Lecture Notes on
Semantic Analysis and Specifications

15-411: Compiler Design
André Platzer

Lecture 13
October 5, 2010

1 Introduction

Now we have seen how parsing works in the front-end of a compiler and how instruction selection and register allocation works in the back-end. We have also seen how intermediate representations can be used in the middle-end. One important question is the last phase of the front-end: *semantic analysis* that is used to determine if the input program is actually syntactically well-formed. Another important question arises in the first phase of the middle-end: *translation* of the dynamic aspects of advanced data structures. Even though both questions belong to different phases of the compiler, we answer them together in this lecture. The static and dynamic semantical aspects need to fit together anyhow.

Some smaller subset of what is covered in this lecture can be found in the textbook [**?**, Ch 7.2], which covers data structures.

2 Semantic Analysis and Static Semantics

Essentially, the semantic analysis makes up for syntactical aspects of the language that are important for understanding if the program makes sense, but cannot be represented (easily) in the context-free domain of deterministic parsing. That is, all consistency checks that need information from the context of the current program location. Typical parts of semantic analysis include *name analysis* that is used to identify which particular variable an identifier x refers to, especially where it has been declared. Is it a local

variable? Is it an formal parameter of a function? Is it a global variable (for programming languages that allow this)? Is it an identifier in a struct? Of course, correct name analysis is important to make sure the right memory locations or registers are accessed when looking up or changing the value of x. Name analysis is usually solved by reading off a symbol table with all definitions and their type information from the abstract syntax tree.

Another part of semantic analysis is *type analysis* that is used to look up the types of all identifiers based on the results of name analysis and make sure the types fit. It is also responsible for simple type inference. If we find an expression $e[t.f + x]$ in the source code, then what exactly is the type of the result? And is it a well-typed expression at all? The answer depends on the type of e which had better be an array type (otherwise the array access would be ill-typed). The answer also depends on the type of t which had better be a struct type s and will then be used to lookup the type of $t.f$ according to the type of the field f declared in s. Finally, the answer depends on x. And if the result of the addition $t.f + x$ does not produce an integer, the whole expression still does not type check. It is crucial to find out whether a program with such an expression is well-typed at all. Otherwise, we would compile it to something with a strange and arbitrary effect without knowing that the source program made no sense at all.

All these answers depend on information from the context of the program. One interesting indicator for a language is how many passes of analysis through the abstract-syntax tree are necessary to perform semantical analysis successfully.

A simple typing rule is that for plus expressions:

$$\frac{e_1 : \textbf{int} \quad e_2 : \textbf{int}}{e_1 + e_2 : \textbf{int}}$$

It specifies that if e_1 and e_2 both have type **int** then $e_1 + e_2$ also has type **int**.

In the following, we will give typing rules that define the static semantics of source program expressions.

3 Dynamic Semantics

The static semantics is necessary to make sense of a source code expression. It only specifies it incompletely, though. We will also explain the dynamic semantics of expressions, i.e., what their effect is when evaluated. This information is required for the translation phase in order to make sure that the intermediate language generated for a particular source code snippet

actually complies with the semantics of the programming language, which hopefully fits to the intention that the programmer had in mind when writing the program.

As a side-note, the job of compiler verification is to make sure that the source program will be compiled to something that has exactly the same effect as prescribed by the language semantics, regardless of whether the source program is doing the right thing. The compiler's job is to adhere to this exactly. Contrast this to program verification, where the job is to make sure that the program fits to the intentions that the programmer has in mind, as expressed by some formal specification of what it is meant to achieve, e.g., in the form of a set of pre/postconditions.

For describing the dynamic semantics of C0, we define how we evaluate expressions and statements of the programming language. We need to describe how an expression e will be evaluated to determine the result. For this purpose, we want to define a relation $e \Rightarrow v$ that specifies that e, when evaluated, results in the value v. We want to define the relation $e \Rightarrow v$ by rules specifying the effect of each expression like

$$\frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad n = add(n_1, n_2)}{e_1 + e_2 \Rightarrow n} \; +?$$

This rule is intended to specify that, when e_1 evaluates to value n_1 and e_2 evaluates to n_2, and value n is the sum of values n_1 and n_2, then the expression $e_1 + e_2$ evaluates to n. Unfortunately, it does not quite do the trick yet. To see why, we first consider two other rules. One simple rules that states that constants just evaluate to themselves (similarly for 5,7,...)

$$\frac{}{0 \Rightarrow 0} \; 0$$

And one rule that evaluates a variable identifier x. But what should a variable evaluate to? Well that depends on what its value is. The value of a variable identifier is stored at some address in memory (or a register, which we talk about in a moment). Let's denote the memory address where x is stored by $addr(x)$. This memory address could be for a local variable on the stack, for a spilled function argument on the stack (near beginning of frame), or somewhere in a global data segment for global variables. Either way, it is in memory. Thus when we evaluate a variable identifier, the result is going to be its value from the memory:

$$\frac{x \text{ variable identifier}}{x \Rightarrow M(addr(x))} \; id?$$

Yet we need to the memory contents for this to make sense. So let us reflect this in the notation and change our judgment to $e@M \Rightarrow v$ to say that expression e, when evaluated in memory state M evaluates to value v.

$$\frac{x \text{ variable identifier}}{x@M \Rightarrow M(addr(x))}\ id$$

Yet now what about local variables x that are stored in registers or function arguments that have been passed in registers? The precise option would be to also include the register state R into the judgment $e@M@R \rightarrow v$. Then a variable x that is stored in the register address $addr(x) = \%eax$ would evaluate to $R(\%eax)$ instead of to $M(addr(x))$.

$$\frac{x \text{ variable identifier} \quad x \text{ stored in register}}{x@M@R \Rightarrow R(addr(x))}\ id_1$$

$$\frac{x \text{ variable identifier} \quad x \text{ stored in memory}}{x@M@R \Rightarrow M(addr(x))}\ id_2$$

That is the formally precise way to do it. The only downside is that the notation is a bit clumsy. So instead, what we will do is just simply pretend the registers would be a special part of memory state M stored at the special addresses $M(\%eax), M(\%rbx), M(\%rdi), ...$ This really doesn't change anything except making the notation easier to read. Formally this notation corresponds to considering the cross product $M \times R$ of the real memory state and the register state and just calling the result M again.

Unfortunately, however, the above approach is still only sufficient for describing pure programming languages where no expression can have an effect, except computing its result. The C0 programming language already has no unnecessary side effects during expression evaluation like preincrement/postincrement etc. Yet it still allows function calls in expressions, and function calls can have arbitrary side effects. In order to make sure we do not miss those effects in the semantics, we thus carry an explicit system state M around through the evaluation. We thus look at the judgement $e@M \Rightarrow v@M'$ capturing that an expression e in system state M evaluates to value v and results in the new system state M'. Here, we primarily consider the memory state M, but other state can be tracked too with this principle. Thus the above rule turns into the more precise

$$\frac{e_1@M \Rightarrow n_1@M' \quad e_2@M' \Rightarrow n_2@M'' \quad n = add(n_1, n_2)}{e_1 + e_2@M \Rightarrow n@M''}\ +$$

Unlike the first rule (+?), the new rule now captures the semantics of left-to-right evaluation order. In the first rule (+?), we could still supply the premisses in an arbitrary order and were not restricted to evaluating the subexpressions e_1 and e_2 in any particular order. The new rule (+) explicitly requires left-to-right evaluation, because the state M' resulting from evaluating e_1 is the starting state for evaluating e_2, whose resulting state M'' will be the resulting state of evaluating the whole expression $e_1 + e_2$.

The static and dynamic semantics together give meaning to all elements of the programming language. We treat the static and dynamic semantics for various elements of the C0 programming language at the same time in the following.

4 Small Types

So far, we have only used a programming language with minimal typing. Basically, the only two types so far were **int** and **bool** and are easily distinguished by their respective syntactical occurrence in the language. Only **int** had been allowed as a type for declared variables, and **bool** only occurred in the test expressions for **if**, **while** and **for**.

Real programming languages have more serious types, including C0.

$$\text{Types} \quad \tau \quad ::= \quad \textbf{int} \mid \textbf{bool} \mid \textbf{struct}\, s \mid \tau* \mid \tau[] \mid a$$

where a is a name of a type abbreviation for some type τ and has been introduced in the form

$$\textbf{typedef}\ \tau\ a$$

We mostly ignore the other C0 types **char** and **string** in this course.

For discussing the layout of the various types, we distinguish between *small types* that can fit into a register and *large types* that have to be stored in memory. First we discuss all small types. For the purpose of memory layout and register handling we define the size $|\tau|$ of small types τ as follows:

$$\begin{aligned} |\textbf{int}| &= 4 \\ |\textbf{bool}| &= 4 \\ |\tau *| &= 8 \\ |\tau[]| &= 8 \end{aligned}$$

That is **int** and **bool** are 32-bit and pointers $\tau*$ are represented by 64-bit addresses on 64-bit machines. Arrays themselves are large values and array constants would be large, because we cannot pass a whole array in a

register. But C0 allocates arrays on the heap like pointers and they are only represented by their starting address. Hence, variables of array type have a small type, because we can fit the array address into a register.

Especially we have data of two different sizes. Pointers are allocated from the heap memory by the runtime system using the `alloc`(τ) library function that returns fresh chunks of memory at a location divisible by 8 ready to hold a value of type τ. In C-like programming languages, the null address 0 (denoted by the constant **NULL**) is special in that it will never be returned by `alloc`(τ) (except to indicate that the system ran out of memory altogether). All memory access to the null pointer is thus considered bad memory access.

5 Large Types

Arrays and structures are large types, because they do not (usually) fit into a register. We define their size as follows

$$|s| \quad = \quad pad(|\tau_1|, \ldots, |\tau_n|)$$

where the structure s has been defined to be

$$\begin{array}{l} \textbf{struct } s \{ \\ \qquad \tau_1\ f_1; \\ \qquad \tau_2\ f_2; \\ \qquad \vdots \\ \qquad \tau_n\ f_n; \\ \} \end{array}$$

The function pad adds the sizes of its arguments, adding padding as necessary in between and at the end. That is, elements of type **int** and **bool** are aligned at memory addresses that are divisible by 4. Elements of type $\tau*$ and $\tau[]$ are aligned at memory addresses divisible by 8. A compiler remembers the byte offset of field f_i in the memory layout of structure s in order to find it later. We denote it by $\mathit{off}(s, f_i)$.

Similarly to distinguishing between small and large types, we distinguish between small values (values of a small type) that fit into a register and large values (values of a large type) that have to be stored in memory.

6 Structs

The typing rule for the static semantics of structs is simple and just says that an access to a field f of a struct value e of type s results in a value of type τ, where τ is the declared type of field f in s:

$$\frac{e : s \quad \textbf{struct}\ s\ \{\ldots\ \tau\ f; \ldots\}}{e.f : \tau}$$

To give a dynamic semantics to structs, we define the operational semantics of what happens when we evaluate an expression involving structs. When evaluating $e.f$, we just evaluate e to an address a and then lookup the memory contents at a with the offset $\mathit{off}(s, f)$ belonging to the field f of struct s in memory, i.e., $M(a + \mathit{off}(s, f))$:

$$\frac{e : s \quad \textbf{struct}\ s\ \{\ldots\ \tau\ f; \ldots\} \quad e@M \Rightarrow a@M' \quad \tau \text{ small}}{e.f@M \Rightarrow M'(a + \mathit{off}(s, f))@M'}$$

Unfortunately, this only works well for small types whose values can be returned into registers right away. For large types, this cannot really work well, because the memory $M(a)$ at location a does not even contain all information, and we cannot store the whole object in a single register anyhow. For large types, evaluation produces an address instead, relative to which the content will be addressed further.

$$\frac{e : s \quad \textbf{struct}\ s\ \{\ldots\ \tau\ f; \ldots\} \quad e@M \Rightarrow a@M' \quad \tau \text{ large}}{e.f@M \Rightarrow a + \mathit{off}(s, f)@M'}$$

7 Pointers

To explain the static semantics of pointers and pointer access, there are simple rules:

$$\frac{e : \tau*}{*e : \tau} \qquad \frac{}{\textbf{alloc}(\tau) : \tau*} \qquad \frac{}{\textbf{NULL} : \tau*}$$

If pointer e has the type $\tau*$ of a pointer to an element of type τ, then the pointer dereference $*e$ has the type τ of the element. Allocation of a piece of heap memory for data of type τ gives a pointer of type $\tau*$, i.e., pointing to τ. The last typing rule is a little tricky, because it gives **NULL** all pointer types at once. This is necessary, because the same **NULL** pointer is used

to represent not-allocated regions of arbitrary types. In particular, the type of **NULL** depends on its context, that is, on the expected type that the context wants **NULL** to have in order to make sense of it. In order to avoid ambiguity of the typing, we disallow $*$**NULL**. The expression $*$**NULL** is tricky to type-check because it can lead to ambiguous situations. For instance $(*\textbf{NULL}).f$ could have a lot of types: essentially all types of field f in arbitrary structs declared in the program.

The operational semantics of pointer evaluation can be described using the notation $M(a)$ to denote the content of memory address a. The operational semantics for a pointer access $*e$ evaluates e to an address and then returns the memory contents at that address. Dereferencing the **NULL** pointer must raise the `SIGSEGV` exception. In an implementation this can be accomplished without any checks, because the operating system will prevent read access to address 0 and raise the appropriate exception just by having page 0 unmapped in the virtual-memory page table. When dereferencing pointers that store address a, which are not the null pointer, we obtain their memory contents $M(a)$ (for small types):

$$\frac{e : \tau * \quad e@M \Rightarrow a \quad a \neq 0 \quad \tau \text{ small}}{*e@M \Rightarrow M(a)}\ ? \qquad \frac{e : \tau * \quad e@M \Rightarrow a \quad a \neq 0 \quad \tau \text{ large}}{*e@M \Rightarrow a}\ ?$$

For large types, the memory $M(a)$ at location a does not even contain all information, and we cannot store the whole object in a register anyhow. So instead, $*e$ evaluates to a itself, relative to which the content will be addressed further. When we dereference a pointer that is null, the program terminates with a segmentation fault:

$$\frac{e : \tau * \quad e@M \Rightarrow a \quad a = 0}{*e@M \Rightarrow \text{SIGSEGV}}\ ?$$

For memory allocation, however, we run into some issues when we want to specify what it is doing. After all, memory allocation modifies the memory by finding a free chunk of memory and by clearing the memory contents to 0. Thus memory changes from the old memory M to the new memory M'. We model this by changing our judgement from $e@M \Rightarrow e'$ into $e@M \Rightarrow e'@M'$, in which we also specify the new memory state M'.

$$\frac{M' \text{ like } M \text{ but } M'(a) = \ldots = M'(a + |\tau| - 1) = 0 \text{ for fresh locations}}{\textbf{alloc}(\tau)@M \Rightarrow a@M'}$$

The values stored in freshly allocated location must be all 0. This can be achieved with `calloc()` and means that values of type **int** are simply 0,

value of type **bool** are **false**, values of type $\tau'*$ are null pointers, all fields of structs are recursively set to 0, and values of array type have address 0 which is akin to a null array reference.

This change of judgment to $e@M \Rightarrow e'@M'$ is reflected in the subsequent modifications of the above rules that now carries the memory state through. When doing that, we also notice that expression evaluation during pointer dereference (just as well as all other expression evaluation) can actually modify the memory contents. We fix this deficiency in our previous specification right away:

$$\frac{e : \tau* \quad e@M \Rightarrow a@M' \quad a \neq 0 \quad \tau \text{ small}}{*e@M \Rightarrow M'(a)@M'}$$

$$\frac{e : \tau* \quad e@M \Rightarrow a@M' \quad a \neq 0 \quad \tau \text{ large}}{*e@M \Rightarrow a@M'}$$

$$\frac{e : \tau* \quad e@M \Rightarrow a@M' \quad a = 0}{*e@M \Rightarrow \text{SIGSEGV}@M'} \qquad \frac{}{\textbf{NULL}@M \Rightarrow 0@M}$$

When no functions with side effects (like memory allocation) occur in the expressions, we need not distinguish between M and M' during expression evaluation.

Note, however, that when combining pointers and structs, we cannot necessarily rely on the operating system to trap null pointer dereferencing. For a very large struct s and a pointer $p : s*$, dereferencing a field $p{->}f$ (which desugars into $(*p).f$), the target address may already be beyond the unmapped virtual memory page 0 if f has a large offset.

8 Arrays

Arrays are almost like pointers. Both are allocated. The difference is that C0 pointers disallow pointer arithmetic, whereas arrays can access its contents at arithmetic integer positions randomly. In particular, in arrays, the question rises what to do with access out of bounds, i.e., outside the array size. Does it just access the memory unsafely at wild places, or will it be detected safely and raise a runtime exception? In early labs, we will follow the unsafe C tradition. In later labs, we will switch to safe compilation more like in Java. First, we give simple typing rules explaining the static semantics:

$$\frac{e : \tau[] \quad t : \textbf{int}}{e[t] : \tau} \qquad \frac{e : \textbf{int}}{\textbf{alloc\_array}(\tau, e) : \tau[]}$$

Now we consider the operational semantics. Note that the evaluation order in array access (like everywhere else) is strictly left-to-right. So expression $e[t]$ will be evaluated by evaluating e first and t second, and then accessing the result of e at the result of t:

$$\frac{e : \tau[] \quad e@M \Rightarrow a@M' \quad t@M' \Rightarrow n@M'' \quad a \neq 0 \quad M''(a + n|\tau|) \text{ allocated } \tau \text{ small}}{e[t]@M \Rightarrow M''(a + n|\tau|)@M''}$$

$$\frac{e : \tau[] \quad e@M \Rightarrow a@M' \quad t@M' \Rightarrow n@M'' \quad a \neq 0 \quad M''(a + n|\tau|) \text{ allocated } \tau \text{ large}}{e[t]@M \Rightarrow a + n|\tau|@M''}$$

For safe array access with array bounds check, we add checks to the above rules ensuring that $0 \leq n < N$ where N is the size of the array, which has to be stored at the time of allocation. We have two choices. The liberal but unsafe choice like in C where we leave the evaluation of array access undefined in all other cases that do not match either rule. Or an unambiguously defined semantics choice where we say precisely how array access fails:

$$\frac{e : \tau[] \quad e@M \Rightarrow a@M' \quad t@M' \Rightarrow n@M'' \quad a \neq 0 \quad M''(a + n|\tau|) \text{ not allocated}}{e[t]@M \Rightarrow \mathit{SIGSEGV}@M''}$$

For the case where the address computation of the array itself yields **NULL**, we can either raise a SIGSEGV before evaluating t or after. Both choices are reasonable. The early choice saves operations in case of a SIGSEGV. The late choice, however, reduces the number of times that violations have to be checked, which we thus prefer:

$$\frac{e : \tau[] \quad e@M \Rightarrow a@M' \quad a = 0}{e[t]@M \Rightarrow \mathit{SIGSEGV}@M'} \quad \text{or} \quad \frac{e : \tau[] \quad e@M \Rightarrow a@M' \quad t@M' \Rightarrow n@M'' \quad a = 0}{e[t]@M \Rightarrow \mathit{SIGSEGV}@M''}$$

The difference between those two choices is not gigantic, because it only affects the memory state after an abnormal termination of the program. Getting this semantics right is more important in programming languages like Java where throwing and catching exceptions is used routinely and, in fact, some programs may rely on exceptions being raised all the time in order to function properly.

For the safe array access semantics with array bounds checks, failed checks for array bounds result in SIGABRT, where N is the size of the array:

$$\frac{e : \tau[] \quad e@M \Rightarrow a@M' \quad t@M' \Rightarrow n@M'' \quad a \neq 0 \quad (n < 0 \vee n \geq N)}{e[t]@M \Rightarrow \mathit{SIGABRT}@M''}$$

Allocation of arrays is very similar to allocation of pointers, except that we also check if the size makes sense:

$$\frac{\begin{array}{l} e@M \Rightarrow n@M' \\ n \geq 0 \\ M'' \text{ like } M' \text{ but } M''(a) = \ldots = M''(a + (n-1)|\tau|) = 0 \text{ for fresh locations} \end{array}}{\textbf{alloc\_array}(\tau, e)@M \Rightarrow a@M'}$$

Values in a freshly allocated array are all initialized to 0. This can again be achieved using `calloc`.

Note again, that when combining pointers and arrays, we cannot necessarily rely on the operating system to trap null pointer dereferencing. For a pointer $p : \tau[]*$ to a very large array, accessing $(*p)[70000]$ may already lead to a target address beyond the unmapped virtual memory page 0.

C does not have special support for multidimensional arrays, but just considers $int[][]$ as $(int[])[]$, i.e., an array of arrays of integers. In ragged representation, this two-dimensional array is represented as a one-dimensional array of pointers to arrays. This results in row-major ordering in which the cells in each row are stored one after the other in memory. In ragged representation, there is no guarantee in general that the rows are stored contiguously without gaps (or reorderings). There isn't even a guarantee that all rows are of the same length.

In contrast, statically declared arrays (which are not allowed in C0) are usually stored contiguously in row-major order, because the dimensions are known statically. For instance,

```
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

corresponds to the matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \quad \text{which is stored in memory as} \quad 1\,2\,3\,4\,5\,6$$

Side-note: an odd thing in C is that $x[i]$ and $i[x]$ are both valid array accesses and equivalent, because both are just defined as $*(x + i)$. C even allows $2[x]$ instead of $x[2]$.

9 Assignments to Lvalues

Assignments to primitive **int** variables are simple and ultimately just implemented by a MOV instruction to the respective temp (see lectures 2 and

3). In more complicated languages with structured data, we can assign to other expressions such as $a[10-i]$ or $*p$ or $x.f$ or even $*x.f$ or $(*x).f$ alias $x->f$. Not all expressions qualify as proper expressions to which we can assign to. It makes no sense to try to assign a value to $x+y$ nor to $f(*x-1)$ that may only appear on the right-hand side of an expression (*rvalues*). The expressions that make sense to appear on the left-hand side of an expression as they identify a proper location (say in memory) are called *lvalues*. Lvalues are well-typed expressions of the form

$$x \mid *e \mid e.f \mid e[t] \mid e->f$$

for (well-typed) expressions e, t, primitive variable x and struct field f. The only syntactically valid assignments in C0 are of the form `l=e` or `l+=e` and so on for an lvalue `l` of type τ and an arbitrary (rvalue) expression `e` of type τ. No implicit type cast conversions or coercions happen in C0. While some programming languages allow assignments to large types and give it a memcopy semantics, C0 does not do so, because it is not clear for pointer types if a shallow or deep copy would make more sense. Thus, in C0, only small types can be assigned to directly.

An lvalue represents a destination location for the assignment, which is either a variable x or an address a in memory. Essentially, for determining the target of an lvalue, we use the rules of the structural operational semantics that we have discussed so far, except that we stop at location a before actually doing the memory access $M(a)$. More precisely, we define the relation $v@M \Rightarrow_l d@M'$ to say that lvalue v, when evaluated in memory state M represents location d and this evaluation changed the memory state to M'. It is defined as:

$$\frac{}{x@M \Rightarrow_l x@M} \qquad \frac{e@M \Rightarrow a@M'}{*e@M \Rightarrow_l a@M'} \qquad \frac{e : s \quad e@M \Rightarrow a@M'}{e.f@M \Rightarrow_l a + \mathit{off}(s,f)@M'}$$

$$\frac{e_1 : \tau[] \quad e_1@M \Rightarrow a@M' \quad e_2@M' \Rightarrow n@M''}{e_1[e_2]@M \Rightarrow_l a + n|\tau|@M''}$$

The side conditions and failure modes for the address computation when evaluating lvalue $e_1[e_2]@M \Rightarrow_l \ldots$ of an array access are just like those for the value evaluation $e_1[e_2]@M \Rightarrow \ldots$.

Using this lvalue relation $\Rightarrow_l$, we can define the effect of an assignment $v = e$. The semantics of a statement does not produce a value, it just has an effect on memory. Thus we just write $e@M \Rightarrow @M'$ to describe the

transition. As a shorthand notation, we write $M''\{M''(a) \leftarrow w\}$ for the memory state M''' that is obtained from a memory state M''

$$\frac{v@M \Rightarrow_l x@M \quad e@M \Rightarrow w@M'}{v = e\ @M \Rightarrow @M'\{V(x) \leftarrow w\}}$$

$$\frac{v@M \Rightarrow_l a@M' \quad e@M' \Rightarrow w@M'' \quad M''(a) \text{ allocated}}{v = e\ @M \Rightarrow @M''\{M''(a) \leftarrow w\}}$$

$$\frac{v@M \Rightarrow_l a@M' \quad e@M' \Rightarrow w@M'' \quad a = 0}{v = e\ @M \Rightarrow \text{SIGSEGV}@M''}$$

The effect of an assignment is undefined otherwise. In particular, whether the assignment segfaults during a bad access or not may depend on whether the compiler implements out of bounds checks.

Note especially, that for an assignment $v = e$, the lvalue v will be evaluated to a destination location before the right-hand side expression e will be evaluated. When both v and e have been evaluated, the assignment to v will actually be performed and the destination address a will only be accessed then. In particular:

1. `*e = 1/0` will raise SIGFPE when `e` evaluates without any other exception, because `e` evaluates to an address (without complications) and then, before this memory location is even accessed, the expression 1/0 is computed which throws an exception.

2. `e[-1] = 1/0` should raise a SIGABRT in safe mode, assuming `e` evaluates without any other exception during evaluation of `e`, because the target address computation for the lvalue itself failed.

3. `e->f = 1/0` will raise SIGSEGV when `e` evaluates to **NULL** without any other exception during evaluation of `e`.

In principle, compound assignment operators $\oplus=$ for an operator $\oplus \in \{+, -, *, /, ...\}$ work like assignments, but with the operation $\oplus$. Yet, the meaning of compound assignment operators changes in subtle ways compared to what it meant for just primitive variables. Now compound assignments are no longer just a syntactic expansion, because expressions can now have side effects and it matters how often an expression is evaluated. For a compound assignment `e[t] += e'`, the lvalue of $e[t]$ is only computed once, quite unlike for the assignment `e[t] = e[t] + e'`, where $e[t]$ is evaluated to an address twice. A compound assignment

$$v \oplus= e$$

with an operator $\oplus$ executes as

$$\frac{v@M \Rightarrow_l x@M \quad e@M \Rightarrow w@M'}{v = e\ @M \Rightarrow @M'\{V(x) \leftarrow V(x) \oplus w\}}$$

$$\frac{v@M \Rightarrow_l a@M' \quad e@M' \Rightarrow w@M'' \quad M''(a) \text{ allocated}}{v \oplus= e\ @M \Rightarrow @M''\{M''(a) \leftarrow M''(a) \oplus w\}}$$

10 Function Calls

Suppose we have a function call $f(e_1, \ldots, e_n)$ to a function f that has been defined as $\tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ \{b\}$. We consider a simplified situation here and just assume there is a return variable called $\%eax$ in the function body b.

$$\frac{e_1@M \Rightarrow v_1@M_1, e_2@M_1 \Rightarrow v_2@M_2, \ldots, e_n@M_{n-1} \Rightarrow v_n@M_n \quad b@M'_n \Rightarrow @M' \quad \tau \text{ small}}{f(e_1, \ldots, e_n)@M \Rightarrow M'(\%eax)@M'}$$

where M'_n is like M_n, except that the values v_i of the arguments e_i have been bound to the formal parameters x_i, i.e., $M'_n(x_1) = v_1, \ldots, M'_n(x_n) = v_n$.

And now we remember back that allocation is actually a function call in C0. Consequently, in the intermediate representation of our C0 compiler, side effects due to allocation can only occur at the statement level not nested within expressions. Hence, specifying the semantics for the intermediate representation is actually easier (it doesn't need complicated M'). But, unlike its intermediate representation, C0 itself still needs to respect memory state passing orders carefully.

11 Type Safety

An important property of programming languages is whether they are type-safe. In a type-safe language, the static and dynamic semantics of a programming language should fit together. If we have an expression e in a program that has the type **int**, then we would be rather surprised to find at runtime a result of evaluating e that is a **float**. If this could happen, then it is rather hard to make sure that the program will always execute reasonably even if the compiler accepted it as a well-typed program.

What we expect from the static and dynamic semantics of a type-safe language is that types are preserved in the following sense. If we have a

program that is well-typed (the static semantics says it's okay) and we follow an evaluation step of the dynamic semantics, then the resulting program is still well-typed (*type preservation*). Otherwise what can happen is that we run a well-typed program and suddenly break the well-typing leading to values out of the type ranges. That is, the property that we want (and need to prove for our static and dynamic semantics) is that

$$\text{If } e : \tau \text{ and } e \Rightarrow v \text{ then } v : \tau$$

For C0 (like for all impure programming languages), the statement is a bit more involved, because the dynamic semantics refers to the memory state M. The program reads values from memory and stores values back in memory. If the program would store an **int** into $M(a)$ and then later on expect to read a pointer from $M(a)$, then type-safety is broken. Consequently, type-preservation is a property of the form

$$\text{If } e : \tau \text{ and } e@M \Rightarrow v@M' \text{ and } M \text{ is okay then } v : \tau \text{ and } M' \text{ is okay}$$

for a suitable definition of when a memory state M is "okay", i..e, the types of the values that it stores are compatible with what the program expects.

The other property that one would expect from type-safe languages is that the dynamic semantics always knows what do do (with well-typed programs). We do not want to be stuck in the middle of a run or an interpretation of the program by the dynamic semantics rules not knowing where to go and not having a rule that allows a transition. For instance, if the program contains the well-typed expression $e + f$ and the dynamic semantics does not know how to evaluate the odd expression "test"+0.5, then we better make sure that the evaluation of e can never lead to a string "test" while, at the same time, the evaluation of f leads to the float 0.5.

$$\text{If } e : \tau \text{ and } e \text{ is not a final value then } e \rightarrow e' \text{ for some } e'$$

Again, the real definition of progress is complicated by the fact that we need to consider memory M.

The conjunction of type preservation and progress properties is called type safety [**?**]. Without the progress property, every language could be given a trivially type-preserving dynamic semantics that just stops evaluating whenever it hits an expression that would not preserve types. But that doesn't help write safer programs.