02, University of Pennsylvania, March 2005.
10. Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)* (2005).
11. Fowler, M. Language workbenches: The killer-app for domain specific languages. `http://www.martinfowler.com/articles/languageWorkbench.html`, June 2005.
12. Gillespie, D. p2c – a Pascal to C translator.
13. Goldberg, A., and Robson, D. *Smalltalk-80: The Language*. Addison-Welsey, 1989.
14. Gottlob, G., Paolini, P., and Zicari, R. Properties and update semantics of consistent views. *ACM Transactions on Database Systems 13* (Dec. 1988).
15. *The Java Language Specification, third edition*. Sun Microsystems, 2005.
16. 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).
17. Mashey, J. R. Languages, levels, libraries, and longevity. *ACM Queue 2*, 9 (Dec. 2004).
18. McCloskey, B., and Brewer, E. ASTEC: A new approach to refactoring c. In *Proceedings of the 10th European Software Engineering Conference* (Sept. 2005).
19. McNamara, B., and Smaragdakis, Y. Functional programming in C++. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'00)* (Sept. 2000).
20. Meertens, L. Designing constraint maintainers for user interaction. manuscript, 1998.
21. Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
22. Necula, 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).
23. *The Objective C Programming Language*. Apple, Oct. 2005.
24. 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.
25. Pyle, I. C. *ADA Programming Language*. Prentice Hall, 1981.
26. Stallman, R. M. *Using and Porting GNU CC (Version 2.0)*. Free Software Foundation, Feb. 1992.
27. Stroustrup, B. *The C++ Programming Language*. Addison Wesley, 1997.
28. Wadler, P. Why no-one uses functional languages. *SIGPLAN Notices 33* (Aug. 1998).
29. Wall, L., Christiansen, T., and Orwant, J. *Programming Perl*. O'Reilly, July 2000.