Compile-Time Array Bounds Checks via Dependent Typing

Team Quagmire (James Gallicchio and Ruiran Xun)

14 December 2021

1 Introduction

In 15-122, we used point-to reasoning to carry out proofs of program safety and correctness. For our L6 project, we sought to automate this reasoning specifically for array safety, since this presents a much more approachable problem than the general case (which we briefly discuss in Section 6).

We achieved this goal by extending the L4 grammar with a restricted form of dependent types. Dependent typing allows types to include expressions from the language. In the type systems of C and L1 through L4, types may include other types, such as how the pointer type `_*` is formed by providing the type it points to.

This extended language (hereafter referred to as L6) represents a substantial deviation from these type systems by allowing (a subset of) expressions to appear in a new "dependently-sized" array type `_[_]`. This lets us write array types like `int[5]` for integer arrays of length 5, or `bool[a]` for bool arrays of length `a`.

Dependent types substantially complicate a type system. One problem to consider is how to define equality of types. Is `int[5]` the same type as `int[2+3]`? Both are valid in L6, but to decide that they are equal requires evaluating the expressions. To complicate matters further, we allow for (non-constant) variables to appear. Is `int[a+b]` the same type as `int[b+a]`? Clearly, yes, but doing so requires proving equality between these arbitrary expressions. And type equality is just one consideration when building a dependently typed system.

However, these new types allow the type system to encode way more information than would otherwise be possible. By incorporating them, we have implemented a compiler that is capable of catching array-out-of-bounds bugs at compile time. This provides us with two performance benefits: it eliminates runtime bounds checks and removes the need to store the length of the array in memory.

This paper explores the technical backbone of our project, the specifics of implementation, our methodology for testing our compiler, and some future directions we can take it. The below commit hash should be referenced alongside this writeup.

`7e946008ba4f5546908a61c59b13209432a2cb7c`

2 Index Terms, Constraints, Static Arrays, and Assignments

2.1 Grammar

The first L6-specific construct we introduce is *index terms*, an embedded language that enables dependent typing in our project. Index terms constrain expressions to integers and binary operations upon integers, specifically addition, subtraction, and multiplication.

$$\langle \text{index} \rangle ::= \textbf{num} \mid \textbf{ident} \mid \langle \text{index} \rangle \texttt{ + } \langle \text{index} \rangle \mid \langle \text{index} \rangle \texttt{ - } \langle \text{index} \rangle \mid \langle \text{index} \rangle \texttt{ * } \langle \text{index} \rangle$$

In order to facilitate making assertions about index terms, we also have *constraints*, which constrain expressions to binary comparisons between index terms and boolean operations upon them (namely conjunction, disjunction, and negation).

$$\langle \text{constr} \rangle ::= \langle \text{index} \rangle \, \langle \text{cmp} \rangle \, \langle \text{index} \rangle \mid \langle \text{constr} \rangle \texttt{ || } \langle \text{constr} \rangle \mid \langle \text{constr} \rangle \texttt{ \&\& } \langle \text{constr} \rangle \mid \texttt{ ! } \langle \text{constr} \rangle$$

We also introduce the type of *static arrays*, a dependent type that incorporates an index term denoting the length of the array.

$$\langle \text{type} \rangle ::= \cdots \mid \langle \text{type} \rangle \texttt{ [} \langle \text{index} \rangle \texttt{]}$$

In order to allocate static arrays and access elements thereof, we extend ⟨exp⟩s and ⟨lvalue⟩s as follows.

$$\begin{aligned} \langle \text{exp} \rangle &::= \textbf{alloc\_array'} \texttt{ (} \langle \text{type} \rangle \texttt{ , } \langle \text{index} \rangle \texttt{) } \mid \langle \text{exp} \rangle \texttt{ [[} \langle \text{index} \rangle \texttt{]]} \\ \langle \text{lvalue} \rangle &::= \cdots \mid \langle \text{lvalue} \rangle \texttt{ [[} \langle \text{exp} \rangle \texttt{]]} \end{aligned}$$

Note that the second argument to **alloc_array'** must be an index term to be well-typed; otherwise, we can't statically guarantee type safety of static arrays. For a basic counterexample, if arbitrary expressions were allowed, the following line of code would be viable.

```
int[3] = alloc_array'(int, foo());
```

In order for this to typecheck, we must be able to demonstrate that `foo() = 3`, but this clearly isn't generally provable.

Lastly, we define *static assignments*, i.e. assignments to integer index terms that we want included in the constraints while typechecking. Notably, the ⟨lvalue⟩ in these static assignments must be an integer (since index terms are integers).

$$\langle \text{simp} \rangle ::= \cdots \mid \langle \text{lvalue} \rangle \texttt{ =' } \langle \text{index} \rangle$$

2.2 Statics

The static semantics for L6 utilize two contexts. The first is the usual context of variable types, Γ. The second is a new context, Θ, representing the index term constraints that are in scope.

First we present a new judgment $\Theta; \Gamma \vdash i$ **index** to describe index terms i in a given context. The judgment is defined below.

$$\frac{(x : \mathsf{int}) \in \Gamma}{\Theta; \Gamma \vdash x \textbf{ index}} \textsc{Dinit}$$

$$\frac{\Theta; \Gamma \vdash i_1 \textbf{ index} \qquad \Theta; \Gamma \vdash i_2 \textbf{ index}}{\Theta; \Gamma \vdash i_1 + i_2 \textbf{ index}} \textsc{Dadd}$$

$$\frac{\Theta; \Gamma \vdash i_1 \textbf{ index} \qquad \Theta; \Gamma \vdash i_2 \textbf{ index}}{\Theta; \Gamma \vdash i_1 * i_2 \textbf{ index}} \textsc{Dmul}$$

For constraints, we present a similar judgement $\Theta; \Gamma \vdash c$ **constr** to describe constraints c in a given context. The judgement is defined below.

$$\frac{\Theta; \Gamma \vdash i_1 \textbf{ index} \qquad \Theta; \Gamma \vdash i_2 \textbf{ index} \qquad \text{cmp} \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{==}, \texttt{!=}\}}{\Theta; \Gamma \vdash i_1 \text{ cmp } i_2 \textbf{ constr}} \textsc{Dcmp}$$

$$\frac{\Theta; \Gamma \vdash c_1 \textbf{ constr} \qquad \Theta; \Gamma \vdash c_2 \textbf{ constr}}{\Theta; \Gamma \vdash c_1 \texttt{ || } c_2 \textbf{ constr}} \textsc{Dor}$$

$$\frac{\Theta; \Gamma \vdash c_1 \textbf{ constr} \qquad \Theta; \Gamma \vdash c_2 \textbf{ constr}}{\Theta; \Gamma \vdash c_1 \texttt{ \&\& } c_2 \textbf{ constr}} \textsc{Dand}$$

$$\frac{\Theta; \Gamma \vdash c \textbf{ constr}}{\Theta; \Gamma \vdash \texttt{!}\, c \textbf{ constr}} \textsc{Dnot}$$

We additionally introduce the following judgement, which indicates that constraint c is provable given the constraint context Θ. Note that this judgement precisely corresponds to logical implication, i.e. whenever the conjunction of all constraints in Θ holds true, ϕ is also true.

$$\Theta \longrightarrow c$$

This rule enables us to define the statics for static array allocation and accesses as follows.

$$\frac{\Theta; \Gamma \vdash l \textbf{ index} \qquad \Theta \longrightarrow 0 \texttt{ <= } l}{\Theta; \Gamma \vdash \textbf{alloc\_array'}(\tau, l) : \tau\,\texttt{[}l\texttt{]}} \textsc{Dalloc}$$

$$\frac{\Theta; \Gamma \vdash \texttt{A} : \tau\,\texttt{[}l\texttt{]} \qquad \Theta; \Gamma \vdash i \textbf{ index} \qquad \Theta \longrightarrow 0 \texttt{ <= } i \texttt{ \&\& } i \texttt{ < } l}{\Theta; \Gamma \vdash \texttt{A[[}i\texttt{]]} : \tau} \textsc{Daccess}$$

For all other expressions carried over from L4, the typing rules remain the same, where the constraint context is simply passed forward without modification.

Another auxiliary judgement we need is the following, which asserts that under typing context Γ and constraint context Θ, the statement s is well-formed with return type τ, and executing s yields a (possibly modified) constraint context Θ'.

$$\Theta; \Gamma \vdash s : [\tau] \Longrightarrow \Theta'$$

The first, most basic use of this judgement is the below rule for static assignment. (It is much more extensively used to specify the statics of control flow in the next section.)

$$\frac{\Theta; \Gamma \vdash x \textbf{ index} \qquad \Theta; \Gamma \vdash a \textbf{ index} \qquad \Theta' = [x_{\text{prev}}/x]\Theta \qquad a' = [x_{\text{prev}}/x]a}{\Theta; \Gamma \vdash x \texttt{ =' } a : [\mathsf{int}] \Longrightarrow \Theta', x \texttt{ == } a'} \textsc{Dassign}$$

This rule uses a notion of index term substitution to deal with the variable x being updated; specifically, it substitutes all instances of x in the constraint context Θ with x_{prev} before adding the new constraint $x \texttt{ == } a$ to the context. This ensures that constraints that were only applicable to x before the assignment aren't applicable afterwards. It also substitutes into a to ensure similar guarantees; for a relevant example, consider the assignment `x =' x + 1`.

3 Control Flow

3.1 Grammar

The work we've done to formalize index terms and constraints truly starts to come to fruition when applied to control flow. In order to do so, we defined separate versions of the existing constructs that are compatible with index terms and constraints, as seen below.

⟨control⟩ ::= ⋯
| **if'** `(` ⟨constr⟩ `;` ⟨constr⟩ `)` ⟨stmt⟩ ⟨elseopt⟩
| **while'** `(` ⟨constr⟩ `;` ⟨constr⟩ `)` ⟨stmt⟩
| **for'** `(` ⟨simpopt⟩ `;` ⟨constr⟩ `;` ⟨simpopt⟩ `;` ⟨constr⟩ `)` ⟨stmt⟩
| **assert'** `(` ⟨constr⟩ `)` `;`

Notably, **if'** statements have two parameters. The first is the usual conditional, and the second is an *invariant* that holds true after the if/else has been executed, regardless of which branch was run.

for' and **while'** also have an extra constraint corresponding to a programmer-specified *loop invariant*. This invariant must hold true before the loop is run, and after an arbitrary iteration of the loop; the compiler seeks to prove this during typechecking.

assert' statements behave as one would expect, except the argument must be in the form of a constraint; the compiler will attempt to prove the constraint, and the code will fail to typecheck if the assertion is unprovable.

3.2 Statics

The typing judgements for these new, statically verified control flow constructs largely resemble those of their counterparts, but with some minor modifications described with each following rule.

$$\frac{\Theta \longrightarrow \phi_{\text{cond}} \qquad \Theta, \phi_{\text{cond}}; \Gamma \vdash s : [\tau]}{\Theta; \Gamma \vdash \textbf{assert'}(\phi_{\text{cond}}), s : [\tau]} \textsc{Dassert}$$

This rule states that, if we assert a constraint ϕ is true, then we may assume it is true for the statement that follows.

$$\frac{\Theta, \phi_{\text{cond}}; \Gamma \vdash s_1 : [\tau] \qquad \Theta, (\neg\phi_{\text{cond}}); \Gamma \vdash s_2 : [\tau] \qquad \Theta, \phi_{\text{inv}}; \Gamma \vdash s : [\tau]}{\Theta; \Gamma \vdash \textbf{if'}(\phi_{\text{cond}}, s_1, s_2, \phi_{\text{inv}}), s : [\tau]} \textsc{Difelse}$$

This rule corresponds to if statements with a non-empty $\langle$elseopt$\rangle$. In the true case, we assume the condition ϕ_{cond} and try to prove that s_1 is well-typed. In the false case, we assume the negation of the condition and try to prove that s_2 is well-typed. If both hold, we get to assume the invariant ϕ_{inv} for the statement that follows.

$$\frac{\Theta, \phi_{\text{cond}}; \Gamma \vdash s_1 : [\tau] \qquad \Theta, (\neg\phi_{\text{cond}}), \phi_{\text{inv}}; \Gamma \vdash s : [\tau]}{\Theta; \Gamma \vdash \textbf{if'}(\phi_{\text{cond}}, s_1, \phi_{\text{inv}}), s : [\tau]} \textsc{Dif}$$

This rule corresponds to if statements without an else case. The true case behaves the same way as in Difelse. In the false case, we assume both the negation of the condition ϕ_{cond} and the invariant ϕ_{inv}, proceeding to try and show the following statement s true.

Before formalizing **for'** and **while'** loops, we introduce a final auxiliary judgement,

$$\Theta; \Gamma \vdash s : [\tau] \rightsquigarrow \Theta'.$$

The motivation is that because loops can modify index terms, when we prove preservation (i.e. the loop invariant at the end of an arbitrary loop's execution), we don't want to keep constraints relating to the index terms' states before the loop executes. The following judgement facilitates this; the statement says that for all index terms defined within s, we perform an index term substitution (similar to that done previously in Dassign).

$$\frac{\{x_1, x_2, \dots\} \text{ defined in } s \qquad \Theta' = [x_{1_{\text{prev}}}/x][x_{2_{\text{prev}}}/x]\cdots\Theta}{\Theta; \Gamma \vdash s : [\tau] \rightsquigarrow \Theta'} \textsc{Dupdate}$$

With this judgement in hand, we now the loop constructs. **while'** is given as follows.

$$\frac{\begin{array}{c}\Theta \longrightarrow \phi_{\text{LI}} \qquad \Theta; \Gamma \vdash s_1 : [\tau] \rightsquigarrow \Theta' \\ \Theta', \phi_{\text{LG}} \longrightarrow \phi_{\text{LI}} \qquad \Theta', \phi_{\text{LG}} \vdash s : [\tau] \Longrightarrow \Theta'' \qquad \Theta'' \longrightarrow \phi_{\text{LI}} \qquad \Theta, (\neg\phi_{\text{LG}}), \phi_{\text{LI}}; \Gamma \vdash s : [\tau]\end{array}}{\Theta; \Gamma \vdash \textbf{while'}(\phi_{\text{LG}}, s_1, \phi_{\text{LI}}), s : [\tau]} \textsc{Dwhile}$$

This judgement is quite hefty; the premises make the following assertions.

1. The constraint context before the loop is adequate to prove the loop invariant initially.
2. After adjusting for definitions inside the loop, the constraint context and the loop guard are able to prove the LI at the start of an arbitrary loop iteration.

3. After executing s, the adjusted constraint context is able to prove the LI at the end of an arbitrary loop iteration.

4. After a loop terminates, the loop invariant and the negation of the loop guard are added to the context and used to verify the rest of the code.

(Side note: in terms of a 15-122 proof, this precisely corresponds to INIT, PRES, and EXIT.)

for' is specified as follows; it is fairly similar to the rule for **while'**, except it accounts for the initialization step.

$$\frac{\begin{array}{c}\Theta;\Gamma \vdash s_{\text{init}} : [\tau] \Longrightarrow \Theta' \qquad \Theta' \longrightarrow \phi_{\text{LI}} \qquad \Theta';\Gamma \vdash s_1 : [\tau] \rightsquigarrow \Theta'' \qquad \Theta'', \phi_{\text{LG}} \longrightarrow \phi_{\text{LI}} \\ \Theta'', \phi_{\text{LG}} \vdash s_1, s_{\text{iter}} : [\tau] \Longrightarrow \Theta''' \qquad \Theta''' \longrightarrow \phi_{\text{LI}} \qquad \Theta', (\neg\phi_{\text{LG}}), \phi_{\text{LI}};\Gamma \vdash s : [\tau]\end{array}}{\Theta;\Gamma \vdash \textbf{for'}(s_{\text{init}}, \phi_{\text{LG}}, s_{\text{iter}}, s_1, \phi_{\text{LI}}), s : [\tau]}\ \textsc{Dfor}$$

The premises of this rule make the following assertions.

1. The constraint context before the loop, taking into account the initialization statement, is adequate to prove the loop invariant initially.

2. After adjusting for definitions inside the loop, the constraint context and the loop guard are able to prove the LI at the start of an arbitrary loop iteration.

3. After executing s_1 and s_{iter}, the adjusted constraint context is able to prove the LI at the end of an arbitrary loop iteration.

4. After a loop terminates, the loop invariant and the negation of the loop guard are added to the context and used to verify the rest of the code.

4 Implementation Details

4.1 Z3 Interface

To prove array accesses are within bounds, and that array allocation lengths are positive, we export the constraints to Z3, a general-purpose SMT solver. Z3 has a built-in theory for bit vectors, which accurately reflects the semantics of integer arithmetic in L4/L6 (particularly overflow and signed comparisons).

We implement a safe, clean interface to Z3 in `typecheck/constraint.ml` which in turn uses Z3's OCaml bindings library. For efficiency, rather than export the entire constraint context every time we need a proof, we incrementally update a Z3 solver model as constraints are added/removed from the context. Z3 has support for stack-like context management, which we make efficient use of.

4.2 Typechecker Changes

Implementing the statics as described in this paper amounted to passing around the new constraint context (as implemented by the `Constraint.ctx` type). When typechecking a statement, we accept a constraint context to begin with, add constraints as needed, and pass an updated constraint context back up to the caller.

The statics of constraints around control flow are subtly different from the statics for variable scope and the statics for initialization, making the implementation unfortunately confusing. We note that most other dependently typed languages unify constraint scope with variable scope, and don't allow for uninitialized values, so this is a difficulty unique to the intersection of C with dependent types.

5 Testing Methodology and Results

We evaluated our compiler for two primary goals. The first is to catch subtle bounds errors via the dependently typed arrays. The second is to use these guarantees to improve performance while in safe mode.

5.1 Running Tests

The tests can be executed locally by running the following series of commands from the root directory of our compiler repository. (Note that Z3 needs to be installed using `opam install z3` for our compiler to build.)

```
$ make
$ bin/c0c --exe tests/<test>
$ gcc -m64 -no-pie tests/<test> ../runtime/run411.o -o tests/<test>.exe
$ tests/<test>.exe
```

Any test cases that fail to compile due to unprovable constraints will fail typechecking with an appropriate error message output to console.

(N.B. `make test` should exhibit the behavior of running `../gradecompiler` on our tests, but since the L5 header file provided in the `dist` repository has `string` as a type, the tests may fail without commenting the string library portion out.)

5.2 Static Bounds Checks

Our compiler for L6 appears to successfully track constraints between control flow constructs, and to successfully catch when those constraints are not strong enough to prove safety of array accesses. We provide a series of basic examples below in order to illustrate how **while'**, **for'**, and **if'** work. These tests, along with some more interesting ones that demonstrate our compiler catching common bugs, can be found in our `tests/` directory.

5.2.1 while' Loops (dwhile.l6)

```
//test return 5

int main() {
  int i =' 0;
  // Here we know i == 0, so the loop invariant
  // holds before running the loop
  while' (i < 5; 0 <= i && i <= 5) {
    // Now we know the guard i < 5, and the LI 0 <= i <= 5,
    // but we no longer know that i == 0
    i =' i + 1;
    // Given i < 5 before the reassignment,
    // we know i + 1 <= 5 before reassignment,
    // e.g. i <= 5 after reassignment:
    assert'(0 <= i && i <= 5);
  }
  assert' (i == 5);
  return i;
}
```

5.2.2 for' Loops (dfor.l6)

```
//test return 9

int main() {

  // 10-elem int array
  int[10] arr = alloc_array'(int,10);

  // Essentially the same as while' loops

  for' (int i =' 0; i < 10; i =' i + 1; i >= 0) {

    // Note we needed i >= 0 as a LI,
    // but we get i < 10 from guard
    arr[[i]] = i;

  }

  return arr[[9]];
}
```

5.2.3 if' Statements (dif.l6)

```
//test return 0

int some_value () {
    return 0;
}

int main() {
    int i = some_value ();
    // Say we want to normalize i to either -1, 0, or 1
    if' (i < 0; -1 <= i && i <= 1) {
        i =' -1; // Fulfills the if invariant
    } else {
        if' (i > 0; i == 0 || i == 1) {
            i =' 1; // Fulfills the inner if invariant
        } else {
            // Note in this case, i >= 0 and i <= 0 so i == 0
            assert'(i == 0); // Fulfills the inner if invariant
        }
        // We know i is either 0 or 1, so that fulfills the outer if invariant
        assert'(-1 <= i && i <= 1);
    }
    // Now we want to ensure that i is non-negative;
    // if it's negative, we will set it to 0
    if' (i < 0; i >= 0) {
        i =' 0;
        // Now clearly i >= 0 so the invariant holds
    }

    // Implicitly, the above has an empty else block, and
    // since i >= 0 in the else clause, the invariant
    // holds in the else case also. Therefore we can prove:
    assert'(i >= 0);
    // Now we want i to be zero.
    i =' 0;
    if' (i != 0; true) {
        // This is clearly dead code. We know i == 0,
        // but this block only runs if i != 0. So, this
        // false assertion is perfectly fine and compiles
        assert'(false);
    }
    assert'(i == 0);
    return i;
}
```

5.2.4 Complex Constraint Example (`triangle.l6`)

This test case is intriguing, because it can prove the exact result of calculating the 10^{th} triangle number via what is essentially an inductive proof.

```
//test return 55

int main() {

  // Calc sum from 0 to max
  int sum =' 0;
  int i =' 0;

  while' (i < 10; 2 * sum == i * (i + 1) && sum >= 0 && 0 <= i && i <= 10) {
    i =' i + 1;
    sum =' sum + i;
  }

  assert'(i == 10);                 // True from guard & LI
  assert'(2 * sum == i * (i+1));    // True from LI
  assert'(2 * sum == 110);          // True by simplification
  assert'(sum == 55);               // Note this only follows because we know sum >= 0 :D

  // This sum was calculated at compile time via inductive proof,
  // NOT by running the loop for 10 iterations :)

  return sum;
}
```

5.3 Performance Improvements

For arrays whose safety is proven at compile time, we do not store the array length nor check array index operations for out of bounds errors. To see the difference in performance, we implemented an (admittedly contrived) example, found at `tests/arrays.l6`, which does lots of array accesses in a loop, and which constructs lots of small arrays in a loop.

The difference in running time and memory use is evident. Running the program 11 times on a machine with Intel i7 and Linux yields the following results.

	L4 –safe	L6 –safe
Avg. Runtime (sec)	2.875	1.437
Max. Memory (kb)	48,388	32,744

Again, this example is contrived to produce such stark differences. But we want to emphasize

the point that no safety guarantees have been lost, yet significant performance gains can be had. Dependent typing allows one to have the best of both safety and performance, at the cost of a more complex typechecker.

6 Future Developments

6.1 Division, Assignment Operators and Postfix Operators

Due to time constraints, we only support a limited set of operators and operations in our index and constraint terms. However, a very quick and easy extension of our work would be to support operators like postfix increment and decrement, or operator assignment like `+=`.

We also wanted to have support for division/modulo, and from our understanding of Z3, we believe it should be as straightforward as the other operations, but we were unsure whether division by zero might introduce subtle unsoundness in our constraint checker. This would require some more careful thinking.

6.2 Data Structure Invariants

We were originally hoping to support safe arrays as fields of structs (whose length is another field of the struct), such as

```
struct bitvec {
    int length;
    bool[length] arr;
}
```

However, we would need to somehow formalize data structure invariants. One major issue with this, and the reason we did not support it here, is that it does not play well with mutation. If one wanted to change the length field of a struct, one would also need to simultaneously update the array field, with some way to ensure the types are correct in the update.

From the perspective of dependent type theory, structs are fundamentally different from functions with dependent types because structs are essentially an existentially quantified type, rather than a universally quantified type like with function signatures. More work would be needed to find a clean way to unify dependent typing with structs.

6.3 Function Contracts

One easier extension to L6 as presented would be to add syntax for contracts at the entry and exit of functions (i.e. preconditions and postconditions). For example, if one wrote a function that takes in an index, one would like to statically require that it be within some range, rather than doing a runtime bounds check within the function. These contracts, we

suspect, would not be hard to implement given what we already have implemented. However, we did not have time to do so for L6.

6.4 Arbitrary Contracts

This work is a good proof of concept for more broadly moving C0's runtime-checked contracts to instead be checked at compile-time via SMT solvers. It should be possible to encode most of C0's semantics into Z3 constraints, thereby enabling automated, compile-time verification of essentially any contract one could write in C0. This would be a pretty large undertaking, but would be immensely useful for catching bugs and for teaching students about provable correctness.

7 Appendix

7.1 L6 Grammar Extensions

⟨index⟩	::=	**num** | **ident** | ⟨index⟩ `+` ⟨index⟩ | ⟨index⟩ `-` ⟨index⟩ | ⟨index⟩ `*` ⟨index⟩
⟨type⟩	::=	$\cdots$ | ⟨type⟩ `[` ⟨index⟩ `]`
⟨exp⟩	::=	**alloc_array'** `(` ⟨type⟩ `,` ⟨index⟩ `)` | ⟨exp⟩ `[[` ⟨index⟩ `]]`
⟨lvalue⟩	::=	$\cdots$ | ⟨lvalue⟩ `[[` ⟨exp⟩ `]]`
⟨simp⟩	::=	$\cdots$ | ⟨lvalue⟩ `='` ⟨index⟩
⟨cmp⟩	::=	`<` | `<=` | `>` | `>=` | `==` | `!=`
⟨constr⟩	::=	⟨index⟩ ⟨cmp⟩ ⟨index⟩ | ⟨constr⟩ `||` ⟨constr⟩ | ⟨constr⟩ `&&` ⟨constr⟩ | `!` ⟨constr⟩
⟨control⟩	::=	$\cdots$
	|	**if'** `(` ⟨constr⟩ `;` ⟨constr⟩ `)` ⟨stmt⟩ ⟨elseopt⟩
	|	**while'** `(` ⟨constr⟩ `;` ⟨constr⟩ `)` ⟨stmt⟩
	|	**for'** `(` ⟨simpopt⟩ `;` ⟨constr⟩ `;` ⟨simpopt⟩ `;` ⟨constr⟩ `)` ⟨stmt⟩
	|	**assert'** `(` ⟨constr⟩ `)` `;`

7.2 L6 Static Semantics

7.2.1 Auxiliary Judgements

- $\Theta; \Gamma \vdash i$ **index**

$$\frac{(x : \mathsf{int}) \in \Gamma}{\Theta; \Gamma \vdash x \textbf{ index}} \textsc{Dinit}$$

$$\frac{\Theta; \Gamma \vdash i_1 \textbf{ index} \qquad \Theta; \Gamma \vdash i_2 \textbf{ index}}{\Theta; \Gamma \vdash i_1 + i_2 \textbf{ index}} \textsc{Dadd}$$

$$\frac{\Theta; \Gamma \vdash i_1 \textbf{ index} \qquad \Theta; \Gamma \vdash i_2 \textbf{ index}}{\Theta; \Gamma \vdash i_1 * i_2 \textbf{ index}} \textsc{Dmul}$$

- $\Theta; \Gamma \vdash c$ **constr**

$$\frac{\Theta; \Gamma \vdash i_1 \textbf{ index} \qquad \Theta; \Gamma \vdash i_2 \textbf{ index} \qquad \text{cmp} \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{==}, \texttt{!=}\}}{\Theta; \Gamma \vdash i_1 \text{ cmp } i_2 \textbf{ constr}} \textsc{Dcmp}$$

$$\frac{\Theta; \Gamma \vdash c_1 \textbf{ constr} \qquad \Theta; \Gamma \vdash c_2 \textbf{ constr}}{\Theta; \Gamma \vdash c_1 \texttt{||} c_2 \textbf{ constr}} \textsc{Dor}$$

$$\frac{\Theta;\Gamma \vdash c_1 \textbf{ constr} \qquad \Theta;\Gamma \vdash c_2 \textbf{ constr}}{\Theta;\Gamma \vdash c_1 \texttt{ \&\& } c_2 \textbf{ constr}} \textsc{Dand}$$

$$\frac{\Theta;\Gamma \vdash c \textbf{ constr}}{\Theta;\Gamma \vdash \texttt{ ! } c \textbf{ constr}} \textsc{Dnot}$$

- $\Theta;\Gamma \vdash s : [\tau] \Longrightarrow \Theta'$
- $\Theta \longrightarrow c$
- $\Theta;\Gamma \vdash s : [\tau] \rightsquigarrow \Theta'$

$$\frac{\{x_1, x_2, \dots\} \text{ defined in } s \qquad \Theta' = [x_{1_{\text{prev}}}/x][x_{2_{\text{prev}}}/x]\cdots\Theta}{\Theta;\Gamma \vdash s : [\tau] \rightsquigarrow \Theta'} \textsc{Dupdate}$$

7.2.2 Language Features

$$\frac{\Theta;\Gamma \vdash l \textbf{ index} \qquad \Theta \longrightarrow 0 \texttt{ <= } l}{\Theta;\Gamma \vdash \textbf{alloc\_array'}(\tau, l) : \tau\texttt{[}l\texttt{]}} \textsc{Dalloc}$$

$$\frac{\Theta;\Gamma \vdash \texttt{A} : \tau\texttt{[}l\texttt{]} \qquad \Theta;\Gamma \vdash i \textbf{ index} \qquad \Theta \longrightarrow 0 \texttt{ <= } i \texttt{ \&\& } i \texttt{ < } l}{\Theta;\Gamma \vdash \texttt{A[[}i\texttt{]]} : \tau} \textsc{Daccess}$$

$$\frac{\Theta;\Gamma \vdash x \textbf{ index} \qquad \Theta;\Gamma \vdash a \textbf{ index} \qquad \Theta' = [x_{\text{prev}}/x]\Theta \qquad a' = [x_{\text{prev}}/x]a}{\Theta;\Gamma \vdash x \texttt{ =' } a : [\mathsf{int}] \Longrightarrow \Theta', x \texttt{ == } a'} \textsc{Dassign}$$

$$\frac{\Theta \longrightarrow \phi_{\text{cond}} \qquad \Theta, \phi_{\text{cond}};\Gamma \vdash s : [\tau]}{\Theta;\Gamma \vdash \textbf{assert'}(\phi_{\text{cond}}), s : [\tau]} \textsc{Dassert}$$

$$\frac{\Theta, \phi_{\text{cond}};\Gamma \vdash s_1 : [\tau] \qquad \Theta, (\neg\phi_{\text{cond}});\Gamma \vdash s_2 : [\tau] \qquad \Theta, \phi_{\text{inv}};\Gamma \vdash s : [\tau]}{\Theta;\Gamma \vdash \textbf{if'}(\phi_{\text{cond}}, s_1, s_2, \phi_{\text{inv}}), s : [\tau]} \textsc{Difelse}$$

$$\frac{\Theta, \phi_{\text{cond}};\Gamma \vdash s_1 : [\tau] \qquad \Theta, (\neg\phi_{\text{cond}}), \phi_{\text{inv}};\Gamma \vdash s : [\tau]}{\Theta;\Gamma \vdash \textbf{if'}(\phi_{\text{cond}}, s_1, \phi_{\text{inv}}), s : [\tau]} \textsc{Dif}$$

$$\frac{\begin{array}{c}\Theta \longrightarrow \phi_{\text{LI}} \qquad \Theta;\Gamma \vdash s_1 : [\tau] \rightsquigarrow \Theta' \\ \Theta', \phi_{\text{LG}} \longrightarrow \phi_{\text{LI}} \qquad \Theta', \phi_{\text{LG}} \vdash s : [\tau] \Longrightarrow \Theta'' \qquad \Theta'' \longrightarrow \phi_{\text{LI}} \qquad \Theta, (\neg\phi_{\text{LG}}), \phi_{\text{LI}};\Gamma \vdash s : [\tau]\end{array}}{\Theta;\Gamma \vdash \textbf{while'}(\phi_{\text{LG}}, s_1, \phi_{\text{LI}}), s : [\tau]} \textsc{Dwhile}$$

$$\frac{\begin{array}{c}\Theta;\Gamma \vdash s_{\text{init}} : [\tau] \Longrightarrow \Theta' \qquad \Theta' \longrightarrow \phi_{\text{LI}} \qquad \Theta';\Gamma \vdash s_1 : [\tau] \rightsquigarrow \Theta'' \qquad \Theta'', \phi_{\text{LG}} \longrightarrow \phi_{\text{LI}} \\ \Theta'', \phi_{\text{LG}} \vdash s_1, s_{\text{iter}} : [\tau] \Longrightarrow \Theta''' \qquad \Theta''' \longrightarrow \phi_{\text{LI}} \qquad \Theta', (\neg\phi_{\text{LG}}), \phi_{\text{LI}};\Gamma \vdash s : [\tau]\end{array}}{\Theta;\Gamma \vdash \textbf{for'}(s_{\text{init}}, \phi_{\text{LG}}, s_{\text{iter}}, s_1, \phi_{\text{LI}}), s : [\tau]} \textsc{Dfor}$$