Data Structures Used in the OpenBSD Kernel
Master's Thesis

June 1, 2012

Author
Adam Martin Britt (311083-ambr)

Supervisor
Joseph Kiniry

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Terms and Notation . . . 2

2 Interface Complexity **2**
- 2.1 Metric Choices . . . 3
- 2.2 Primitive (Built-In) Data Types . . . 3
  - 2.2.1 Void . . . 4
  - 2.2.2 Char . . . 4
  - 2.2.3 Integral Numbers . . . 5
  - 2.2.4 Floating-Point Formats . . . 5
  - 2.2.5 _Bool . . . 5
  - 2.2.6 _Complex . . . 6
- 2.3 Signedness, Qualifiers, Storage Classes and Enum . . . 6
- 2.4 Pointers . . . 6
- 2.5 Typedef . . . 7
- 2.6 Composition Rules . . . 7
  - 2.6.1 Struct . . . 7
  - 2.6.2 Union . . . 8
  - 2.6.3 Functions . . . 9
  - 2.6.4 Array . . . 10
- 2.7 Macros . . . 10
- 2.8 Combining Elements in an API . . . 11

3 Interface Complexity in Practice **11**
- 3.1 Doubly-Linked List . . . 11
  - 3.1.1 Creation . . . 11
  - 3.1.2 Inserting . . . 12
  - 3.1.3 Deleting . . . 12
  - 3.1.4 Iterating . . . 13

4 Hypotheses of Interface Complexity **13**
- 4.1 Ambiguous Primitive Type Size . . . 13
- 4.2 Inverse Relation with Cyclomatic Complexity . . . 13
- 4.3 Measurement of a Language's Expressiveness . . . 14
- 4.4 Low Numbers of Arguments . . . 14
- 4.5 Currying . . . 14

5 A Tool for Calculating Interface Complexity **15**

6 Resource Complexities **15**

7 Implementation Complexity **16**
7.1 Cyclomatic Complexity 16
7.2 Kolmogorov Complexity 17
7.3 Halstead Complexity Metrics 17

8 Britt API Complexity **18**
8.1 Doubly-Linked List 19

9 Conclusion **20**
9.1 Future Work 20

A Primitive Data Types and Composition Rules **23**

B Interface Complexity Tool **26**
B.1 cparse.py 26
B.2 britt.py 27
B.3 utils.py 30

C C Sizes **32**

D libds **32**
D.1 list.h 32
D.2 list.c 35

Abstract

A common approach to data structure analysis, and interfaces in general, is to look at its running time and memory footprint. These are two important aspects in analysing and using interfaces. Proposed is a new method of analysing an interface based on type, as well a new metric which combines all these different perspectives.

Interface Complexity is a look at the complexity of an interface from the viewpoint of the user. Any library developer must decide how much of their library to expose, balancing complexity and ease of use. Proposed is a method to quantify interface complexity based on type signatures.

The inspiration for this new approach was born out of studying the data structures used by the OpenBSD kernel. OpenBSD source code, along with other samples, are used for various examples.

1 Introduction

In software development, abstraction is a necessity. Whether it be a programming language, a file transfer over a network or a data structure, implementation details are hidden. This is a good thing. A Java developer working on business logic does not need to know the details of displaying pixels to a screen. The developer can focus on his problem and not worry about the thousands of other systems he interacts with to accomplish his task.

However, there are times when a deeper understanding of a particular abstraction is necessary. Knowing the internal workings of an abstraction one works with can be the difference between solving the task at hand or not, or being satisfied with the solution or not.

A simple example is the use of a singly-linked list, where appending an element requires linear time but prepending an element takes constant time[8, p. 254]. If a developer was unaware of the difference between appending and prepending, he could easily write code that was needlessly inefficient.

Unfortunately, modern software is built upon so many layers of abstractions and has so many capabilities, that no single person can have adequate understanding of all systems he works with. Furthermore, it can be overwhelming to identify which abstraction deserve greater attention. This is the motivation behind interface complexity.

Alone, interface complexity assists in identifying potential poor use of abstractions. Another use is to provide an indication of the maintainability of a code base. These are important considerations to make when developing software, but they are by no means the only considerations. This is where Britt API Complexity comes in. The Britt API Complexity takes interface complexity along with computational complexity, memory overhead, and an implementation complexity to provide a developer with a broader picture of the abstractions he works with.

1.1 Motivation

The idea to create a metric based on types comes from the problem of analysing data structures in OpenBSD. The first dilemma of analysing a data structure was the question "What is a proper way to analyse a data structure?"

There are many existing complexities and metrics that exist which try to answer this question. All of them come from a particular point-of-view. Is the running time important? Memory usage? Ease of use or maintainability?

In the context of OpenBSD, it was clear that all of these were important.

For this reason, the focus became developing a good way to include many different perspectives in the analysis of not just data structures but of code in general.

To that end, an evaluation of the current methods of analysis was made, as well as considerations for new methods. The result is that a new metric has been defined that is designed to be used together with other existing metrics. The new metric is named interface complexity, where it measures complexity based on type.

1.2 Terms and Notation

All of the examples are in C. A term that may be unfamiliar is an external declaration, which is anything that can be exported across translation units. Examples are function prototypes, structs, unions and typedefs. A translation unit is what the C compiler is fed, in other words, a preprocessed file.

The notation $|i|$ will be used to signify the complexity of i.

2 Interface Complexity

An interface is an abstraction that is the point of interaction between components. An interface can exist independent of an implementation, which means an interface can have many different implementations. The interface hides the complexity of the implementation but sometimes knowledge of the implementation is required to use the component correctly or efficiently.

Interface Complexity is a metric designed to quantify the complexity of an interface based on types, so that quick assessment can be made about the necessity of understanding the implementation. Types offer information about the components they are part of. Quantifying this information provides the developer with a hint as to how much he needs to know to use the component effectively.

The metric is defined from the bottom up. First, all primitive types are defined, based on the physical space they occupy and their deviation from the default. More complex types are created using a set of composition rules. This combination of primitives and composition rules means that any type's complexity can be measured recursively. Once a type signature has been measured, it can be compared with other type signatures.

There is a distinction between types and new definitions of types. Defining a new type is the process of adding a new complexity to an environment.

Once the type is defined, it can be used to define new types. Typedef, #define, struct and union are operations that define new types.

Interface Complexity can serve at least two useful purposes. First, it can be used by a developer to get an idea of what libraries might be worth becoming acquainted with. Secondly, it can be used to assist in writing simpler, more maintainable code.

While much effort is put into justifying the complexities based on theory, the complexity is still attempting to measure something subjective. The goal is to have as much of the metric grounded as possible, but there are some instances where intuition and judgment prevail or where it is the only option.

2.1 Metric Choices

There are many decisions and variables to consider when designing a metric. For this metric, the driving concern is "usefulness to a typical programmer." To that end, emphasis is on conciseness, simplicity and familiarity. The complexity is mapped to $\mathbb{R}^+$, which developers should be comfortable working with.

A positive real means that the metric has a total order. This is an advantage because each component of an interface can be measured and subsequently compared with one other, whether it be a primitive type or the entire interface.

A positive real is also dense. This works well because of the use of logarithms in the metric and it also helps with classifying complexity classes. Initially, the metric used positive integers, which quickly grew very large. The thought was that integers were easier for a developer to relate to, but comparing a 8,123,231,589,532 to 1080 does not yield much useful information.

2.2 Primitive (Built-In) Data Types

The languages built-in data types are the building blocks of interface complexity. Each built-in needs to be explicitly defined and will be used as literals in the composition rules in section 2.6.

In many cases, the values for these primitives are ambiguous as defined in the C standard library. The following descriptions are based on a 32-bit OpenBSD operating system and uses the GNU C Compiler(gcc). For proper values in a different environment, refer to the implementation of the Standard C Library of that environment. The most notable headers to investigate are limits.h, float.h, and stdint.h.

It should be noted that different default sizes, such as a 32-bit integer versus a 64-bit integer will result in different complexities. The hypothesis discussed in section 4.1 goes into about the ramifications this has on the total ordering. A utility to help determine the sizes on a local environment is described in the appendix C.

The primitives are all defined by the logarithm of the total number of values that can be expressed by that data type. The following table shows the complexities for all of the primitive data types.

primitive	complexity	result
void	c	c
char	$log_2(2^8)$	8
short	$log_2(2^{16})$	16
int	$log_2(2^{32})$	32
long	$log_2(2^{32})$	32
long long	$log_2(2^{64})$	64
float	$log_2(2^{32} \cdot 2^4)$	36
double	$log_2(2^{64} \cdot 2^8)$	72
long double	$log_2(2^{96} * 2^{12})$	108
_Bool	$log_2(2)$	1
_Complex		

2.2.1 Void

In C, the simplest primitive data type is void. This is represented with zero bytes and can only be used as a function return type. Therefore, void is assigned the lowest value, which will be referred to as c. Void corresponds to the Unit type in type theory [14, p. 119].

It should be noted void* is something very different than a plain void in C. Void* is is more like the bottom type ($\perp$) of a type lattice.

2.2.2 Char

A char is one byte in size and is the simplest primitive data type. The standard C definition states:

"The implementation shall define char to have the same range, representation, and behavior as either signed char or unsigned char. . . char is a separate type from the other two and is not compatible with either." [6, p 35]

In the environment used for all tests, the default behavior of char is signed. Therefore, char is assigned the same value as signed char and the unsigned

version is assigned a greater complexity, which is described in section 2.3.

A char uses eight bits, so its complexity is defined as $log_2(2^8)$.

2.2.3 Integral Numbers

The C standard states that both short and int use a minimum of two bytes. In the environment described in the experiments of section 3, the environment uses two bytes for short and four bytes for int.

The complexity of these types is the log_2 of the total number of values that can be represented. This gives a complexity of $log_2(2^{16}) = 16$ for short and $log_2(2^{32}) = 32$ for int. A long must be a minimum of four bytes, which is what the test environment uses. The complexity is $log_2(2^{32}) = 32$. Lastly, a long long, is defined as a minimum of 8 bytes, which has a complexity of $log_2(2^{64}) = 64$.

2.2.4 Floating-Point Formats

The implementation of float is not explicitly defined in the c99 specification but the most common implementation is IEEE 754 single-precision binary floating-point format, which is also referred to as *single* or *binary32*.

Based on personal experience, floats are more difficult to work with than integers. Firstly, most people are more experienced working with integers than they are working with reals. Secondly, since floats cannot precisely represent all reals, there are occasional unexpected results to the unaware. Therefore, floats are assigned an additional complexity, to make it more complex than integral numbers.

The implementation for float uses four bytes, which yields a complexity of $log_2(2^{8*4} * 2^4)$. A double is implemented in the same way as float but uses 8 bytes for increased precision, so a double is defined as $log_2(2^{8*8} * 2^8)$. Finally, a long double is uses 12 bytes for increased precision, thus long float is defined as $log_2(2^{8*12} * 2^{12})$.

2.2.5 _Bool

_Bool was not introduced until the c99 standard, so many code bases have their own definition of _Bool. _Bool is defined using a set of macros defined in stdbool.h. The definition states that _Bool must be large enough to hold the values 0 or 1. In the library used in the experiments, this is implemented using a char. The number of possibilities is not enforced, which means that _Bool can be assigned any value a char can. complexity of _Bool is $log_2(2^1) = 1$.

2.2.6 _Complex

_Complex comes in three types; float, double, and long double. _Complex numbers are stored as a two-element array, corresponding to its real type, where the first element is equal to its real part and the second element is equal to its imaginary part. Therefore, complex is treated as a special case of an array declarator, giving a complexity of:

$(|i| + 1)$ where i will be a float, double or long double.

2.3 Signedness, Qualifiers, Storage Classes and Enum

Integral data types are signed by default [6, p. 99], therefore adding signedness to an integral data type has no change on it's complexity. An unsigned integral data type increases the complexity by a factor of 1.5. The idea is to give additional complexity for diverging from the default, but without increasing complexity by an order of magnitude. The use of log_2 means that it must be a factor between 1 and 2.

There are three type qualifiers in C [6, p. 108]. They are *const, restrict,* and *volatile*. Restrict and volatile are used to assist the compiler in optimization. Const is used to make a read only variable. All of these changes modify the original behavior but not the type so the complexity is increased by a addition constant.

Storage classes define either where and how a variable is stored. The storage classes in C are auto, register, static and extern. These classes do not change the type but do change behavior, therefore the complexity is increased by an addition constant relative to the change in default behavior.

Enum is implemented as an int and there are no compile-time or run-time checks to enforce this. The following is all legal:

```
1  enum NAMES {adam, david, joe, mike} names;
2  names = mike;
3  names = (enum NAMES) (-100);
4  names = 1020;
```

So while an enum should reduce complexity by reducing the set of valid values, the lack of enforcement means that the complexity is at least equal to an integer.

2.4 Pointers

Pointers are are an example of a reference type in type theory. Pointers in C provide allocation, dereferencing and assignment[14, p. 153]. The complex-

ity of a pointer is directly related to the type it is pointing to. Therefore the complexity of a pointer is twice the complexity of the type it points to.

2.5 Typedef

A typedef is used to assign an alternate name to an existing type. This is a process that increases abstraction but which normally makes code more readable and less complex. Therefore, a typedef reduces the complexity by one order of magnitude, or $\frac{1}{2} \cdot |i|$ or 1 whichever is greater.

2.6 Composition Rules

It should be noted that the notation $|i|$ represents the complexity of type i

Rule	Notation	Complexity
Struct	$\times$	$\sum_{i=0}^{n} \|i\|$
Union	$+$	$log_2(\sum_{i=0}^{n} size(i))$
Function	$\rightarrow$	$log_2(size(r) + \prod_{p=0}^{n} size(p))$
Array		$(\|i\| + 1) + log_2(elems)$

2.6.1 Struct

In type theory, a struct in C is best represented as a record. A record is an n-ary tuple where each field is annotated with a unique label. The type constructor of a record is the Cartesian product of the fields [14, p 129].

The complexity of a struct is defined by the logarithm of total numbers of values that the struct can represent. That is:

$log_2(\prod_{i=0}^{n} size(i)) \ \ \forall\, i \in S$ where S is the struct

This formula can be simplified to:

$\sum_{i=0}^{n} |i| \ \ \forall\, i \in S$ where S is the struct and $|i|$ is the complexity of i

Here is an example of a struct and how to calculate its complexity:

```
1 struct course {
2     char* name:
3     unsigned int id;
4     char* teacher;
5     _Bool active;
6
7 }
```

$(2 \cdot 8) + (1.5 \cdot 32) + (2 \cdot 8) + 1 = 81$

There is a special case when a struct has a recursive definition. This is commonly used in linked lists and tree structures, which normally have pointers to the same type. In this case, the complexity of the struct is doubled for each self reference.

Here is an example of a self-referencing struct and how to calculate its complexity:

```
1 struct node {
2     int data;
3     struct node* left;
4     struct node* right;
5 }
```

$2 \cdot 2 \cdot log_2(2^{32}) = 128$

2.6.2 Union

A union in C is similar to a struct, but instead of allocating memory for each type, memory is allocated for the largest field, and only one field can be in use at a time. In type theory, this can be represented as an option type. An option type represents a choice between a finite set of signatures [2, p 22]. Like a struct, each field requires a unique label.

However, since only one value can be active at a time the range of possible values is equal to the size of the largest element. There is complexity in the number of different types that the union can represent. To encapsulate both these concerns, the union is defined as the logarithm of the sum of the elements in the union. That is:

$log_2(\sum_{i=0}^{n} size(i))\ \forall\, i \in U$ where U is the Union

Here is an example of a union and how to calculate its complexity:

```
1 union data {
2       int counter;
3       unsigned int id;
4       _Bool active;
5 }
```

$log_2(2^{32} + (1.5 \cdot 2^{32}) + 2^1) = 33.321928095156$

2.6.3 Functions

In C, a function type is defined by specifying a return type and an argument list. The argument list is similar to a record type, so the complexity of the argument list is treated the same way. Intuitively, the complexity of the argument list should grow more than linearly with the number of arguments because the more arguments a method takes, the more difficult it is to use[13, p 79]. The return type is summed with the argument list because it adds complexity linear to the parameter list. The following rule is used for functions:

$log_2(size(r) + \prod_{p=0}^{n} size(p))$ where r is the return type and p is a parameter

Here is an example of a function and how to calculate its complexity:

```
1 size_t strlcpy(char *dst, const char *src, size_t siz)
```

$log_2(2^{32} + ((2 * 2^8) * 2 * (2 * 2^8) * (2^{32}))) = 51.00000275172$

Currying is not part of the C language, so there is no need to address problems with currying now. Section 4.5 discusses how currying could be included into this metric.

2.6.4 Array

An array has no analogous representation in type theory. An array is more complex than a single element of that type but less complex than a struct with the same number of fields. Intuitively, it also feels less complex than a union.

In C, an array is represented by a pointer to the first element. Therefore and array should be at least as complex as a pointer to the base type. The complexity of an array should grow with the number of elements. To that end, the growth in complexity is logarithmic to the number of elements. Therefore, the equation for calculating the complexity of an array is:

$(|i| + 1) + log_2(elems)$ where $elems$ is the number of elements.

2.7 Macros

Macros do not fit nicely into a type theory category because the macro is changed by the time the type checking occurs. However, macros are a part of the language. Also, interface complexity is measuring the complexity from the perspective of a developer, what he needs to think about and what he types at the keyboard, which makes including macros important.

There are two major types of macros in C, object-like macros and function-like macros. An object like macro is very similar to a typedef, at least from a developers point of view. Therefore, the complexity of an object-like macro is the same as for a typedef; see section 2.5.

Function-like macros are also similar to normal functions, except there is no return value. Therefore the complexity of function-like macros is the product of the types of its arguments.

$$log_2(\prod_{p=0}^{n} size(p))$$

There is an issue where a macro makes changes to the grammar. An example in OpenBSD, is the identifier for a type is a parameter to the function-like macro. The problem is how to define the complexity of the name of an identifier. The solution used is to treat it as an pointer to a char array, which may be giving a complexity greater than it deserves. An example can be seen in section 3.1.1.

2.8 Combining Elements in an API

It is useful to be able to combine complexities for all the external declarations. Two use cases are comparing the complexities of two interfaces or for combining some parts of an interface. One might want to combine parts of an interface to get a complexity of functionality that spans multiple external declarations. For example, creating a list in OpenBSD requires several steps, so it is useful to be able to calculate the complexity of those steps combined.

The collection of external declarations resembles a struct, so the definition is be similar.

$\sum_{i=0}^{n} size(i) \; \forall \; i \; \in \; S$ where S is the set of external declarations

3 Interface Complexity in Practice

The following are examples of the Britt API for data structures on multiple implementations. The implementations come from the OpenBSD project [4] and from an open source library named libds[11]. The OpenBSD implementation is written using macros, while the libds is written using C only.

3.1 Doubly-Linked List

3.1.1 Creation

The creation of a linked list is very different between the two libraries. In the case of OpenBSD, a user supplied struct is needed to hold the data. This struct has the requirement that it contain a field using the macro LIST_ENTRY which contains a self refernece(1). This macro adds the pointers for the next and previous nodes. Once the struct is created, the head node can be created(2) and initialized(3). An example is:

```
1 (1) struct entry {
2         int data;
3         LIST_ENTRY(entry) entries;
4       } *np;
5
6 (2) LIST_HEAD(listhead, entry) head;
7 (3) LIST_INIT(&head);
```

In the libds library the creation of a list requires calling the function:

```
1    list_p create_list();
```

The complexity of creating a doubly-linked list in the two libraries is:

	OpenBSD	libds
Create	$2 \cdot (9 \cdot log(length(head)^*) + (4 \cdot \|data\|)) + (4 \cdot \|data\|)$	47.584963

$^*length(head)$ is the length of the string for the struct identifier

The OpenBSD implementation is more complex that the libds implementation for larger sizes of data. This is due to the extra setup required for using a macro. Also, the OpenBSD implementation requires the data type be defined in order to have pointers to that type while the libds uses a void*.

The LIST_HEAD macro uses the name parameter as the name of the struct. This means that name is not really part of the type. It also means that the name parameter is not required because structs do not require an identifier. Even though the parameter is not necessary and does not make the result more complex, it is something the developer needs to consider so the complexity of the function-like macro is increased. Since it is an identifier, the complexity is the same as a character array of the length of the supplied name.

It is possible to call LIST_HEAD without a name. It looks like this:

LIST_HEAD(, TYPE) head, *headp;

3.1.2 Inserting

Inserting into a list is done differently in both implementations. OpenBSD has three methods for inserting an item; at the head, before an element, and after an element, while the libds only allows inserting at the tail.

	OpenBSD	libds
Insert Tail	$8 \cdot \|data\| + (9 \cdot log_2(length(typename)^*)$	80.584963
Insert Head	$8 \cdot \|data\| + (9 \cdot log_2(length(typename)^*)$	-
Insert Arbitrary	$8 \cdot \|data\| + (9 \cdot log_2(length(typename)^*)$	-

$^*length(typename)$ is the length of the string for the struct identifier

3.1.3 Deleting

Both implementations offer the removal of elements. In the OpenBSD implementation, there is one method for removing an arbitrary element. In the libds implementation, only the head or the tail can be removed.

	OpenBSD	libds
Delete	$4 \cdot \lvert data \rvert + (9 \cdot log_2(length(typename)^*)$	55.584963

$^*length(typename)$ is the length of the string for the struct identifier

3.1.4 Iterating

Both implementations have a method for seeking the next element in the list. OpenBSD requires an element and returns the next element. The libds implementation keeps a separate iterator type, which saves state across calls.

	OpenBSD	libds
Iterating	$4 \cdot \lvert data \rvert + (9 \cdot log_2(length(typename)^*)$	15.000088

$^*length(typename)$ is the length of the string for the struct identifier

4 Hypotheses of Interface Complexity

With the development of interface complexity, there are many new relationships and ideas to explore in the new space. The following are hypotheses for the new metric.

4.1 Ambiguous Primitive Type Size

The sizes of the primitive data types are not strictly defined by the C standard. The C standard only defines a minimum. Furthermore, the metric depends on the size of the data types. This means that different compiler implementations and Operating Systems may have different complexity definitions.

In the case of overall complexity of an API, this should have no effect. If a binary tree is less complex than a hash map in an environment where integers are 32 bits, it should stand that that relationship holds when integers are 64 bits.

The possibility exists that this does not hold in special cases, but that these cases will occur rarely, if at all, in normal source code.

4.2 Inverse Relation with Cyclomatic Complexity

Cyclomatic complexity measures the number of branches within a block of sequential code. If measuring the complexity of an API, one can reduce the amount of external declarators, say all the functions, to a single function. This single function could take all the parameters from all the functions, combining like parameters, and add an extra parameter to signify which case

to take in the the new function. Each case would be a function body from the old functions.

The opposite is also true, one can reduce the amount of certain types of branching by breaking up the function into many functions.

This relationship would mean that increasing cyclomatic complexity would reduce the interface complexity and vice versa. However, there exist ways to increase the cyclomatic complexity without decreasing the interface complexity. Likewise, there exist ways to increase the interface complexity without decreasing the cyclomatic complexity.

If one is attempting to reduce complexity while maintaining functionality, this relationship could indicate that there exists no simpler solution. Additionally, this property could be used by designers of libraries to find an optimal balance between the amount of methods in an interface and the complexity of those methods.

4.3 Measurement of a Language's Expressiveness

Some languages are better at representing certain things better than other languages. Interface complexity could be a way to measure the expressiveness of a language relative to other languages by comparing abstract concepts that can be implemented in both languages, such as data structures.

It could be that a binary tree in language A is less complex than a hash map but that in language B hash map is less complex than binary tree. This could lend validity to the notion that some languages are better for some things and other languages are better for other things.

4.4 Low Numbers of Arguments

Intuitively, as the number of arguments to a function increases, so does the difficulty of using that function. Interface complexity should model this intuition, since the complexity increases as the number of arguments increases. Empirical evidence suggests that the number of functions drops dramatically as the number of arguments goes up [13, p. 88]. This could be a more formal verification of this observation.

4.5 Currying

The way the metric is designed, currying a function will reduce its complexity and uncurrying a function will increase its complexity, as can be seen below:

1. $(T_1 \times T_2) \to T_3$ which give a complexity of $log_2(T_1 \cdot T_2 + T_3)$

2. $T_1 \to T_2 \to T_3$ gives a complexity of $log_2(T_1 + T_2 + T_3)$

This problem was conveniently avoided in C, where there is no notion of currying. However, applying this complexity to a language that does allow currying will require remedying this problem. Intuitively, the two functions should be the same.

5 A Tool for Calculating Interface Complexity

A tool has been written as a proof of concept, the source of which can be found in Appendix B. The tool is written in python and uses the pycparser library[1], an open source C parser. It takes a file or folder as input and will return the interface complexity of every external declaration for each translation unit.

The parsing library takes preprocessed source and returns an ast. The tool walks the ast, recursively calculating the complexity of each external declaration. All the valid types that exist in the translation unit are stored in a dictionary, which is then used for calculating the composition rules.

The tool cannot calculate the complexity of macros. This is because the parser only works on source that has been preprocessed, so the macro expansion has already occurred. This is a difficult problem for any tool based in C that wants to analyse what the developer inputs and not the final C source code.

Another limitation of the library is that it strictly supports c99. This means that compiler specific source code fails to parse. This can be problematic in code bases that rely heavily on compiler specific features. There are some workarounds, such as defining macros that remove the compiler specific parts, since the parser works on source that is already preprocessed. The parser can also be subclassed to include this functionality.

6 Resource Complexities

Resource complexities concern both the expected amount of time and space an algorithm will use. The amount of resources that an interface uses is a vital part of deciding the usefulness of the interface.

The most common way of analysing an algorithm structure is to consider the running time. This is typically represented in big-O notation, which is a bound on the rate of growth as the input size increases. This information is

invaluable and gives the developer a quick way to assess the usefulness of an algorithm.

Computational complexity is the classification of an algorithm according to running time, where constants and lower complexity classes are dropped. This allows for comparison of those classes and works well for sufficiently large sizes of input. Computational complexity is measured in the number of primitive operations, or "steps" executed [3, p. 7].

This information is useful for deciding if an algorithm is suitable for solving the current problem within a reasonable amount of time. Once a developer has an understanding of computational complexity, he can quickly make a reasonable assessment of the running time of an algorithm based on its complexity.

Space complexity is similar to computational complexity, in that it uses a similar classification system. Instead of measuring the number of execution steps, space complexity measures the amount of memory cells required for some arbitrary input, where a memory cell is some base unit of memory [9].

7 Implementation Complexity

There are several different methods of measuring the complexity of a corpus of code. Several metrics relate to implementation and have been considered for inclusion into the Britt API Complexity.

7.1 Cyclomatic Complexity

Cyclomatic complexity measures the executions paths of a program's source code. The purpose of this measurement is to "identify modules that will be difficult to test or maintain." [12]

This metric works by constructing a graph, where nodes represent sequential blocks of code and edges represent branches taken in the program. Branching statements in C include if statements, switch statements, and loops. The graph has one entry node but can have multiple exit nodes, where an exit node corresponds to a return statement. The formula for calculating the cyclomatic complexity is:

$Complexity = E - N + P$
where E is the number of edges
where N is the number of nodes
where P is the number of exit nodes
A simple example, showing both the code and the graph is shown below:

```
1     if(c1())
2       f1();
3     else
4       f2();
5     if(c2())
6       f3();
7     else
8       f4();
9       f5();
```

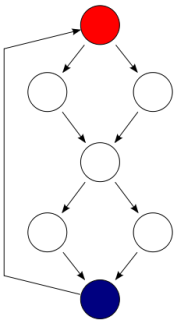

Figure 1: The source code and the cyclomatic graph[7].

Note that f4() and f5() form a single node.
The Complexity is $9 - 7 + 1 = 3$

7.2 Kolmogorov Complexity

Kolmogorov complexity comes from the branch of information theory and is an extrapolation of a long sequence of symbols [15]. For the purposes of this discussion, the long sequence of symbols is a string and the focus will be on the minimum string required for implementation . This can be used to measure how much an implementation deviates from a minimum implementation, in terms of the length of characters used to implement it.

It should be noted that a minimum implementation is in most cases, not desirable. One purpose of an implementation complexity for the Britt API Complexity, is to measure ease of use for a developer. A more concise implementation is easier to understand than a bloated, verbose implementation, but an implementation that uses symbols which represent concepts outside of a normal developers capacity to hold the symbol's definition in his head is equally difficult to use.

7.3 Halstead Complexity Metrics

Halstead complexity metrics are a quantitative measure of complexity directly from the operators and operands in a program. The metric derives several measures, such as difficulty level, effort to implement and number of delivered bugs, amongst others. The fundamentals of the metric are as follows [5]:

n_1 = number of distinct operators

n_2 = number of distinct operands
N_1 = number of operators
N_2 = number of operands

An operand consists of all identifiers, type specifiers and const.

An operator is defined by storage class specifiers, type qualifiers, all other reserved words expect for type specifiers and const, and all operators, such as =, +, <.

From this definition, Halstead metrics derive equations that represent various complexities, such as:

Program Length: $N = N_1 + N_2$
Vocabulary size: $n = n_1 + n_2$
Difficulty Level: $D = (n_1/2) * (N_2/n_2)$

8 Britt API Complexity

Britt API complexity is a meta complexity that is designed to give developers a quick overview of each part of an API. It consists of four different complexities, which give four unique perspectives of an API. The four parts are:

1. Computational Complexity
2. Memory Complexity
3. Interface Complexity
4. Implementation Complexity

This combination offers developers a quickly consumable outline of the API, that tells them both resource use expectation, run time expectation, as well as a hint of how much is actually happening under the hood. The Britt API metric equips the developer so the he can make well informed decisions.

In software development, readable, easily understood code can be just as important as code the runs fast. This tool provides the developer information that can aid in writing code that can be maintained easier. There are circumstances where less efficient code that is more readable or easier to maintain is chosen over overly complex code.

Therefore, developers need a tool that can assist in making decisions that gives insight into both the expected resource use and the readability and usability of the code.

The implementation complexity is represented by cyclomatic complexity. Cyclomatic complexity is a well established, popular metric. It also produces a single value that works on code blocks similar scope to interface complexity, that is, functions. Lastly, the cyclomatic complexity has a potential relation to interface complexity. Implementation and interface complexities also work well in cases where methods have the same function prototype but have different implementations, as is the case in 3.1.2.

8.1 Doubly-Linked List

The following is an example of a doubly-linked list implementation. The implementation comes from the libds library[11]. The cyclomatic complexity has been calculated using the tool CCCC[10]. The source code for list can be found in Appendix D.

	Time	Space	Interface	Cyclomatic
linked_node	-	-	5.000000	-
lnode_p	-	-	6.000000	-
list_iter	-	-	14.000000	-
list_iter_p	-	-	15.000000	-
list_next	$O(c)$	$O(c)$	15.000088	6
list_previous	$O(c)$	$O(c)$	15.000088	6
list_current	$O(c)$	$O(c)$	15.000088	4
list	-	-	46.584963	-
list_p	-	-	47.584963	-
create_list	$O(c)$	$O(n)$	47.584963	1
list_first	$O(c)$	$O(c)$	47.584963	1
list_last	$O(c)$	$O(c)$	47.584963	1
list_pop	$O(c)$	$O(c)$	47.584963	3
list_poll	$O(c)$	$O(c)$	47.584963	3
destroy_list	$O(n)$	$O(c)$	47.584963	1
list_remove	$O(c)$	$O(c)$	55.584963	3
list_iterator	$O(c)$	$O(n)$	55.584963	4
list_add	$O(c)$	$O(n)$	80.584963	1

Empty values represent non-function types.

9 Conclusion

Interface complexity is a new metric for measuring the complexity of types. It takes a corpus of code and generates a positive real number for each external declaration of that code. This number represents a complexity based on the range of values that type can express and how far from the default the type is.

This complexity alone offers developers hints to which libraries should be studied to have potential efficiency gains. It can also be used to keep code complexity to a maintainable level.

Moreover, this complexity harmonizes well with existing complexities, which together can provide a broader picture of a software project. The other types of complexities suggested take into account both resource use and implementation complexity.

9.1 Future Work

There is still much work to be done. Interface complexity has been defined using intuition and type theory. The next step is to define it more formally, perhaps as inference rules.

The interface complexity metric could be tweaked to better reflect real code bases. While the metric attempts to base its rules in concrete theory, the very nature of the metric is subjective. Adjusting the metric based on both large code bases and user testing would be a great step forward. In the case of the Britt API Complexity, more user testing of could lead to choosing other measures to include in the metric.

Another interesting pursuit would be to apply interface complexity to other languages, especially languages that are not imperative. The metric could show that one language is better equipped or more expressive for certain types of applications over other languages. It could also be that certain types of interfaces are more complex than others. For example, in Language A, a linked list might be less complex than a binary tree but in Language B, a linked list might be more complex. There is a real possibility of this because part of the definition of interface complexity is what is default in the language.

References

[1] Eli Bendersky. pycparser. `http://code.google.com/p/pycparser/`, May 2012.

[2] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 431–507. Springer-Verlag, 1991.

[3] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[4] The OpenBSD Foundation. Openbsd. `https://www.openbsd.org`, May 2012.

[5] Verifysoft Technology GmbH. Measurement of halstead metrics with testwell cmt++ and cmtjava (complexity measures tool). `http://www.verifysoft.com/en_halstead_metrics.html`, May 2012.

[6] ISO. Iso c standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[7] JulesH@en.wikipedia.com. Control flow graph of function with two if else statements. `http://en.wikipedia.org/wiki/File:Control_flow_graph_of_function_with_two_if_else_statements.svg`, May 2012. Released into the public domain (by the author).

[8] Donald E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[9] Dexter Kozen. *Theory of Computation*. Springer, New York, 2006.

[10] Tim Littlefair. Cccc. `http://cccc.sourceforge.net`, May 2012.

[11] Zhehao Mao. libds. `https://github.com/zhemao/libds`, May 2012.

[12] Thomas J. McCabe. A complexity measure. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[13] Bertrand Meyer. *Reusable Software - The Base Object-Oriented Component Libraries*. Prentice Hall International (UK) Limited, 1st edition, 1994.

[14] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[15] Ray J. Solomonoff. A formal theory of inductive inference. part i. *Information and Control*, 7(1):1–22, 1964.

A Primitive Data Types and Composition Rules

The composition rules shown below are written to resemble BNF notation but with the interface complexity semantics included. This abuse of notation was designed to make it simpler to see the entire grammar and semantics at one time. This is because the complexity itself does not take up much space, so they all fit together nicely. This makes is easier to read and use as a reference.

To read this document, everything on the left of ::= is a symbol and can be replaced by an expression on the right. An expression consists of one or more symbols. A | represents a choice and is semantically equivalent to the word "or".

Everything in brackets (<or >) is a non-terminal. Everything on the left side of an equals sign (=) and which lack brackets represents a terminal state.

The right hand side of the equals sign are the complexity rules. The complexity rules follow the substitution of an expression, so that they accumulate as the symbols expand.

On the right hand side of the equals sign, the notation $|i|$ is short for the interface complexity of i.

<type> ::=
  <opt-storage-class> <opt-signedness> <type-specifier> |
  <opt-storage-class> <struct-or-union-specifier> |
  <enum-specifier>

<type-specifier> ::=
  <void> = c |
  <char> = $log_2\ (2^8)$ |
  <short> = $log_2\ (2^{16})$ |
  <int> = $log_2\ (2^{32})$ |
  <long> = $log_2\ (2^{32})$ |
  <long long> $log_2\ (2^{64})$ |
  float = $log_2\ (2^{36})$ |
  double = $log_2\ (2^{72})$ |
  long double = $log_2\ (2^{108})$ |
  _Bool = $log_2\ (2^1)$ |
  float _Complex = $log_2\ (2^{36})\ +\ 1$ |
  double _Complex = $log_2\ (2^{72})\ +\ 1$ |
  long double _Complex = $log_2\ (2^{108})\ +\ 1$

<short> ::= short | signed short | short int | signed short int

<int> ::= int | signed | signed int

<long> ::= long | signed long | long int | signed long int

<long-long> ::= long long | signed long long | long long int | signed long long

<opt-signedness> ::=
  signed = 1 * | //this hangs because it is part of an incomplete expression
  unisgned = 1.5 * //this hangs because it is part of an incomplete expression

<struct-or-union-specifier> ::=
  struct = $\sum_{i=0}^{n}\ |i|$ |
  union = $log_2\ (\sum_{i=0}^{n}\ size(i))$ |
  struct : <constant-expression> = $\sum_{i=0}^{n}\ |i|\ *\ 2$ |

union : <constant-expression> = $log_2\ (\sum_{i=0}^{n}\ size(i))\ *\ 2$

<enum-specifier> ::=
  <enum> = $log_2\ (2^{32})$

<type-qualifiers> ::=
  const <type> = 1 + |type| |
  restrict <type> = 1 + |type| |
  volatile <type> = 1 + |type|

<opt-storage-classes> ::=
  auto <type> = |type| |
  register <type> = 2 * |type| |
  static <type> = 2 * |type| |
  extern <type> = 2 * |type| |

<pointer-declarator> ::=
  * <type> = 2 * |type|

<array-declarator> ::=
  <type> <id>[<num>] = 1 + |type| * $\log(num)$

<function-declarator> ::=
  <return-type> <id>(<parameter-type-list>) = $log_2\ (size(r)\ +\ \prod_{p=0}^{n}\ size(p)\ )$

<typedef> <type> ::= .5 * |type| or 1 if result is < 1

<preprocessing-directives> ::=
  <object-like-macro>
  <function-like-macro>

<object-like-macro> ::=
  #define <id> <preprocessing-directive> = |preprocessing-directive| |
  #define <type> = .5 * |type| or 1 if result is < 1

<function-like-macro> ::=
  #define <id>(<parameter-list>) <function-body> = $\prod_{p=0}^{n}\ size(p)\ )$

B Interface Complexity Tool

B.1 cparse.py

```
1  import sys
2  import os
3
4  from pycparser import parse_file, c_ast, c_parser
5  from subprocess import Popen, PIPE
6  from britt import *
7  from utils import *
8
9  ignore_dirs = ['CVS', '.git', '.hg']
10 parse_files = ['c', 'C', 'h', 'H']
11 preprocess_directives = ['#include', '#define', '#ifdef
       ', '#else', '#endif']
12
13
14 def file_walker(root):
15   results = []
16   for directory, dirs, files in os.walk(root):
17     #remove dirs we don't need to walk
18     for d in ignore_dirs:
19       if d in dirs:
20         dirs.remove(d)
21     for f in files:
22       if f.split('.')[-1] in parse_files:
23         results.append(parse(os.path.join(directory, f)
               ))
24   return results
25
26 def parse(fil):
27   pipe = Popen(['gcc', '-E', '-nostdinc', '-I/home/adam
       /programming/glib/', '-I/home/adam/programming/
       glib/glib/', '-I/home/adamin/programming/interface
       -complexity/fake_libc_include/', fil], stdout=PIPE
       , universal_newlines=True)
28   #pipe = Popen(['gcc', '-E', '-nostdinc', '-I/home/
       adam/programming/openbsd/src/sys/sys', '-I/home/
       adam/programming/openbsd/src/sys', '-I/home/adam/
       programming/openbsd/src/sys/arch/i386', '-I/Users/
```

```
        adam/programming/interface-complexity/
        fake_libc_include/', fil], stdout=PIPE,
        universal_newlines=True)
29    text = pipe.communicate()[0]
30    #text = extractCode(text)
31    parser = c_parser.CParser()
32    ast = parser.parse(text)
33    ed = ExtDecl()
34    ed.visit(ast)
35    return fil, ed
36
37  if __name__ == '__main__':
38    f = sys.argv[1]
39    if os.path.isdir(f):
40      res = file_walker(f)
41    else:
42      res = [parse(f)]
43    for f, r in res:
44      print f
45      print r
```

B.2 britt.py

```
1  from pycparser import c_ast
2  import math
3  from operator import itemgetter
4
5  pow2 = lambda x: 2**x
6  log2 = lambda x: math.log(x, 2)
7
8  _c = 1
9  prim_void = _c
10 prim_char = pow2(8)
11 prim_short = pow2(16)
12 prim_int = pow2(32)
13 prim_long = pow2(32)
14 prim_long_long = pow2(64)
15 prim_float = pow2(36)
16 prim_double = pow2(72)
17 prim_long_double = pow2(108)
```

```
18 prim_Bool = pow2(1)
19
20 """
21 This is the list of all known types and their
       complexity.
22 """
23 types = {
24     'void': prim_void,
25     'char': prim_char,
26     'short': prim_short,
27     'int': prim_int,
28     'long': prim_long,
29     'long long': prim_long_long,
30     'float': prim_float,
31     'double': prim_double,
32     'long double': prim_long_double,
33     '_Bool': prim_Bool,
34     }
35
36
37 class ExtDecl(c_ast.NodeVisitor):
38   """
39   This class represents external declarators. This is
         where all types are
40   defined, with the exception of recursive types, which
         are created as needed
41   in the metric and added to the dict of known types
         when needed.
42
43   It uses the node visitor convention.
44   """
45   def __init__(self):
46     self.complexities = []
47
48   def visit_Decl(self, node):
49     if type(node.type) == c_ast.Struct:
50       self.complexities.append((node.type.name, log2(
             britt_metric(node))))
51
52   def visit_Typedef(self, node):
```

```
53      self.complexities.append((node.name, log2(
            britt_metric(node))))
54
55    def visit_FuncDef(self, node):
56      self.complexities.append((node.decl.name, log2(
            britt_metric(node))))
57
58    def __str__(self):
59      res = ['  %s complexity: %f' % item for item in
            sorted(self.complexities, key=itemgetter(1))]
60      return '\n'.join(res)
61
62
63  def britt_metric(ast, seen=[]):
64    """
65    This is where the complexity of each type is
        calculated.
66    """
67    global types
68    typ = type(ast)
69
70    if (typ == c_ast.FuncDef):
71      return britt_metric(ast.decl)
72    elif (typ == c_ast.Decl):
73      return britt_metric(ast.type)
74    elif (typ == c_ast.FuncDecl):
75      prod = 0
76      if ast.args:
77        prod = 1
78        for arg in ast.args.params or []:
79          prod *= britt_metric(arg)
80      return britt_metric(ast.type) + prod
81    elif (typ == c_ast.TypeDecl):
82      return britt_metric(ast.type)
83    elif (typ == c_ast.IdentifierType):
84      try:
85        return types[ast.names[0]]
86      except KeyError:
87        print 'KEY ERROR - type not in environment'
88        print ast
89        raise SystemExit
```

```
90    elif (typ == c_ast.ArrayDecl):
91      return 2 * britt_metric(ast.type)
92    elif (typ == c_ast.PtrDecl):
93      return 2 * britt_metric(ast.type)
94    elif (typ == c_ast.Typedef):
95      score = britt_metric(ast.type)
96      types[ast.name] = score
97      return score
98    elif (typ == c_ast.Typename):
99      return britt_metric(ast.type)
100   elif (typ == c_ast.Struct):
101     if (ast.name in types):
102       return types[ast.name]
103     prod = 1
104     #self referencing struct
105     if ast.name in seen:
106       return 2
107     #empty list is for case when there is no struct
          body
108     for decl in ast.decls or []:
109       prod *= britt_metric(decl, seen.append(ast.name))
110     types[ast.name] = prod
111     return prod
112   elif (typ == c_ast.Enum):
113     return len(ast.values.enumerators)
114   else:
115     print 'ast not yet defined'
116     print ast
117     raise SystemExit
```

B.3 utils.py

```
1  import re
2
3  regex = re.compile(r'# \d+ "[-\w/\.]+" (?P<flag>\d+)')
4
5  def inString(l, s):
6    for i in l:
7      if i in s:
8        return True
```

```
9     return False
10
11  def extractCode(text):
12    """
13    This is a method that strips out all of the include
          files after a file has been preprocessed
14    """
15    extracted = []
16    include_depth = 0
17    for line in text.split('\n'):
18      reg = regex.match(line)
19      if reg:
20        flag = reg.group('flag')
21        if (flag == '1'):
22          include_depth += 1
23        elif (flag == '2'):
24          include_depth -= 1
25      if (include_depth == 0):
26        extracted.append(line)
27    return '\n'.join(extracted)
```

C C Sizes

```c
#include <stdio.h>
#include <limits.h>


int main()
{
  short a = 1;
  int b = 1;
  long c = 1;
  long long d = 1;
  float e = 1.0;
  double f = 1.0;
  long double g = 1.0;
  enum NAMES {adam, david, joe, mike} names;
  size_t h = 10;

  names = (enum NAMES) (-100);
  names = 1020;

  printf("size of short\t%zu\n", sizeof(a));
  printf("size of int\t%zu\n", sizeof(b));
  printf("size of long\t%zu\n", sizeof(c));
  printf("size of long long\t%zu\n", sizeof(d));
  printf("size of float\t%zu\n", sizeof(e));
  printf("size of double\t%zu\n", sizeof(f));
  printf("size of long double\t%zu\n", sizeof(g));
  printf("signedness of char\t%d\n", CHAR_MIN);
  printf("enum\t\t%d\n", names);
  printf("size of size_t\t%zu\n", sizeof(h));
  return 0;
}
```

D libds

D.1 list.h

```c
#ifndef __LIBDS_LIST_H__
```

```
2  #define __LIBDS_LIST_H__
3
4  /* A C implementation of a doubly-linked list. Contains
       void pointer values.
5     Can be used as a LIFO stack of FIFO queue. */
6
7  #define FRONT 0
8  #define BACK 1
9
10 struct linked_node{
11         void* data;
12         struct linked_node* next;
13         struct linked_node* prev;
14 };
15
16 typedef struct linked_node* lnode_p;
17
18 struct list{
19         int length;
20         lnode_p first;
21         lnode_p last;
22         void (*destructor)(void*);
23 };
24
25 typedef struct list * list_p;
26
27 struct list_iter{
28         lnode_p current;
29         char started;
30 };
31
32 typedef struct list_iter * list_iter_p;
33
34 /* Create a linked_list object. This pointer is created
       on the heap and must be
35    cleared with a call to destroy_list to avoid memory
         leaks */
36 list_p create_list();
37
38 /* Create a list_iter object for the linked_list list.
      The flag init can be
```

```
39      either FRONT or BACK and indicates whether to start
            the iterator from the first
40      or last item in the list */
41  list_iter_p list_iterator(list_p list, char init);
42
43  /* Add an item with the given value and size to the
        back of the list.
44      The data is copied by value, so the original pointer
            must be freed if it
45      was allocated on the heap. */
46  void list_add(list_p list, void* data, int size);
47
48  /* Gets the data stored in the first item of the list
        or NULL if the list is empty */
49  void* list_first(list_p list);
50  /* Gets the data stored in the last item of the list or
        NULL if the list is empty */
51  void* list_last(list_p list);
52
53  /* Removes the last item in the list (LIFO order) and
        returns the data stored
54      there. The data returned must be freed later in
            order to remain memory safe. */
55  void* list_pop(list_p list);
56  /* Removes the first item in the list (FIFO order) and
        returns the data stored
57      there. The data return must be freed later in order
            to remain memory safe. */
58  void* list_poll(list_p list);
59  /* Convenience function for completely destroying an
        item in the list. If the end
60      flag is FRONT, an item will be polled from the front
            of the list and its data
61      freed. If the end flag is set to BACK, an item will
            be popped off the end of
62      the list and the data freed. */
63  void list_remove(list_p list, char end);
64
65  /* Completely free the data associated with the list.
        */
66  void destroy_list(list_p list);
```

```
67
68 /* Return the data held by the current item pointed to
      by the iterator */
69 void* list_current(list_iter_p list);
70 /* Advances the iterator to the next item in the list
      and returns the data
71    stored there. */
72 void* list_next(list_iter_p list);
73 /* Advances the iterator to the previous item in the
      list and returns the data
74    stored there. */
75 void* list_prev(list_iter_p list);
76
77 #endif
```

D.2 list.c

```
1  #include "list.h"
2  #include <stdlib.h>
3  #include <string.h>
4
5
6  list_p create_list(){
7          list_p list = (list_p) malloc(sizeof(struct
              list));
8          list->length = 0;
9          list->first = NULL;
10         list->last = NULL;
11         list->destructor = free;
12         return list;
13 }
14
15 list_iter_p list_iterator(list_p list, char init){
16         list_iter_p iter = (list_iter_p)malloc(sizeof(
              struct list_iter));
17         if(init==FRONT){
18                 iter->current = list->first;
19         }
20         else if(init==BACK){
21                 iter->current = list->last;
```

```
22          }
23          else return NULL;
24          iter->started = 0;
25          return iter;
26  }
27
28  void list_add(list_p list, void* data, int size){
29          lnode_p node = (lnode_p)malloc(sizeof(struct
               linked_node));
30          node->data = malloc(size);
31          memcpy(node->data, data, size);
32
33          if(list->first==NULL){
34                  node->prev = NULL;
35                  node->next = NULL;
36                  list->first = node;
37                  list->last = node;
38          }
39          else{
40                  list->last->next = node;
41                  node->prev = list->last;
42                  node->next = NULL;
43                  list->last = node;
44          }
45          list->length++;
46  }
47
48  void* list_current(list_iter_p iter){
49          if(iter->started&&iter->current!=NULL)
50                  return iter->current->data;
51          return NULL;
52  }
53
54  void* list_next(list_iter_p iter){
55          if(!iter->started&&iter->current!=NULL){
56                  iter->started=1;
57                  return iter->current->data;
58          }
59          if(iter->current!=NULL){
60                  iter->current = iter->current->next;
61                  return list_current(iter);
```

```
62          }
63          return NULL;
64  }
65
66  void* list_prev(list_iter_p iter){
67          if(!iter->started&&iter->current!=NULL){
68                  iter->started=1;
69                  return iter->current->data;
70          }
71          if(iter->current!=NULL){
72                  iter->current = iter->current->prev;
73                  return list_current(iter);
74          }
75          return NULL;
76  }
77
78  void* list_first(list_p list){
79          return list->first->data;
80  }
81
82  void* list_last(list_p list){
83          return list->last->data;
84  }
85
86  void* list_pop(list_p list){
87          lnode_p last = list->last;
88          if(last == NULL) return NULL;
89          list->last = last->prev;
90          void* data = last->data;
91          last->prev->next = NULL;
92          free(last);
93          return data;
94  }
95
96  void* list_poll(list_p list){
97          lnode_p first = list->first;
98          if(first == NULL) return NULL;
99          list->first = first->next;
100         void* data = first->data;
101         first->next->prev = NULL;
102         free(first);
```

```
103         return data;
104 }
105
106 void list_remove(list_p list, char end){
107         void * data;
108         if(end == FRONT)
109                 data = list_poll(list);
110         else if (end == BACK)
111                 data = list_pop(list);
112         else return;
113         list->destructor(data);
114 }
115
116 void destroy_list(list_p list){
117         lnode_p cur = list->first;
118         lnode_p next;
119         while(cur!=NULL){
120                 next = cur->next;
121                 list->destructor(cur->data);
122                 free(cur);
123                 cur = next;
124         }
125         free(list);
126 }
```