Dr Jekyll and Mr C

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

Abstract

Jekyll is a high level programming language that can be translated losslessly to and from human readable, human editable C. This makes it possible to maintain Jekyll and C versions of the same file, with any changes made to one file being automatically reflected into the other.

By being losslessly inter-translatable with C, Jekyll reduces the switching costs normally associated with moving to a new language. If a programmer does not know Jekyll, or a tool does not understand Jekyll, or the Jekyll compiler ceases to be available, then developers can simply use the C version of a file, rather than the Jekyll version.

Jekyll enhances C with many high level features, including safety, generic types, lambda expressions, and type classes. All features have been carefully designed so that they map elegantly to and from C.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Code generation; D.3.2 [*Programming Languages*]: Applicative (functional) languages

1. Introduction

The programming language community has produced many programming languages that improve on C in useful ways. They have produced languages that are easier to use, easier to understand, safer, more portable, more reusable, and better able to express concurrency. But, despite all this, a large proportion of software projects continue to use C.

Prior work suggests that developers continue to use C because it has built up such a strong ecosystem that the switching costs associated with moving to a new language are too great [35, 20]. 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 maintained in 10 years time

Historically, languages that have achieved widespread adoption have done so using a combination of three techniques:

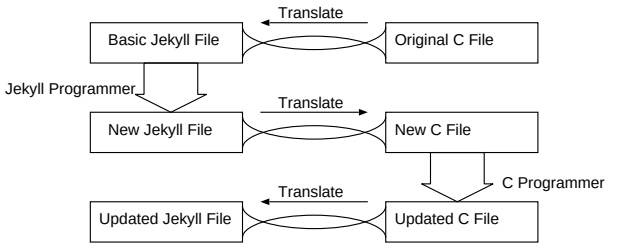

Figure 1. A Jekyll programmer and a C programmer can work on the same file

- Attack a niche in which no other language has built up a strong ecosystem (e.g. Perl [37] for string processing or SQL [6] for database queries).
- Build up a new ecosystem from scratch (e.g. Java [14] and C# [5], using the muscle of large global companies).
- Exploit the ecosystem of an existing language by having some degree of compatibility with that language (e.g. C++ [33] and Objective C [27], which are supersets of C).

1.1 Lossless Translation

Previous languages have exploited C's ecosystem by being interoperable with C, meaning that modules written in the new language can be linked against C modules; or by being supersets of C, meaning that modules written in C can be compiled in the new language.

Our language, Jekyll, goes a step further, 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).

This approach significantly reduces the switching costs of using Jekyll in place of C. It is not necessary for all tools and programmers to know Jekyll, since they can work with 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.

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. This is important as past research has shown that developers will not accept tools that make widespread changes to their programs [4].

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 [29]. Amongst other things, Jekyll supports algebraic data-types, pattern matching, lambda expressions, type classes [12, 36, 16], parametric polymorphism, and full type safety.

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 4.

1.3 Contributions

The contributions of this paper are:

- The design of a new language (Jekyll) that is losslessly inter-translatable with C (Section 2).
- A method of encoding Jekyll into C such that the C can be translated back to Jekyll (Section 3).
- A method for ensuring that translation between C and Jekyll is lossless (Section 4).

We believe that the lossless translation approach described in this paper could be applied more generally. Indeed we are exploring designs for a language that is inter-translatable with COBOL, a language that is inter-translatable with Verilog, and a language that is inter-translatable with Java.

This paper does not attempt to give a detailed description of the Jekyll language and its features. Instead, the aim of this paper is to give an overview of the lossless translation approach that Jekyll uses.

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

2. The Jekyll Language

The Jekyll language is a superset of C, and thus most C programs are valid Jekyll programs. The only cases in which compatibility is not preserved are where the C program uses names that conflict with Jekyll keywords, where the C program uses macros in a way that Jekyll does not understand (Section 2.8), or where the C program uses compiler-specific extensions that Jekyll does not yet support (Section 2.9).

The current version of Jekyll adds the following features to C:

- Tagged unions / Algebraic datatypes (Section 2.1)
- Generic types / Parametric polymorphism (Section 2.2)
- Stack-allocated Closures and Lambda expressions (Section 2.3)
- Extended Initialiser Expressions (Section 2.4)
- Type-classes (Section 2.5)
- Safety (Section 2.6)

In the following sections, we will give an overview of how each of these features work, with reference to the example Jekyll code on the left hand side of Figure 2. The syntax of Jekyll is given formally[1] in Figure 3, in which we show the changes we made to the standard EBNF syntax of ANSI C [17, 31].

2.1 Tagged Unions

Tagged unions make it easy and safe to work with values that can have several types. Jekyll's tagged unions are very similar to Cyclone's [15] tagged unions and ML's datatypes [24]. The syntax for declaring a tagged union is the same as for a struct or union, except that the `tagged` keyword is used. For example, line 11 of Figure 2 defines a tagged union called `List` that is either a `Node` or nothing.

Unlike a normal union, a value of tagged union type knows which of the given types it is. To read from a tagged union, one can use a `switch` statement to extract the correct value from a tagged union. For example line 17 examines c, binding `node` to a `Node` if c is a `NODE`.

2.2 Generic Types

Generic types are perhaps most commonly used for collections. They allow one to define a type that is parameterised over one or more other types. Without generic types, one finds oneself having to repeatedly cast values to and from `void*`, which is not only awkward but potentially unsafe.

Jekyll's generic types are very similar to those provided by C++ [33], ML [24], and later versions of Java [14]. Jekyll allows one to declare a name to be a type variable name using the `typevar` keyword, as used on line 1. Declarations of `struct` and `tagged` types can be parameterised by one or more type variables, as seen on lines 5 and 11. To instantiate a parameterised type, one must specify the type arguments in the declarator, as shown in line 7.

Function arguments and return values can also contain type variables. Line 16 shows a `map` function that can perform an operation on all the elements in a list, regardless of the type of the list's elements.

Unlike C++, Jekyll only allows one to refer to type variables by pointer. The following declaration is thus illegal:

```
struct<a> withint {a data, int extra};
```

A field cannot be of type "`a`" since the size of "`a`" is not known at compile time. Unlike C++, Jekyll generates only one C definition for each Jekyll definition, and thus all sizes must be statically resolvable. While relaxing this restriction would make Jekyll more expressive, it would also make the generated C code less readable.

2.3 Stack-Allocated Closures and Lambda Expressions

A closure is a function that carries an environment with it. Jekyll defines a closure type, distinguished from the standard C function type by the use of a "`!`" symbol before the function arguments (line 27). Like ALGOL [25], Jekyll allocates closure environments on the stack. It is thus illegal for a value of closure type to be either returned or stored in a structure.

Lambda expressions (line 52) allow one to define a closure. In the current version of Jekyll, closures are only permitted as function arguments. The syntax for passing a closure resembles the syntax for `if` and `while` — the closure body is wrapped in braces and is placed after the function call parenthesis. If the closure has any arguments then these are separated from the closure body by a colon.

To return a value, a lambda expression must use the `ret` keyword, rather than `return`. This maintains consistency with `while`,

[1] The syntax of our current implementation is actually a bit broader than given here. We have omitted the syntax for features not described in this paper.

Jekyll Code ⇓

```
typevar a,b,col;
typedef tagged List List;
typedef struct Node Node;

struct<a> Node{
    a *element;
    List<a> *tail;
};

/* a list of elements of type 'a' */
tagged<a> List{
    Node<a> NODE;
    void EMPTY;     /* an empty list */
};

List<b> *listmap(List<a>* c, b *f!(a *x)){
    switch(*c){
        case EMPTY: return new EMPTY;
        case NODE n:
            return new NODE {
                f(n.element),
                listmap(n.tail,f)};
    }
}

interface Mappable col{
    col<b> *map(col<a> *c, b *f!(a *x));
};

implement Mappable tagged List{
    List<b> *map(List<a>* c, b *f!(a *x)){
        return listmap(c,f);
    }
}

interface Num a{
    a* plus(a* x, a* y);
    a* fromInt(int x);
};

implement Num int{
    int* plus(int* x, int* y){return new *x + *y;}
    int* fromInt(int x){return new x;}
}

a* plusint<Num a>(a *x,int y){
    return plus(x,fromInt(y));
}

/* demonstrate lambda expressions */
List<int>* all_plus2(List<int>* l){
    return map(l){int* x: ret plusint(x,2);};
}
```

C Code ⇒ (using macros from Figure 4)

```
#include <jekyll_1.h>
typevar a,b,col;
typedef tagged List List;
typedef struct Node Node;

struct _p(a) Node{
    a *element;
    List _p(a) *tail;
};

/* a list of elements of type 'a' */
tagged _p(a) List{enum{NODE, EMPTY} tag;union{
    Node _p(a) NODE;
    _void (EMPTY);     /* an empty list */
}body;};

List _p(b) *listmap(List _p(a)* c, b *f(_envarg, a *x), void* f_env){
    switch(_match (*c).tag){
        _fwd Node _p(a) n;
        _temp List *_tmp1;
        _temp List *_tmp0;
        case EMPTY:
            _tmp0 =(List*) GC_malloc(sizeof(List));
            (*_tmp0).tag= EMPTY; return _tmp0;
        case NODE: _tagbody n =(*c).body.NODE;
            _tmp1 =(List*) GC_malloc(sizeof(List));
            (*_tmp1).tag= NODE; {
                (*_tmp1).body.NODE.element =
                f(_env f_env, n.element);
                (*_tmp1).body.NODE.tail =
                listmap(n.tail,f,_env NULL);}
            return _tmp1;
    }
}

interface Mappable _p(col){
    col _p(b) *(*map)(_envarg, col _p(a) *c, b *f(_envarg, a *x), void* f_env);
};

implement (Mappable, tagged, List,,);
    List _p(b) *List_map(_dictenv(Mappable, List)
            ,List _p(a)* c, b *f(_envarg, a *x), void* f_env){
        return listmap(c,f,_env NULL);
    }
_dictionary struct {
    List _p(b) * (*map)(_dictenv(Mappable, List)
            ,List _p(a) *c, b *f(_envarg, a *x), void* f_env);
} List_Mappable_dict = {&List_map};

interface Num _p(a){
    a* (*plus)(_envarg, a* x, a* y);
    a* (*fromInt)(_envarg, int x);
};

implement (Num,, int,,);
    int* int_plus(_dictenv(Num, int), int* x, int* y){
        _temp int *_tmp0;
        _tmp0 =(int*) GC_malloc(sizeof(int));
        (*_tmp0) = *x + *y;return _tmp0;}
    int* int_fromInt(_dictenv(Num, int), int x){
        _temp int *_tmp0;
        _tmp0 =(int*) GC_malloc(sizeof(int));
        (*_tmp0) = x;return _tmp0;}
_dictionary struct {
    int * (*plus)(_dictenv(Num, int), int *x, int *y);
    int * (*fromInt)(_dictenv(Num, int), int x);
} int_Num_dict = {&int_plus, &int_fromInt};

a* plusint(_typaram(Num, a), a *x,int y){
    return _argdict(Num,a)->plus(_argenv(Num,a), x,
        _argdict(Num,a)->fromInt(_argenv(Num,a), y));
}

_localfun int *all_plus2_lambda_0(_envarg,int* x){ ret plusint(
    _globdict(Num,int),_env NULL, x,2);}

/* demonstrate lambda expressions */
List _p(int)* all_plus2(List _p(int)* l){
    return List_map(_env NULL, l,
        _g(void*(*)(void*,void*))_localfun &all_plus2_lambda_0,_env NULL);
}
```

Figure 2. Jekyll and C views of an example program — The C code uses the macros from Figure 4

struct-or-union	::=	**tagged** | **struct** | **union**
struct-or-union-specifier	::=	struct-or-union {⟨ *tyvar-name*+/, ⟩}? *identifier* **{** struct-declaration+ **}** | ...
type-specifier	::=	*tyvar-name* | **typevar** | **void** | **char** | **short** | **long** | ...
declarator	::=	{⟨ type-name+/, ⟩}? pointer? direct-declarator
direct-declarator	::=	direct-declarator {⟨ tyvar-context+/, ⟩}? **!**? (parameter-type-list) | ...
tyvar-context	::=	*identifier tyvar-name*
direct-abstract-declarator	::=	{direct-abstract-declarator}? **!**? (parameter-type-list) | ...
expression	::=	init-expression+/,
init-expression	::=	struct-init | tagged-init | alloc-init | unsafe-expression
struct-init	::=	**{** init-expression+/, **}**
tagged-init	::=	*tag-name* init-expression
alloc-init	::=	{**alloc** | **new**} init-expression
unsafe-expression	::=	**unsafe** unsafe-expression | assignment-expression
postfix-expression	::=	postfix-expression (init-expression*/,) lambda-expression* | ...
lambda-expression	::=	**{** parameter-declaration* **:** declaration* statement* **}**
expression-statement	::=	unsafe-expression+/, **;**
labeled-statement	::=	**case** case-pattern **:** statement | *identifier* **:** statement | **default :** statement
jump-statement	::=	**ret** expression | **return** expression | **break** | **continue** | **goto** identifier
case-pattern	::=	constant-expression | *tag-name* | *tag-name identifier*
external-declaration	::=	function-definition | declaration | interface-definition | implement-definition
interface-definition	::=	**interface** *identifier tyvar-name* **{** declaration* **}**
implement-definition	::=	**implement** *identifier type-name* {**:** typevar-context+/,}? **{** function-definition* **}**

Figure 3. Jekyll Extensions to ANSI C Syntax (underline = modification, "+/," = comma separated list)

since `return` inside a `while` loop would return from the top level function, not from a closure.

2.4 Extended Initialiser Expressions

Jekyll introduces several new forms of initialiser expression:

- Allocation initialisers (lines 18 and 20) allocate a new block of data on the heap, and initialise its contents. The "`new`" keyword indicates that the data should be freed by a garbage collector, while the "`alloc`" keyword indicates that the programmer will free the memory manually. This keyword is followed by an initialiser for the data contents.
- Tagged initialisers (lines 18 and 20) create a new tagged value. If the tag body type is non-void then the tag must be followed by an initialiser for the tag body.
- Struct initialisers (line 20) are like struct initialisers in C, except that they can be used as normal expressions.

The Jekyll type checker will attempt to infer the type of the initialised value from its surrounding context. If the type checker cannot infer an expressions's type, then it may be necessary to specify the type of the expression using a cast. For example:

```
(MyStruct) {3,4}
```

Although Jekyll provides syntax that makes it easy to use a garbage collector, Jekyll does not require that a garbage collector be available. If a Jekyll program does not use the `new` keyword, then it is not necessary for the program to be linked against a garbage collection library.

2.5 Type Classes

Type Classes [12, 36, 16] are similar to Java's [14] interfaces in that they allow one to define collections of operations that can be implemented for many types; however they differ in several important respects:

- Type classes allow one to implement interfaces for types that have already been defined. This makes it easy to use interfaces with types that are defined in a library, or in parts of the program that should not be modified.
- Type classes can be implemented using dictionary passing (Section 3.3), rather than using a vtable. This avoids the need to change the data-representation of objects or use a special allocator.
- Type class methods can refer to the implementing type any number of times in their function arguments and return type, rather than having an implicit `this` argument. This can make some programming idioms simpler [36].

Jekyll implements the full type class system from Haskell98 [29], including constructor classes. We have however renamed most of the type class concepts in order to make them more approachable for programmers who are familiar with languages such as C++ and Java. In particular "class" is now called "interface" and "instance" is now called "implement".

The syntax for defining a new interface is similar to Java. Line 36 defines an interface `Num` for types that implement a `plus` operation and can be converted from an integer. Line 41 shows how we can implement `Num` for `int`. Note that, unlike Java, the implementation of `Num` is separated from the definition of `int` (which is built in).

Line 26 defines an interface `Mappable` for types that implement a `map` function. This is a constructor class [36] – the type `col` must be passed a type argument (the element type) in order to make a complete type.

Line 46 defines a function that can add an `int` to any type provided that the type implements `Num`. The call to `fromInt` has its type resolved by its return type, which is inferred from the type of the first argument to `plus`.

2.6 Safety

Like C# [5], Jekyll requires that all unsafe expressions be marked as unsafe, by preceding them with the `unsafe` keyword. This tells

the Jekyll translator that the programmer is aware that the expression is potentially unsafe and that the Jekyll translator should not warn the programmer about that expression. If the unsafe keyword is omitted, then Jekyll will warn about potentially unsafe expressions, including those that use pointer arithmetic, unsafe casts, and unchecked array accesses.

2.7 Limitations

In the design of Jekyll, there is an unavoidable trade-off between making the language elegant, and making it inter-translatable with elegant C. As part of this trade-off, we have decided to impose the following limitations on Jekyll:

- Type variables cannot be referred to other than through pointers (Section 2.2).
- Jekyll's type safety can be broken if one links two Jekyll object files that define symbols that have the same name but different types. This problem can be easily fixed by using a build tool that checks for this issue.
- In some places Jekyll's syntax is restricted by the need to be a superset of C.
- Jekyll programmers cannot assume the presence of tail recursion optimisation.
- Jekyll does not make evaluation order guarantees beyond those provided by C.

2.8 Macros

The current implementation has only a very simple understanding of the C preprocessor. If a macro is a simple constant definition then Jekyll will handle it correctly without needing further direction. If a macro is a function, or a complex expression whose type cannot easily be inferred, then it is necessary to explicitly specify a type for the macro, using a line such as the following:

```
unsafe macrotype int max(int,int);
```

In the future we intend to extend Jekyll's macro support to understand more uses of macros without requiring manual direction. We are currently investigating a design that expands macros during the lexer, and then re-collapses them during the pretty-printer. When implemented, this should allow Jekyll to understand all uses of C macros. An alternative would be for use Macroscope [21].

2.9 Compiler-Specific Extensions

The current version of Jekyll supports all the features in ANSI C, but many C program make use of compiler-specific extensions, particularly those provided by GNU C [30]. Jekyll currently supports some of these extensions, and it is our intention to eventually support all of the extensions in common use.

When a programmer uses a feature in a Jekyll program that is supported by some C compilers but not all, the Jekyll translator has a choice. It can either pass that feature through into the generated C code and rely on the the C compiler to support it, or it can encode the feature using more commonly supported features. The current implementation passes some features through, and encodes some others. In the future, we intend to offer a choice of multiple C encodings that assume the availability of different compiler-specific extensions (Section 3.5).

2.10 Future Features

While Jekyll is already a very usable language, there are a number of additional features that we plan to add in the near future that will make it significantly more powerful, including the following:

- Concurrency features

```
#define typevar       typedef void
#define tagged        struct
#define interface     struct
#define ret           return
#define unsafe        /* nothing */
#define macrotype(x)  /* nothing */
#define _p(x)         /* nothing */
#define _match        /* nothing */
#define _fwd          /* nothing */
#define _temp         /* nothing */
#define _dictionary   /* nothing */
#define _localfun     /* nothing */
#define _void(x)      /* nothing */
#define _tagbody      /* nothing */
#define _env          /* nothing */
#define _g(x)         (x)
#define _envarg       void* _denv
#define _argdict(iface,ty) ty##_##iface##_dict
#define _argenv(iface,ty) ty##_##iface##_env
#define _typaram(iface,ty) \
    struct iface* _argdict(iface,ty), \
    void* _argenv(iface,ty)
#define _dictenv(iface,ty) \
    struct _argenv(iface,ty)* _denv
#define implement(iface,kind,ty,tyargs,context) \
    struct _argenv(iface,ty) { context ;};
#define _needs(iface,ty) \
    struct iface* _argdict(iface,ty); \
    void* _argenv(iface,ty);
#define _globdict(iface,ty) \
    ((struct iface*)&_argdict(iface,ty))
```

Figure 4. The contents of `jekyll_1.h`

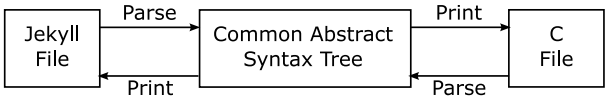

Figure 5. Jekyll and C code parses to a Common AST

- Safe pointers to stack-allocated variables (which are currently treated as unsafe)
- Existential types / Object-carried vtables
- Heap-allocated closures
- Distinguishing between nullable and non-nullable pointers
- User-defined operators
- User-defined automatic coercions
- Curried function application
- More expressive pattern matching support
- Effect types
- Removing the pointer restriction on type variables (Section 2.2)
- More complete macro support (Section 2.8)
- A library of useful types, including safe arrays and safe strings

We also plan to improve the format of the generated C code to make it more readable.

3. Encoding Jekyll Features in C

The right hand side of Figure 2 shows the C code that results when we translate our Jekyll example to C. The Jekyll and C versions of the program parse to the same AST and each can be exactly

recreated from the other (Figure 5). We can note the following properties in the C encoding:

- Special macros (defined in Figure 4) are used to encode Jekyll features such that Jekyll's C parser can decode them
- The C file may place some expressions in a different order to the Jekyll file (e.g. line 26).
- Some functions have additional *dictionary* arguments (e.g. C line 69)
- Some functions have additional *environment* arguments (e.g. C line 74)
- Both files have the same layout and comments
- The C file starts with `#include <jekyll_1.h>`

In the following sections we will discuss each of these topics further.

3.1 Special Macros

If a Jekyll statement only uses features that are present in C then the corresponding C statement will be identical to the Jekyll statement.

If a Jekyll statement uses features that are not present in C then the Jekyll-specific features are encoded in C using a number of special macros. The primary purpose of these macros is to allow Jekyll's C parser to recognise the Jekyll feature that the C code is implementing. The C definitions of these macros are given in `jekyll_1.h`, the source of which is given in Figure 4.

The meanings of these macros should be fairly self-evident. `typevar`, `tagged`, `interface`, `ret` and `_match` tell Jekyll that the corresponding Jekyll feature is being used and that the following C code should be interpreted accordingly. `_fwd`, `_temp` and `_localfun` provide forward declarations of temporary variables or lambda expressions. Most of the other macros are used to implement dictionary passing (Section 3.3) and environment passing (Section 3.4).

To avoid name clashes, all Jekyll macros are considered to be reserved words in Jekyll, and the translator disallows variable names that clash with generated temporary names.

3.2 Code Reordering

In some cases, the C file places expressions in a different order to the Jekyll file. For example lambda expressions will be defined before their use (Jekyll line 52 and C line 75), and initialisers will be executed before their results are used (Jekyll line 20 and C line 26).

Such code reordering is often necessary in order to produce an elegant encoding into C. This reordering is one of the major factors complicating the implementation of lossless translation (Section 4).

3.3 Dictionary Passing

Like GHC [28], we implement Type Classes using Dictionary Passing. If a function requires that a type variable implements an interface (e.g. Jekyll line 46 in Figure 2) we add an additional *dictionary* argument to the C version of the function (e.g. C line 69 in Figure 2). This dictionary contains pointers to the functions that implement that interface for that type variable.

This is in contrast to languages such as C++ [33] and Java [14] in which method implementations are obtained from the data, rather than the caller.

3.4 Environment Arguments

In some cases, the C version of a function will have extra *environment* arguments that were not present in the Jekyll version. For example lambda expressions are passed an environment containing definitions for any free variables (e.g. C line 74 in Figure 2).

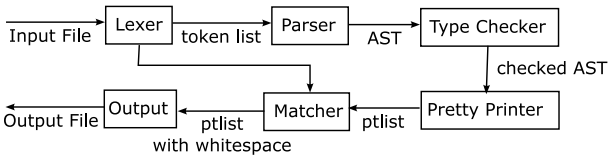

Figure 6. Top level structure of the Jekyll translator. (Types given in Section 4.4)

Such additional arguments are tagged with Jekyll macros so that the Jekyll translator can recognise them when translating back to Jekyll code.

3.5 The Jekyll Include Line

The first line of the C file `#include`s a Jekyll header. This line serves two purposes: *(i)* it tells the C compiler what the Jekyll macros mean, and *(ii)* the name of the included file tells the Jekyll translator which C encoding has been used for the file (in this case version 1).

Although our current implementation only supports one encoding into C, we plan to support a number of different encodings. These different encodings would use different coding conventions and make use of different language extensions that might be available in a C compiler. For example an encoding that relied on GCC extensions would need less code reordering, and an encoding that targeted C++ could make use of C++ features.

4. Lossless Translation

As we stated in Section 1.1, the translation between Jekyll and C is guaranteed to be lossless. By this we mean that if a Jekyll file is translated to C and back again, the resulting file is guaranteed to be bit-for-bit identical to the original file, preserving layout, comments, and everything else. In this section we explain how we do this.

4.1 Token Twinning

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*.

An example of twins is illustrated in the top half of Figure 7 which shows example token twins in a section of code from Figure 2. Grayed out C tokens are those that are untwinned.

A token's twin may be a different string. For example "new" is twinned with "GC_malloc" in Figure 7.

4.2 Whitespace

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;
```

If two tokens are twins, then they have the same whitespace. This allows us to store Jekyll whitespace in the C file such that it can be retrieved when the C file is translated back to Jekyll. The links at the bottom of Figure 7 give examples of cases where Jekyll whitespace is carried over into C.

The whitespace for an untwinned token is chosen by the translator. For example the third line in Figure 7 has been indented to match the sixth line.

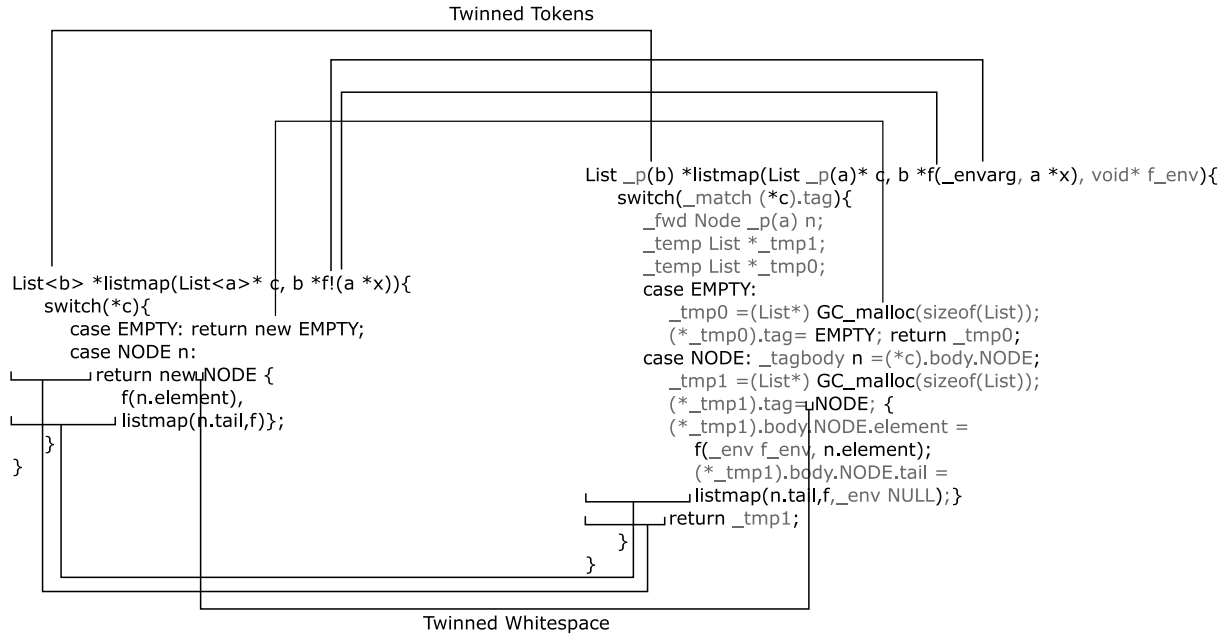

Figure 7. How tokens and whitespace map between Jekyll and C versions of a file

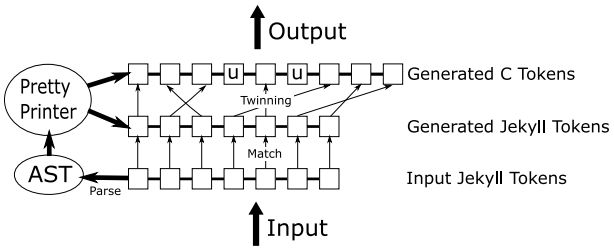

Figure 8. Whitespace from from Jekyll to C (u = untwinned)

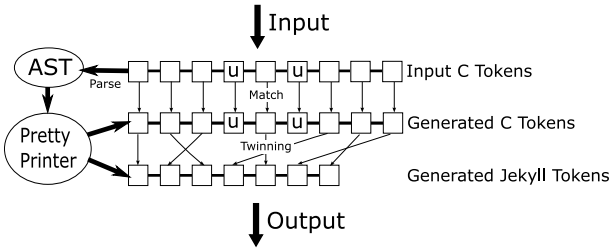

Figure 9. Whitespace flow from C to Jekyll (u = untwinned)

4.3 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. Given such twinned token lists, the translator can follow the procedure in Figures 6 and 8 to translate a file from Jekyll to C:

- Lex the Jekyll file, producing a token list annotated with whitespace.
- Parse the token list, giving an abstract syntax tree (AST).
- Typecheck the abstract syntax tree.
- Pretty print the abstract syntax tree, giving token lists for Jekyll and C, with each Jekyll token twinned with a C token — the generated Jekyll token list should be identical to the input Jekyll token list, except that it has no whitespace information.
- Match the generated Jekyll token list with the parsed token list, and thus determine the whitespace for each generated Jekyll token — the generated Jekyll token list is now identical to the input token list, including whitespace.
- Using the twinning information, determine the whitespace for each twinned C token.
- Output the C tokens, complete with whitespace taken from the Jekyll

The procedure for converting C to Jekyll (Figure 9) is essentially the same. We match the generated C code to the parsed C code to obtain the whitespace for the generated Jekyll code. It should be easy to see that, provided the output parses to the same AST as the input, Figure 9 is the inverse of Figure 8.

When converting C to Jekyll it is possible that the process of matching the pretty printed tokens to the parsed tokens might fail. We discuss this in Section 4.8.

This twinned-pretty-printing approach has several advantages over manually linking input tokens to output tokens. Firstly, the parser and pretty printer do not have to think about the tokens from the input list, since they are dealt with in the matching stage. Secondly, and perhaps more importantly, the parser does not have to worry about checking the correctness of C boilerplate code, since it will be checked by the matcher (see Section 4.8).

4.4 Twinning Combinators

We use a set of pretty printing combinators to allow us to easily write pretty printers that produce twinned token lists for Jekyll and C.

```
type token = {
    body : string; mutable white : string;
    mutable twin : token option}
type ptlist = token list * token list

let mk s = {body = s; white = ""; twin = None}
let link t1 t2 = t1.twin <- Some t2;
                 t2.twin <- Some t1

let (<+>) (j1,c1) (j2,c2) = (j1 @ j2,c1 @ c2)
let empty = ([],[])
let twin j c = let jt,ct = mk j, mk c in
                 link jt ct; ([jt],[ct])
let onlyc s = ([],[mk s])
let extract_c (j,c) = ([],c)
let extract_jkl (j,c) = (j,[])
```

Figure 10. Simplified Implementation of Twinned Combinators

The combinators we use are loosely based on those of Hughes [13] and Wadler [34], with extensions to support the generation of twinned token lists. A simplified O'Caml [18] implementation of these operations is given in Figure 10. Their types are as follows:

```
val <+>         : ptlist -> ptlist -> ptlist
val empty       : ptlist
val twin        : string -> string -> ptlist
val onlyc       : string -> ptlist
val extract_c   : ptlist -> ptlist
val extract_jkl : ptlist -> ptlist
```

The type `ptlist` consists of two `token` lists, one for Jekyll and one for C. A `token` contains a body string, some whitespace, and, unless it is untwinned, a reference to its twin.

The combinators are as follows:

- `<+>`[2] concatenates two `ptlists` together.
- `empty` is a `ptlist` that contains no tokens
- `twin` generates twinned tokens. The first argument is the string for the Jekyll token and the second argument is the string for the C token. For convenience, we define `str x = twin x x`.
- `onlyc` generates an untwinned token. This token is only emitted into the C list.
- `extract_c` and `extract_jkl` are used to reorder tokens. `extract_c` extracts the C part of a `ptlist` ignoring the Jekyll part. Similarly, `extract_jkl` extracts the Jekyll part and ignores the C part. Every call to `extract_c` should be accompanied by a call to `extract_jkl` with the same argument.

As an example, consider the following O'Caml expression:

```
let x = twin "a" "b" in
extract_c x <+> twin "c" "d" <+> extract_jkl x
```

This will produce the Jekyll list "`[c,a]`", and the C list "`[b,d]`", where "`a`" and "`b`" are twins and "`c`" and "`d`" are twins.

Our full combinator library contains a larger, richer set of combinators. In particular, it contains a number of combinators that are used to specify what whitespace should be attached to untwinned C tokens.

[2] Hughes calls his operator `<>`. We could not use this name as it means inequality in OCaml [18].

```
let token_match t_in t_gen =
    if t_in.body <> t_gen.body then warn ();
    if t_gen.white <> t_in.white
         && t_gen.twin = None then warn ();
    t_gen.white <- t_in.white

let white_flow t = match t.twin with
    | Some p -> p.white <- t.white
    | None -> ()

let jkl_to_c (j,c) input =
    iter2 token_match input j;
    iter white_flow j;
    c
let c_to_jkl (j,c) input =
    iter2 token_match input c;
    iter white_flow c;
    j
```

Figure 11. Simplified Implementation of Translation

4.5 Doing the Translation

The actual translation is done by the functions `c_to_jkl` and `jkl_to_c`, given in Figure 11. Their O'Caml signatures are as follows:

```
val jkl_to_c  : ptlist -> token list -> token list
val c_to_jkl  : ptlist -> token list -> token list
```

`jkl_to_c` and `c_to_jkl` use a `ptlist` to translate an input token list into an output token list. `jkl_to_c` translates a Jekyll to C and `c_to_jkl` translates C to Jekyll. They do this using the procedure given in Section 4.3: input tokens are matched against generated tokens (`token_match`) and then whitespace flows through the twin links to output tokens (`white_flow`).

The definitions of these functions make use of the standard O'Caml [18] functions `iter` and `iter2`. `iter` applies a function to all elements of a list, and `iter2` applies a function to all elements of two lists – e.g. both first elements, then both second elements, etc.

4.6 Whitespace for Untwinned Tokens

The definition of `onlyc` in Figure 10 gives untwinned tokens empty whitespace. In our real implementation the translator uses additional combinators to specify what whitespace should be attached to untwinned tokens. For example the `newline` combinator specifies a point at which a new line should start, and the `maybebreak` combinator specifies a point at which a new line should start if the line would otherwise be too long. When a new line is started, the translator uses the surrounding lines to determine how much the line should be indented. Examples of this can be seen in Figure 2.

4.7 Mismatched Whitespace for Untwinned Tokens

The definition of `token_match` in Figure 10 will warn if asked to match an untwinned C token with a generated token that has different whitespace. This warning indicates the fact that the whitespace attached to this token will not be preserved, since it is different to the whitespace that will be chosen by the translator when the file is translated back to C. The translator will draw particular attention to lost whitespace that contains comments. If there are no warnings then the translation is guaranteed to have been lossless.

In practice, we have not found this potential loss of whitespace to be a problem. Whitespace can only be lost if a programmer has actively edited C code that encodes Jekyll features. This means that

any changes to whitespace will be in areas of the program in which the programmer has been working and thus version control systems will not show spurious changes.

4.8 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 with equivalent meaning. In particular, a C file may contain sections of boilerplate code (e.g. line 45 in Figure 2). If this boilerplate is modified incorrectly then it will not correspond to a valid Jekyll construct.

Our token matching approach 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 4.3) to check that the parsed C tokens are the same as those that it would pretty print for the parsed abstract syntax tree.

If the matching stage finds that the input tokens do not match the generated tokens, it will warn the user. This will result in the user seeing a warning message such as the following:

```
File examples/demo.c : 66 characters 12 - 15
Malformed input file: expected fromInt but given fInt

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

5. Related Work

Many of the ideas in Jekyll are similar to ideas that have appeared in previous work. Many other people have developed languages with similar features, languages that extend C, languages that are embedded in C, compilers that use C as a back end, tools that translate one language into another, tools that transform programs while preserving formatting and comments, or tools that present multiple representations of a program.

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

5.1 Languages with Similar Features

All the language features present in Jekyll have appeared previously in other languages. Jekyll's type system has been largely lifted from Haskell [29], including type classes [12, 36, 16], parametric polymorphism, algebraic datatypes, and closures. Jekyll's stack-allocated closures are similar to those of ALGOL [25]. Many of Jekyll's features also appear in O'Caml [18] and other functional languages. Recently, Microsoft's LINQ [1] project has extended C# with similar features, including lambda expressions, and separation of implementation from types.

Jekyll's advantage over these other languages is that it can be losslessly translated to and from C. Jekyll's disadvantage is that it is notably less elegant than these other languages, due to the requirement that it be losslessly inter-translatable with C.

5.2 Languages that extend C

Many languages have extended C with new features. Cyclone [15], Vault [7], Ivy [2], C++ [33], Objective C [27] and many others all add useful new features to the core C language.

While existing C code is usually valid in these languages, any use of new features will prevent the program being a valid C program. As with Section 5.1, these languages are often more elegant than Jekyll since they do not need to be inter-translatable with C.

5.3 Languages embedded in C

Several authors have designed systems that use similar techniques to those described in Section 3 to embed extra features into C programs. CCured [26] 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++ [22] 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.

5.4 C as a Back End

Many compilers for high-level languages translate to C as part of their compilation process. Examples of this include GHC [28] and CFront [32]. Unlike Jekyll, the generated C is not intended to be human readable, and the translation is not intended to be reversible.

5.5 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 [11] translates Pascal to C.

Like Jekyll, the resulting program is expected to be readable. Unlike Jekyll, the translation is expected to be a one-off one-way event, and so there is no need to exactly preserve formatting or make the transformation reversible.

5.6 Format Preserving Code Transformation

We are not the first to attempt to transform a program while preserving formatting. Many tools for program refactoring [23, 9], such as HaRe [19] and Eclipse [8] make changes to a program while preserving the program formatting. Like Jekyll, such tools typically work simultaneously with both an abstract syntax tree and a token list [19].

Unlike Jekyll, refactoring editors do not need to map one language into another, or need to guarantee that their transformations are reversible. Refactoring editors thus have no need for the twinned-token techniques we described in Section 4.

5.7 Language Workbenches

A Language Workbench [10] is an integrated development environment (IDE) in which a programmer writes software using a number of user-defined domain specific languages (DSLs). In some cases it may be possible for the IDE to represent the same abstract syntax tree using several different DSLs (e.g. graphical and Java representations of a GUI).

Unlike Jekyll, it is not necessary to preserve layout information, since it is assumed that DSLs are edited graphically or that there is a canonical correct textual representation. There is also no requirement that each language contain all information present in the others, since the persistent AST is the authoritative representation, not the program text.

6. Conclusions

By being losslessly inter-translatable with C, Jekyll makes it practical for developers to move away from C. Lossless translation allows developers to make use of C programmers, C tools, and C libraries in a way that would not be practical otherwise. Moreover lossless translation avoids 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.

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.

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 COBOL, 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`

The example shown in Figure 2 can be found in the file "examples/dr_jekyll_examples.jkl" in the Jekyll distribution.

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, Claus Reinke, Richard Sharp, and Simon Thompson for providing useful suggestions.

References

[1] BOX, D., AND HEJLSBERG, A. The LINQ project. Microsoft MSDN Library, Sept. 2005.

[2] 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).

[3] FOR_C: Converts FORTRAN into readable, maintainable C code. http://www.cobalt-blue.com.

[4] CORDY, J. R. Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In *Proceedings of the IEEE Workshop on Program Comprehension (IWPC'03)* (May 2003).

[5] *C# Language Specification*. ECMA, June 2005.

[6] DATE, C. J. *A Guide to the SQL Standard*. Addison-Wesley, 1986.

[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] The Eclipse Project. `http://www.eclipse.org`.

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

[10] FOWLER, M. Language workbenches: The killer-app for domain specific languages. `http://www.martinfowler.com/articles/languageWorkbench.html`, June 2005.

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

[12] 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).

[13] HUGHES, J. The Design of a Pretty-printing Library. In *Advanced Functional Programming* (1995), J. Jeuring and E. Meijer, Eds., Springer Verlag, LNCS 925, pp. 53–96.

[14] *The Java Language Specification, third edition*. Sun Microsystems, 2005.

[15] 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).

[16] JONES, S. P., JONES, M., AND MEIJER, E. Type classes: exploring the design space. In *Proceedings of the ACM Haskell Workshop* (1997).

[17] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language*, second ed. Prentice Hall, Englewood Cliffs, NJ, 1988.

[18] LEROY, X., DOLIGEZ, D., GARRIGUE, J., REMY, D., AND VOUILLON, J. *The Objective Caml system release 3.08*, July 2004.

[19] LI, H., THOMPSON, S., AND REINKE, C. The Haskell refactorer, HaRe and its api. In *Proceedings of the fifth workshop on Language Descriptions, Tools and Applications (LDTA'05)* (2005).

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

[21] MCCLOSKEY, B., AND BREWER, E. ASTEC: A new approach to refactoring c. In *Proceedings of the 10th European Software Engineering Conference* (Sept. 2005).

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

[23] MENS, T., AND TOURWE, T. A survey of software refactoring. *IEEE Transactions of Software Engineering 30*, 2 (Feb. 2004).

[24] MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[25] NAUR, P., AND BACKUS, J. W. Revised report on the algorithmic language ALGOL. *Communications of the ACM 6* (Jan. 1963).

[26] 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).

[27] *The Objective C Programming Language*. Apple, Oct. 2005.

[28] 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.

[29] 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.

[30] STALLMAN, R. M. *Using and Porting GNU CC (Version 2.0)*. Free Software Foundation, Feb. 1992.

[31] STONE, J. D. The syntax of C in backus-naur form. `http://www.math.grin.edu/~stone/courses/languages/C-syntax.xhtml`, Jan. 2002.

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

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

[34] WADLER, P. A prettier printer. In *The Fun of Programming* (Apr. 1997).

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

[36] WADLER, P., AND BLOTT, S. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the ACM Symposium on Principles of Programming Languages* (1989).

[37] WALL, L., CHRISTIANSEN, T., AND ORWANT, J. *Programming Perl*. O'Reilly, July 2000.