# Lecture Notes on Semantic Analysis

15-411: Compiler Design
André Platzer

Lecture 13b

## 1 Introduction

Last time, we have seen how we can define the static and dynamic semantics of a programming language like C0. But there were a number of loose ends. Basically, we have seen how to give a meaning to expressions by evaluation. But we have not yet given a meaning to statements. We will do this in this lecture.

## 2 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 $\tau$ and an arbitrary (rvalue) expression `e` of

type $\tau$. 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:

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

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

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 $M\{a \mapsto w\}$ for the memory state $M''$ that is obtained from a memory state $M$ by changing the contents of memory location $a$ to the value $w$

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

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

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

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

labs, you will implement a safe compiler for C0 where out of bounds problems have to be checked. The difference between the first two rules is whether the lvalue evaluates to a primitive variable name $x$, so that the effect will be to change the memory contents of the corresponding address $addr(x)$, or whether the lvalue evalutes to an address right away. Notice that we would not necessarily need the first rule had we defined the following rule instead:

$$\frac{}{x@M \Rightarrow_l addr(x)@M}$$

We prefer to split up the rules, however, to make the difference in required actions more apparent. For example, that the memory state will not change when determining lvalues of primitive variables and that we do not need to check whether the memory has been allocated because that is by construction (e.g., local variables are assigned statically to registers or to positions on the stack).

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

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

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

## 3 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 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.

## 4 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 (and other 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 [WF94]. 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.

# Quiz

1. Which of the rules conveys important secret information about how to implement a compiler correctly that are easy to miss?
2. How many ways are there to implement accesses like $(*a)[i]$?
3. Why is 2[i] not allowed in the C0 language when it is allowed in C?
4. Is it important how exactly the compiler implements things like `e[-1] = 1/0` or not?
5. How can you make sure that you always generate the most effective code for the subtleties in the rules? What information do you need for that? Define a dataflow analysis that solves (some) of these issues.
6. In the rules discussed here, what would happen if you would move the primes of memory $M$ around? Which permutations still give a good language semantics? And which permutations are still good for implementation purposes? And which permutations spoil everything?
7. Under which assumptions can you implement a compiler correctly using the rules that do not track $@M$?
8. Can you write a compiler that does not distinguish between Lvalues and Rvalues? Can you write a parser that does not?
9. Should programming languages have multidimensional arrays or should they have an understanding of nested arrays of arrays of arrays instead?
10. Some old C libraries use one-dimensional arrays. These libraries were often translated from Fortran. They probably just didn't know how to write proper C, did they?
11. Suppose you hired a high-school student to translate a Fortran library for numerical computation to C. Suppose it doesn't work or occasionally produces unexpected results. What is your first question?
12. Why is there a difference comparing `e=e+a` and `e+=a`? Should there be a difference? Doesn't this difference only confuse the user?
13. List all advantages and disadvantages that type preservation has when writing a compiler.

14. List all advantages and disadvantages that type preservation has when using a compiler.
15. List all advantages and disadvantages that type progress has when writing a compiler.
16. List all advantages and disadvantages that type progress has when using a compiler.
17. Is your job as a compiler designer easier if you can change the static semantics of the programming language? How?
18. Is your job as a compiler designer easier if you can change the dynamic semantics of the programming language? How?
19. Is your job as a compiler designer easier if you can change the type preservation aspects of the programming language? How?
20. Is your job as a compiler designer easier if you can change the type progress aspects of the programming language? How?
21. In the last questions: what are the downsides for the user?
22. You want to add threads to C0. Which rules do you need to change for that and how? Where are the difficulties?

## References

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.