15-611 Lab 6 Report (C0 and Beyond)

Ethan Cheong
Wu Meng Hui

May 7, 2024

1 Introduction

1.1 Overview

For Lab 6, we implement chars and strings from the L5 language, and contracts and generic pointers from the L6 language. Our compiler supports the default runtime with standard C0 libraries; to support this, we add in a new size `BYTE`, and change our internal representation of bools to be 1-byte.

To support the L6 syntax, we include new parsing rules to parse the keywords required for contracts, and also pattern-match on casts to accept these new expressions.

Furthermore, as part of our extension, we implement array-checking contracts, `forall` and `exists`, which can refer to two new expressions, `element` and `index`. We also provide a test suite `tests` which test various aspects of our compiler.

As per the L6 handout, our compiler no longer supports the `--unsafe` flag. Code can be found at SHA:32e151b6e4ec51b64782161ceefc1c5f9935806b.

1.2 Summary of Results

Our L6 compiler passes all the relevant test cases provided in the `../tests/l6-c1` directory, as well as all the tests we provide. We also test our compiler on the L1-L4 test cases, and our new grammar and compiler is able to pass the relevant test cases. Some previous test cases, such as those which involve annotations or contracts, fail due to invalid syntax. Some previous test cases involving NULL and casting now fail because they expect an error, but instead run to completion because use of NULL and typecasting is supported by the L6 syntax.

2 Extensions

2.1 L5 String and Chars

The behaviours of the following language extensions (except for our Contract Extension) are exactly as described in the L6 handout.

2.1.1 Static Semantics

Strings are a type of their own and are not considered to be an array or a pointer. Chars can be compared with other chars for equality and value. Strings cannot be compared with other strings.

$$\frac{}{\Gamma \vdash c : \text{char}} \qquad \frac{}{\Gamma \vdash s : \text{string}}$$

$$\frac{\Gamma \vdash c_1 : \text{char} \quad \Gamma \vdash c_2 : \text{char}}{\Gamma \vdash c_1 == c_2 : \text{bool}} \qquad \frac{\Gamma \vdash c_1 : \text{char} \quad \Gamma \vdash c_2 : \text{char}}{\Gamma \vdash c_1 \neq c_2 : \text{bool}}$$

$$\frac{\Gamma \vdash c_1 : \text{char} \quad \Gamma \vdash c_2 : \text{char}}{\Gamma \vdash c_1 > c_2 : \text{bool}} \qquad \frac{\Gamma \vdash c_1 : \text{char} \quad \Gamma \vdash c_2 : \text{char}}{\Gamma \vdash c_1 \geq c_2 : \text{bool}}$$

$$\frac{\Gamma \vdash c_1 : \text{char} \quad \Gamma \vdash c_2 : \text{char}}{\Gamma \vdash c_1 < c_2 : \text{bool}} \qquad \frac{\Gamma \vdash c_1 : \text{char} \quad \Gamma \vdash c_2 : \text{char}}{\Gamma \vdash c_1 \leq c_2 : \text{bool}}$$

2.2 Generic Pointers

2.2.1 Static Semantics

Casts are a type of expression. Casts can apply to pointers, and involves casting from one type to another after casting to a void type. The first rule shown below describes how any expression that is not of type void * can be cast as a type void *, and an expression of type void * can be cast as another pointer type not of void *.

$$\frac{\Gamma; \Omega; \Delta \nvdash e : \text{void } *}{\Gamma; \Omega; \Delta \vdash \text{cast}(\text{void } *, e) : \text{void } *} \qquad \frac{\Gamma; \Omega; \Delta \vdash e : \text{void } * \quad \Gamma; \Omega; \Delta \vdash \tau * \neq \text{void } *}{\Gamma; \Omega; \Delta \vdash \text{cast}(\tau *, e) : \tau *}$$

2.2.2 Dynamic Semantics

Generic pointers follow the same dynamic semantics presented in Written 4. $\texttt{tagof}(\tau)$ is a function that takes an argument of type τ and returns an 8-byte tag that uniquely represents τ.

$$H; S; \eta \vdash \texttt{tag}(\tau^*, e) \triangleright K \to H; S; \eta \vdash e \triangleright (\texttt{tag}(\tau^*, \_), K).$$

$$H; S; \eta \vdash a \triangleright (\texttt{tag}(\tau^*, \_), K) \to H'; S; \eta \vdash a' \triangleright K \quad (a \neq 0)$$

where $a'' = H(\text{next}), a' = a'' + 8, H' = H[a'' \mapsto \texttt{tagof}(\tau), a' \mapsto a, \text{next} \mapsto a'']$

$$H; S; \eta \vdash a \triangleright (\texttt{tag}(\tau^*, \_), K) \to H'; S; \eta \vdash 0 \triangleright K \quad (a = 0)$$

$$H; S; \eta \vdash \texttt{untag}(\tau^*, e) \triangleright K \to H; S; \eta \vdash e \triangleright (\texttt{untag}(\tau^*, \_), K).$$

$$H; S; \eta \vdash a \triangleright (\texttt{untag}(\tau^*, \_), K) \to H; S; \eta \vdash 0 \triangleright K \quad (a = 0).$$

$$H; S; \eta \vdash a \triangleright (\texttt{untag}(\tau^*, \_), K) \to \text{exception}(\text{mem}) \quad (a \neq 0, H(a-8) \neq \texttt{tagof}(\tau)).$$

$$H; S; \eta \vdash a \triangleright (\texttt{untag}(\tau^*, \_), K) \to H; S; \eta \vdash H(a) \triangleright K \quad (a \neq 0, H(a-8) = \texttt{tagof}(\tau)).$$

2.3 Contracts

2.3.1 Static Semantics

Contracts are not considered statements or expressions, but they do follow some rules. We will define rules for contracts by referencing their position in the code.

We consider a new space, Φ to store context information about the type of annotation being evaluated. This context is used to store information about whether the annotation exists in a requires, ensures, loop_invariant, or assert statement.

Requires and Ensures For requires and ensures, they must exist in the function definition only. Let a represent a contract (or an annotation). Annotations can be considered to always return a boolean value as part of its checking.

The following rule describes that for all the annotations of a function, they have to be either an ensures or a requires annotation. The first part of the rule is the same as the rule described in the static semantics in the L3 handout for function definitions. The second part of the rule involves checking for all annotations of the function, that the annotation is either an `ensures` or `requires` annotation which typechecks correctly as well. The second part of the rule includes res in context Γ and $\{res : t\}$ in Δ', storing the return type of the function.

$$\frac{\begin{array}{l}\Delta; \Omega \vdash [t]f(t_1 x_1, \cdots, t_{n'} x_{n'}) \rightsquigarrow \Delta', f \quad \text{defined}; \Omega \\ \Gamma, res : t; \Omega; \Delta' \cup \text{res}; \Phi \vdash \forall a_i \in [a_1, \cdots, a_n], a_i : (\text{ensures} \quad || \quad \text{requires})\end{array}}{\Gamma; \Omega; \Delta; \Phi \vdash [t]f(a_1, \cdots, a_n | t_1 x_1, \cdots t_{n'} x_{n'}) \rightsquigarrow \Delta', f \quad \text{defined}; \Omega}$$

For the requires and ensures annotation, they can only contain references to the input elements. For the ensures annotation, it can contain references to the result of the function as well, defined by the symbol res.

Requires and ensures properly evaluate if the expressions contained them evaluate to a boolean type given that the expressions are part of a require or ensure annotation. The following rules describe how to typecheck `ensures` or `requires` annotations. An annotation `\text{requires}(e)` is of type `requires` only if the inner expression evaluates to a boolean value, and the same is said for `ensures` annotations.

$$\frac{\Gamma; \Omega; \Delta; \Phi \cup (\text{requires} : true) \vdash e : \text{bool}}{\Gamma; \Omega; \Delta; \Phi \vdash \text{requires}(e) : \text{requires}} \qquad \frac{\Gamma; \Omega; \Delta; \Phi \cup (\text{ensures} : true) \vdash e : \text{bool}}{\Gamma; \Omega; \Delta; \Phi \vdash \text{ensures}(e) : \text{ensures}}$$

Only the ensures annotation can access the `res` type, and the `res` type exists if res has been as part of the function definition.

$$\frac{\Gamma(\text{res}) = \tau \quad \text{res} \in \Delta \quad \Gamma; \Omega; \Delta; \Phi \vdash \text{ensures} : true}{\Gamma; \Omega; \Delta; \Phi \vdash \text{res} : \tau}$$

Loop Invariants Loop invariants are the only annotations that can annotate a loop body, as shown here. In the case of loop invariants, they are checked before the exit condition is checked. Hence, when a loop invariant annotation is used, even though it annotates the loop body, it is also checked before entering the loop.

Only loop invariants are allowed to annotated loop bodies, as described by the following rule. In the context Φ, `in_loop: true` is added to prevent assert annotations from being used on the statement.

$$\frac{\Gamma;\Omega;\Delta;\Phi \vdash \forall a_i \in [a_1,\cdots,a_n] : (\text{loop\_invariant}) \quad \Gamma;\Omega;\Delta;\Phi \cup \{\text{in\_loop} : \mathit{true}\} \vdash \text{while}(e,s) : [\tau] \to \Delta}{\Gamma;\Omega;\Delta;\Phi \vdash \text{while}(e, \text{anno}(a_1,\cdots,a_n|\text{s})) : [\tau] \to \Delta}$$

$$\frac{\Gamma;\Omega;\Delta;\Phi \cup (\text{loop\_invariant} : \mathit{true}) \vdash e : \text{bool}}{\Gamma;\Omega;\Delta;\Phi \vdash \text{loop\_invariant}(e) : \text{loop\_invariant}}$$

Assert Assert statements can annotate all other statements, except for loops. Thus, the inference rule checks all statements and determines if they are of type `assert`, rejecting these annotations if the context indicates that they are in a loop. In the rule, `in_loop: true` is checked to ensure that assertions are not being used to annotate loop statements. However, since child statements of loops can be annotated with assertions, `in_loop: true` is taken out afterwards when evaluating the statement.

$$\frac{\begin{array}{l}\Gamma;\Omega;\Delta;\Phi \vdash \forall a_i \in [a_1,\cdots,a_n] : \text{assert} \\ \Phi(\text{in\_loop}) \neq \text{true} \\ \Gamma;\Omega;\Delta;\Phi \setminus \{\text{in\_loop}\} \vdash s : [\tau] \to \Delta\end{array}}{\Gamma;\Omega;\Delta;\Phi \vdash \text{anno}(a_1,\cdots,a_n|\text{s}) : [\tau] \to \Delta}$$

$$\frac{\Gamma;\Omega;\Delta;\Phi \cup (\text{assert} : \mathit{true}) \vdash e : \text{bool}}{\Gamma;\Omega;\Delta;\Phi \vdash \text{assert}(e) : \text{assert}}$$

Length Length is an expression that exists only in an annotation/contract, taking in an expression of type array and returns an int. It checks whether the context Φ indicates that the expression is part of any annotation.

$$\frac{\begin{array}{l}\Gamma;\Omega;\Delta;\Phi \vdash e : \tau[] \\ \Phi(\text{ensures}) = \mathit{true}|| \\ \Phi(\text{requires}) = \mathit{true}|| \\ \Phi(\text{loop\_invariant}) = \mathit{true}|| \\ \Phi(\text{assert}) = \mathit{true}\end{array}}{\Gamma;\Omega;\Delta;\Phi \vdash \text{length}(e) : \text{int}}$$

2.4 Contract Extension

Elem, Index, Forall, Exists Our extension to contracts involves an additional feature to check across arrays to ensure all or any element meet a condition given the element and/or index. The intended use is as follows:

```
    //@requires \forall {arr} (\index % 2 == 0 ? \element == 1 : true);
```

The contract above requires that for every single element in the array `arr`, the condition within the parentheses must evaluate to true; that is, if an element is located at an even index, it must be a 1.

```
//@ensures \exists {arr} (\index % 2 == 0 ? \element == 0 : false);
```

Similarly, this contract ensures that there exists at least one element in `arr` at an even index that is equal to 0.

2.4.1 Static Semantics

Forall and exists are expressions which can only exist in an annotation/contract. Thus, the rule checks that the first expression evaluates to an array and that the second expression evaluates to a boolean type. Elements are included in the contexts Γ and Δ as they may be used as part of the contract.

$$\frac{\begin{array}{l}\Gamma;\Omega;\Delta;\Phi \vdash e_1 : \tau[] \quad \Gamma, \text{elem} : \tau; \Omega; \Delta \cup \{elem\}; \Phi \vdash e_2 : bool \\ \Phi(\text{ensures}) = true \mid\mid \\ \Phi(\text{requires}) = true \mid\mid \\ \Phi(\text{loop\_invariant}) = true \mid\mid \\ \Phi(\text{assert}) = true\end{array}}{\Gamma;\Omega;\Delta;\Phi \vdash \text{forall}(e_1, e_2) : \text{int}}$$

$$\frac{\begin{array}{l}\Gamma;\Omega;\Delta;\Phi \vdash e_1 : \tau[] \quad \Gamma, \text{elem} : \tau; \Omega; \Delta \cup \{elem\}; \Phi \vdash e_2 : bool \\ \Phi(\text{ensures}) = true \mid\mid \\ \Phi(\text{requires}) = true \mid\mid \\ \Phi(\text{loop\_invariant}) = true \mid\mid \\ \Phi(\text{assert}) = true\end{array}}{\Gamma;\Omega;\Delta;\Phi \vdash \text{exists}(e_1, e_2) : \text{int}}$$

Elem and Index are new expressions, with elem referring to an element within the array. Elements have type τ if it is within context, and only if it exists in an annotation. Similarly, indexes have type `int` only if it exists in an annotation. They also only exist if `elem` has a type, since that implies that indexes are accessed as part of e_2 of a `forall` or `exists` annotation.

$$\frac{\begin{array}{l}\Gamma(\text{elem}) = \tau \quad \text{elem} \in \Delta \\ \Phi(\text{ensures}) = true \mid\mid \\ \Phi(\text{requires}) = true \mid\mid \\ \Phi(\text{loop\_invariant}) = true \mid\mid \\ \Phi(\text{assert}) = true\end{array}}{\Gamma;\Omega;\Delta;\Phi \vdash \text{elem} : \tau} \qquad \frac{\begin{array}{l}\Gamma(\text{elem}) = \tau \quad \text{elem} \in \Delta \\ \Phi(\text{ensures}) = true \mid\mid \\ \Phi(\text{requires}) = true \mid\mid \\ \Phi(\text{loop\_invariant}) = true \mid\mid \\ \Phi(\text{assert}) = true\end{array}}{\Gamma;\Omega;\Delta;\Phi \vdash \text{index} : \text{int}}$$

3 Compilation

3.1 Frontend

3.1.1 Lexing and Parsing

For the front-end, ASCII characters have to be pattern-matched. Thus, there are 3 new types of tokens for L5: character escapes for chars, string escapes for characters in strings, and printable ASCII keys for all valid ASCIIs from space to tilde. There are two separate escape character sets because of the '
0' character, which is a valid char but not valid in strings. Double-quotes and single-quotes now indicate the start of a string and char respectively for L5 lexing.

For L6, there is no new lexing required for casts. However, there are new tokens required for contracts. We show them below. Additionally, to validate these tokens, we track a new variable, `in_anno_block`. This variable allows us to check if the state is currently in an annotation, since `@` characters are treated as whitespace in annotations and rejected otherwise (except in chars or strings). We also use this variable to check whether `assert` is a contract, or whether it is a statement and we emit the corresponding token for parsing afterwards.

There are now new lexical tokens to support, as shown in A.

For parsing, we followed the parsing rules as described in the handout. Additionally, for our extension, we include the following parsing rules:

```
<exp> ::=
    ...
    | \forall {<exp>} (<exp>)
    | \exists {<exp>} (<exp>)
    | \element
    | \index
```

3.1.2 Data Types for AST

To represent the new data types in the AST, we include the following new expressions

```
    let exp =
    ...
    | Anno_result
    | Anno_elem of tau (* Get the type information from arr *)
    | Anno_index
    | Anno_length of mexp
    | Cast of
      { pointer_type : tau
      ; operand : mexp
      ; orig_type : tau
      }
    | Forall of
      { arr : mexp
      ; condition : mexp
      }
    | Exists of
      { arr : mexp
      ; condition : mexp
      }
```

In particular, cast includes not only the type that the operand will be casted to, but also the original type. The original type will be stored on the heap, allowing us to check at runtime whether the casting from an original type of `void *` to the current pointer type is valid.

We also include annotations/contracts as a list of specifications, as shown.

```
...
let anno = spec list
and spec =
| Requires of mexp
| Ensures of mexp
| Loop_invariant of mexp
| Assert_spec of mexp
...


type program_block =
...
| Function_Def_Anno of
    { ret_type : tau
    ; ident : Symbol.t
    ; params : param list
    ; anno : anno
    ; fn_block : mstm
    }
| Function_Def_Anno_Intermediate of
    { ret_type : tau
    ; ident : Symbol.t
    ; params : param list
    ; anno : anno list
    ; fn_block : mstm list
    }
...
```

These lists of annotations will be elaborated into a single annotation, as referenced by the twice-annotated statement described in the Lab 6 specification.

3.2 Backend

3.3 Introduction of Bytes

In order to represent Bools and Chars with a single byte (so that our compiler is compatible with the default runtime), we introduce a `BYTE` size for Temps, which represent a single byte. Bytes are represented internally with `Int32`. All intermediate representations in the `\lib\data` folder are updated accordingly; for example, consts in the IR tree `Tree.const` now have a `Byte of Int32.t` variant which represents a single byte constant.

The internal representation for Bools changes as well. Previously, the result of boolean expression was simply represented as an integer constant, and as such took up 4 bytes; now, since we have a dedicated Bool type, they instead take up a single byte.

Array allocation also changes to accommodate the new single-byte data. In L4,
`alloc_array(array_type, size)` was translated as follows: (we follow the convention from lecture, where `ins(e)` is a list of commands resulting from the translation of `e`, and `res(e)` is an expression containing the result)

```
ins(alloc_array(array_type, size)) =
[ins(size),
n <- res(size),
if n < 0 then mem_exn else arr_alloc,
mem_exn:,
exception(mem),
arr_alloc:,
t3 <-zero-ext- n,
t4 <- t3 * (size_of_tau arr_type / 4),
t5 <- t4 + 2,
t1 <- calloc(t5, 4),
M[(t1)] <- n,
t2 <- t1 + 8]
res(alloc_array(array_type, size)) =
t2
```

Notice that the code above assumes that `size_of_tau arr_type` is divisible by 4; this was indeed the case in L4. However, translation of Bool and Char arrays has to be treated differently:

```
ins(alloc_array(array_type, size)) =
[ins(size),
n <- res(size),
if n < 0 then mem_exn else arr_alloc,
mem_exn:,
exception(mem),
arr_alloc:,
t3 <-zero-ext- n,
t1 <- calloc(t3, 1),
M[(t1)] <- n,
t2 <- t1 + 8,
]
res(alloc_array(array_type, size)) =
t2
```

This new translation rule means that bools and chars are always located at 1-byte offsets on the heap, allowing our compiler's assembly to be compatible with the libraries provided.

However, to allow for reusability of our previous code for function calls, Bools and Chars are located on a 8-byte offset on the stack, similar to how our Ints are also located on an 8-byte offset on the stack, despite being 4-byte sized. A more space-efficient method (which may reduce stack overflows in some extreme cases) would be to count the number of Byte temps, and then have the first n bytes of the stack used to store that. However, this would also result in more reads from unaligned memory, which might be slower. x86 allows for the lower byte of all 16 registers to be referenced, and we make use of this together with the `movb` instruction. This effectively allows us to treat register allocation entirely the same, regardless of if Temps are `DOUBLE`, `QUAD` or `BYTE`.

3.4 Strings

Consider the following simple program (`simple_strings.l5`):

```
int main() {
    string s = "hello";
    string t = "goodbye";
    return 0;
}
```

Our compiler produces the following assembly:

```
.LC1:
.string "goodbye"
.LC0:
.string "hello"

_c0_main:
subq $24, %rsp
movq  $.LC0, 0(%rsp)
movq  $.LC1, 8(%rsp)
movl  $0, %eax
addq $24, %rsp
ret
```

Strings are represented internally by OCaml's `string` type; all intermediate representations contain an analogous variant that is used to track strings. This persists until the `codegen` phase, where strings have special characters replaced with escape sequences and are loaded into a string table for the entire program. Each time a new string is referenced, the mapping from strings to their labels is updated.

3.5 Generic Pointers

The changes to the code for generic pointers occur solely within `trans.ml`, where a rule is added for translating `Cast {pointer_type; operand; orig_type}`. Since the code contains the translation rule (which in turn is very similar to a question in Written 4) we do not include it here.

The `tagof` function maps types to a unique 64-bit integer; it simply makes reference to a global hashmap `tau_to_tag` which maps all referenced types in the code to their corresponding tag. Types are numbered in the order they are seen during translation.

3.6 Contracts

Again, the changes to the code for contracts occur almost entirely within `trans.ml`; however, the translation of code results in some complications which are worth mentioning. Any omitted translation rules here can be found in `trans.ml`. Contracts are translated into assert statements with some modifications.

Firstly, function definitions can now be annotated and will contain `Requires` and `Ensures` specs if so. The requires and ensures specs are partitioned into separate lists and treated separately. Before translation of the function body, the requires specs are all translated and appended to the IR; before the translation of the expression `exp` in `Return exp`, the ensures specs are translated and appended to the IR.

`Anno_result` can only ever appear in an ensures spec. We maintain a ref `current_return_value` that is updated during the translation of return statements. Since this is updated before `Anno_result` is translated, we can always refer to this when doing the translation.

3.7 Contract Extension

Our extension to contracts introduces even more complications. Forall and Exists are translated directly into a "for-loop" which examines each element of the array, checks the condition and updates a boolean accumulator variable. For example, the translation rule for `forall(arr, condition)` is

```
ins(forall(arr,condition))) = [
i <- 0,
a <- res(arr),
acc <- True,
if a == 0 then mem_exn,
else address_calc:,
mem_exn:,
exception(mem),
address_calc:,
n <- M[-8(a)],
goto while_loop_check,
while_loop_body:,
c <- res(condition),
acc <- acc && (c),
i <- i + 1,
while_loop_check: if i < n then while_loop_body else while_loop_end,
while_loop_end:
]
res(forall(arr,condition)) = acc
```

The difficulty comes with translating `Anno_elem`. We would like to access an element of an array, but `Anno_elem` doesn't contain the expression for the array address, or the index we want to access. We deal with this similarly to how we deal with `Anno_result`; we maintain refs `current_array_temp` and `current_array_index_temp` which keep track of the temps storing the last array and index in a `Forall` or `Exists` expression. The translation is then nearly identical to the translation for array accesses. In addition, maintaining `current_array_index_temp` makes translation of `Anno_index` extremely simple and gives us a language feature almost for free.

4 Examples

Our test cases are located in the `tests` directory, and test aspects of compilation not covered by the provided test suite, as well as the semantics, correctness and use cases of our Contract Extension. We list some examples in this section.

4.1 Chars and Strings

4.1.1 Chars

Characters are treated as single-byte words. Instructions involving characters are translated using the byte-size suffix, as shown in B, such as the `movb` instruction, or comparing values using the lower 8-bits of a register, such as `cmp  %r10b, %al`.

4.1.2 Strings

Since strings are represented using the `.string` directive, they are immutable. For instance, in C, strings defined in the code are referenced using the `.string` directive. If they are used in function calls, their object module is passed into the respective function-call register and treated as a pointer. If the result of the function call is a string, the return register `%rax` is treated as storing a memory location that refers to the new string and used accordingly.

Since strings are not treated as an array of characters, values in strings cannot be accessed like in arrays. The following test case will throw an error in typechecking:

```
int main() {
  return char_ord("abcdefghi"[5]);
}
```

An array of characters can be accessed akin to how arrays are normally accessed. The following test case therefore passes:

```
//test return 1;
int main() {
  char[] cs = alloc_array(char, 5);
  cs[2] = 'o';
  cs[3] = 'p';
  assert(cs[2] == 'o');
  return 1;
}
```

To access individual characters in strings, external function calls have to be made. In D, the function `string_charat` is used to access characters in the string for the binary search function.

4.2 Casts

4.2.1 Previous Test Cases

Some of the previous test cases are no longer valid because `void *` can be used as a type.

For instance, `l4-large/cassini-error_no_void_ptrs.l4` now passes. The code is shown below:

```
//test error
int main() {
    int *x = alloc(int);
    void *v = NULL;
    return 0;
}
```

In L4 and L5 syntax, `void *` is not a valid type. However, because we can do casting, we can store intermediate pointers of type `void *`. Thus, the current L6 compiler will run this code to completion and return 0.

4.2.2 Current Test Cases

In `tests/pointers-null-pointer.l6`, we check for the correctness of types by using `NULL`. Since `NULL` has a type of any∗, equality can be compared for `NULL` pointers. The code is shown below:

```
//test return 0
// "Casting should not affect the null pointer"
int main() {
  int* p = NULL;
  void* q = (void*) p;
  char* r = (char*) q;
  assert(r == NULL);
  return 0;
}
```

We verify the correctness of the tags at runtime when different pointer types are cast from one type to another. In `tests/pointers-back-and-forth-fail-1.l6`, since `y` is assigned `x` and `x` is originally of type bool∗, `y` cannot be compared with `z`, which has type int∗.

```
//test memerror

int main() {
  bool* x = alloc(bool);
  *x = true;
  void* y = (void*) x;
  int* z = (int*) y;
  return *z;
}
```

An interesting case of casting occurs in `tests/deref-magic.l6`. Initially `x` is of type void∗. We can cast `x` to be of type int∗ and de-reference it to store an integer in `x`. Subsequently, we can de-reference `x` again while casting it as an integer pointer, allowing us to access the value stored in `x` despite it being originally of void∗.

```
//test return 5

int main() {
    void * x = (void *) (alloc(int));
   *(int *) x = 5;
    return *(int *)x;
}
```

4.3 Contracts

4.3.1 Previous Test Cases

There are many test cases previous which included annotations. Since the L4 and L5 syntax treat these annotations as comments, they do not affect the correctness of the code. In the current L6 compiler, we parse annotations and thus some of the previous test cases fail. For instance, `l4-large/dobby-t45.l4`. A snippet of the code is shown below:

```
...
hset_t hset_new(int capacity)
/*@requires capacity > 0 && equiv != NULL && hash != NULL; @*/
/*@ensures \result != NULL; @*/ ;
...
```

This code fails to parse correct as the annotations exist for a function declaration, which we reject. Not all function declarations will be defined, meaning we may not be able to annotate these functions correctly. Thus, we only allow annotations to be done for function definitions.

Another example is `l4-large/dobby-t43.l4`. A snippet of the code is shown below:

```
...
void merge(int[] A, int lo, int mid, int hi)
//@requires 0 <= lo && lo < mid && mid < hi && hi <= \length(A);
//@requires is_sorted(A, lo, mid) && is_sorted(A, mid, hi);
//@ensures is_sorted(A, lo, hi);
{
...
}
```

The `is_sorted` function is not declared or defined prior to the definition of merge. Since the function is not within scope, the code fails to typecheck.

4.3.2 Current Test Cases

In `tests/contracts-unit-loop.l6`, we verify that expressions in contracts follow the same static semantics as described. Since `x` is uninitialized, it cannot be used in the loop invariant annotation.

```
//test error

int main() {
  int n = 5;
  int x;
  int acc = 0;
  for (int i = 0; i < n; i++)
  //@ loop_invariant x == 0;
  {
    acc++;
  }
```

```
    return acc;
  }
```

4.4 Contract Extension

We can use the `\forall` expression to verify loop invariants, such as in `tests/extend-loop-invar.l6`. In the code, we ensure that there is at least one element that is 0. The check is run prior to the evaluate of the termination condition, as described in the L6 handout.

```
int main() {
  int[] arr= alloc_array(int, 100);

  for (int i=0; i<99; i++)
  //@loop_invariant \exists {arr} (\element == 0);
  {
    arr[i] = 100;
  }

  //@assert (arr[99] == 0);
  return 0;

}
```

`\index` is particularly useful when combined with ternary expressions; this allows us to check only certain indices of an array. For example,
`extend-ensures-requires.l6`:

```
void modify_arr(int [] arr)
// Check that all even-indexed elements are 1 and all odd-indexed elements are 0
//@requires \forall {arr} (\index % 2 == 0 ? \element == 1 : true);
//@requires \forall {arr} (\index % 2 == 1 ? \element == 0 : true);
// Check that after the function finishes there is a non-1 even-indexed element
//@ensures \exists {arr} (\index % 2 == 0 ? \element == 0 : false);
{
  arr[4] = 0;
}

int main() {
  int [] arr = alloc_array(int, 100);
  for (int i = 0; i < 100; i = i + 2) {
    arr[i] = 1;
  }
  modify_arr(arr);
  return 0;
}
```

We can also combine `\index` with `\length` to check elements of an array within a certain range (although the code gets slightly verbose, and we have to remember to pass the same array as an argument to both `\length` and the enclosing expression). The test `extends-check-tail.l6` checks all but the last 2, then the last 2 elements of an array:

```
int main() {
  int[] A = alloc_array(int, 10);
  A[9] = 1;
  A[8] = 0;
  //@assert \forall {A} ((\index < \length(A) - 2) ? (\element == 0) : true);
  //@assert \exists {A} ((\index >= \length(A) - 2) ? (\element == 1): false);
  return 0;
}
```

5 Analysis

5.1 Language

5.1.1 Casting Ambiguity

There is a small ambiguity that is introduced when casting is implemented. Casting appears in the form $(\tau*)$exp, which may result in shift-reduce conflicts for ternary expressions. For instance, $(\tau*)\ e_1\ ?\ e_2\ :\ e_3$ can be interpreted as $((\tau*)\ e_1)\ ?\ e_2\ :\ e_3$ or $(\tau*)\ (e_1\ ?\ e_2\ :\ e_3)$. We chose to implement the casting by interpreting it as the former (casting e_1 instead of casting the result of the expression), as we found it natural to associate the casting with the nearest expression, similar to the resolution of the dangling else problem.

5.1.2 Contract Extension

As we alluded to in the previous section, while the contract extension is flexible, it is slightly verbose. Language learners have to learn new patterns for doing things like checking a condition holds for certain indices (combine a for-all loop with a ternary statement with a true in the else slot). We considered a different implementation using list comprehensions, where we would introduce a new array comprehension syntax of the form

```
[⟨exp⟩ for ( ⟨simpopt⟩ ; ⟨exp⟩ ; ⟨simpopt⟩ )]
```

which is an expression that evaluates to an array of the same type as `exp`. `forall` and `exists` would then take a boolean array as input, which could be pre-allocated or provided through a list comprehension. While this would make the language easier to write, and also adds the convenience of array comprehensions, it would mean an extra boolean array has to be allocated each time the contract extension is used. Users of the language might also not realise they are allocating memory each time the expression is used (unlike the current language, where `alloc_array` is explicitly called) which might result in more harm than good (memory leaks).

5.2 Compiler and Runtime

As mentioned in the backend section, there may be improvements we can make due to the fact that we now have 1-byte chars and temps. Storing all 1-byte elements together is one; because x86 allows for the lower and upper bytes of the bottom word of the general purpose registers to be accessed separately, we might even be able to modify register allocation to increase the number of registers we have for certain programs. For example, we may be able to treat `AL` and `AH` as separate registers.

Since our compiler is compatible with the provided runtime we are satisfied with that.

5.3 Future Improvements

5.3.1 Casting

Currently, casting is only valid for pointers. For future improvements, since the type for all variables can be known at compile time, we can also consider implementing casting for non-pointer variables. This feature involves using the type of the variable at compile time to cast variables into others. We can treat certain types as subsets of others, such as bool $\subset$ char $\subset$ int. Because bools are 1-byte (but only take up values of 0 or 1), chars are 1-byte, and ints are 4 bytes, we can cast bools as ints or chars as ints without risking truncation. To prevent ambiguity, casting may be explicit, and look like (τ) exp. Thereafter, the type of the expression is checked at compile-time, and any invalid casts are rejected at compile time. For casting bools to ints and chars, the rules may look like the following:

$$\frac{\Gamma; \Omega; \Delta \vdash e : \text{bool}}{\Gamma; \Omega; \Delta \vdash \text{cast\_nonpointer}(\text{char}, e) : \text{char}} \qquad \frac{\Gamma; \Omega; \Delta \vdash e : \text{bool}}{\Gamma; \Omega; \Delta \vdash \text{cast\_nonpointer}(\text{int}, e) : \text{int}}$$

Similar rules can be implemented for casting of chars to int.

5.4 Longs

We can seek to implement longs in the C1 language. To implement longs, the instruction selection section of our compiler will have to change. The changes include using quad-word instructions for binary operations, such as `addq rax, rdi` to support additions and subtractions. Additional care has be be taken for the `div` instruction. Current implementations for division involve using the `cltd` instruction, but since longs are 64-bits, the new instruction for clearing the upper bits prior to a division is `cqto`, converting a quad word to an octal word.

6 Appendix

A Frontend Lexing and Parsing

```
let char_escapes = ['t' 'r' 'f' 'a' 'b' 'n' 'v' '\'' '\"' '\\' '0']
let printable_asciis = [' '-'[' ']'-'~']
let string_escapes = ['t' 'r' 'f' 'a' 'b' 'n' 'v' '\'' '\"' '\\']
...
rule initial = parse
...
| '\'' { char_parse lexbuf; !char_buf }
| '"' { Buffer.clear string_buf; string_parse lexbuf;
T.String_const (Buffer.contents string_buf) }
| "assert" { if !in_anno_line || !in_anno_block then T.Anno_assert else T.Assert }
| "requires" { T.Requires }
| "ensures" { T.Ensures }
| "loop_invariant" { T.Loop_invariant }
| "\\length" { T.Anno_length }
| "\\result" { T.Anno_result }
| "\\exists" { T.Exist }
| "\\forall" { T.Forall }
| "\\element" { T.Arrelement }
| "\\index" { T.Arrindex }
| "/*@" { in_anno_block := true; T.Anno_start }
| "@*/" { in_anno_block := false; T.Anno_end }
| "//@" { in_anno_line := true; T.Anno_line }
| "@" { if (not !in_anno_line) && (not !in_anno_block)
    then error lexbuf ~msg:"@ outside contracts are not whitespace";
    initial lexbuf }
...
and char_parse = parse
| printable_asciis as n { char_buf := parse_char n;     char_end lexbuf }
| '\\' (char_escapes as n) { char_buf := parse_escapes n; char_end lexbuf }
| _ { error lexbuf ~msg:"Invalid string for char"; char_end lexbuf }
and char_end = parse
| '\'' { () }
| eof  { error lexbuf ~msg:"Reached EOF before another ' for chars";
       ()
     }
| "\n"  { error lexbuf ~msg:"Reached new line another ' for chars";
        ()
     }
| _  { error lexbuf
       ~msg:(sprintf "Illegal character '%s'" (text lexbuf));
   }
```

```
and string_parse = parse
| '"' { () }
| '\\' (string_escapes as s) { update_string_buffer (escape_to_ascii s);
  string_parse lexbuf }
| printable_asciis as c { update_string_buffer (char_to_ascii c);
  string_parse lexbuf }
| _  { error lexbuf
       ~msg:(sprintf "Illegal character '%s'" (text lexbuf));
   }
```

B Char Example

Source code in L5:

```
//test return 1;

int main() {
  char[] cs = alloc_array(char, 5);
  cs[2] = 'o';
  cs[3] = 'p';
  assert(cs[2] == 'o');
  return 1;
}
```

Output assembly:

```
.file "tests/char_test.l5"
.globl "_c0_main"

_c0_main:
...
movb  $111, %bpl
...
movb  $111, %r10b
cmp  %r10b, %al
...
```

C String Example

Source code in L5:

```
//test return 0
int main() {
  int n = 1000;
  string[] ss = alloc_array(string, n);
  for (int i = 0; i < n; i++) {
    ss[i] = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
  }
  string acc = "";
  for (int i = 0; i<n; i++) {
    acc = string_join(acc, ss[i]);
  }
  assert(string_length(acc) == (50 * n));
  return 0;
}
```

Output assembly:

```
.file "tests/strings-long.l5"
.globl "_c0_main"
.LC0:
 .string "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
.LC1:
 .string ""
...
_c0_main:
...
movq  $.LC0, %rbp
...
call string_join
addq $0, %rsp
movq  %rax, %rbx
...
movq  %rbx, %rax
...
movq  %rax, %rdi
call string_length
...
```

D String Character Access

```
//test return 22

int binary_search(string s, int len, char target) {
    int left = 0;
    int right = len - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (string_charat(s, mid) == target) {
            return mid;
        } else if (string_charat(s, mid) < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}

int main() {
  string s = "#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  return binary_search(s, string_length(s), '9');
}
```