Lecture Notes on Mutable Store

15-411: Compiler Design

Frank Pfenning and Jan Hoffmann[*]

Lecture 14
October 20, 2020

1 Introduction

In this lecture we extend our language with the ability to allocate data structures on the so-called *heap*. Addresses of heap elements serve as pointers which can be dereferenced to read stored values, or used as destinations for write operations. Similarly, arrays are stored on the heap[1] and via appropriate address calculations.

Adding mutable store requires yet again a significant change in the structure of the rules of the dynamic semantics. By contrast, the static semantics is relatively easy to extend.

2 Pointers

We extend our language of types with $\tau*$, where τ is a (non-void) type.

$$\tau ::= \mathsf{int} \mid \mathsf{bool} \mid \tau*$$

In the language of expressions, we can allocate a cell on the heap that can hold a value of type τ, we have a distinguished null pointer, and we can dereference a pointer to obtain the stored value.

$$e ::= \ldots \mid \mathsf{alloc}(\tau) \mid *e \mid \mathsf{null}$$

They have the following typing rules:

$$\frac{}{\Gamma \vdash \mathsf{alloc}(\tau) : \tau*} \qquad \frac{\Gamma \vdash e : \tau*}{\Gamma \vdash *e : \tau} \qquad \frac{}{\Gamma \vdash \mathsf{null} : \tau*}$$

[*]Typos fixed by Seth Goldstein, 2019
[1]C0 does not have stack-allocated arrays

At first glance they might be harmless, but the third rule should raise a red flag: we previously claimed in our mode analysis of typing, that given Γ and e we can synthesize the type of e (if it exists). However, in the rule for null that's not the case.

3 Detail: Typing $*\mathsf{null}$

We cannot synthesize a *definite* type for null. Unfortunately, we also cannot, in general, know what type to check an expression against. So we'll synthesize an indefinite type, let's call it $\mathit{any}\,*$, the type of a pointer to data of potentially any type.

Now we have to walk through all the constructs in the language to see whether we can resolve $\mathit{any}\,*$, assuming it can only arise for null. Let's consider *pointer equality* first, that is, an expression p == q where p and q are pointers. If p and q both have definite type $\tau*$, we just treat it as well-typed. If one has type $\tau*$ and the other $\tau'*$ for $\tau \neq \tau'$, we reject the comparison as ill-typed. If one is definite $\tau*$ and the other indefinite, we allow the comparison, because the indefinite type has only one value (null) which can be compared to a pointer of any definite type. If both are indefinite, we would be comparing null with null, which is also fine.

One way to capture this is to have a so-called *type subsumption rule* that allows a "silent" transition:

$$\frac{\Gamma \vdash e : \mathit{any}\,*}{\Gamma \vdash e : \tau*}$$

Then three rules suffice for our overloaded equality:[2]

$$\frac{\Gamma \vdash e_1 : \tau* \quad \Gamma \vdash e_2 : \tau*}{\Gamma \vdash e_1 \texttt{==} e_2 : \mathsf{bool}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \texttt{==} e_2 : \mathsf{bool}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \mathsf{bool}}{\Gamma \vdash e_1 \texttt{==} e_2 : \mathsf{bool}}$$

A difficulty arises with the dereferencing operator: $*\mathsf{null}$ would have *any* type, which means it could essentially appear anywhere. Of course, when run, it will always yield an exception, since dereferencing the null pointer is disallowed. We therefore rewrite our earlier rule to disallow dereferencing values of indefinite type.

$$\frac{\Gamma \vdash e : \tau* \quad \Gamma \nvdash e : \mathit{any}\,*}{\Gamma \vdash *e : \tau}$$

In particular $*\mathsf{null}$ is disallowed, and so is $*(b \; ? \; \mathsf{null} \; : \; \mathsf{null})$ and variants thereof, because the conditional still has indefinite type $\mathit{any}*$. Of course, indefinite types are not part of the source language and only used internally during type checking.

[2]Actually, in this language fragment just one would suffice, since elements of all types can be compared for equality.

4 Dynamic Semantics for Pointers

A value of type $\tau*$ is just an address where a value of type τ is stored, or the special address 0. Allocation returns an unused address, and dereferencing the pointer retrieves the stored value. But where is the store? We currently only carry an environment η that maps variables to their values. We now also carry a *heap* H that maps addresses to stored values.

A question that arises is how we should represent addresses, that is, the domain of the heap H. One possibility would be to say that addresses have 64 bits like in our target architecture. The benefit of this approach is that the abstract machine remains very close to the real machine on which compiled programs execute. The downside is that we only have a finite amount of addresses and that we can run out of memory: for example if we allocate space for an integer 2^{64} times then execution will get stuck or have to throw an exception. At first sight, this seems to be a good specification. However, what if the runtime provides a garbage collector that automatically frees memory that is not reachable from the stack anymore? Then our dynamic semantics would potentially require to throw an out of memory exception even though there is still plenty of memory left.

To deal with all possible implementations and behaviors of the operating system, we are treating memory failures in the same way we treat stack overflow. We do not model them in the high-level dynamic semantics but allow that they can happen at runtime. This is why we assume we have an infinite address space and that the heap maps natural numbers to values.

To keep track of the free memory we also store a pointer to the next available address. For simplicity, we assume that this pointer is part of the heap and stored at a special address next

$$H : (\mathbb{N} \cup \{\mathsf{next}\}) \to \mathsf{Val}$$

Evaluation of expressions *may change the heap*, because it may call a function that changes its state. The state of the abstract machine therefore carries heaps, stacks, and continuations.

$$H \; ; S \; ; \eta \vdash e \rhd K$$

Here the semicolon ';' is just a separator between the heap H, the stack S, and the current environment η.

> *Given heap H, stack S, and environment η, we evaluate expression e with continuation K.*

Execution of statements is similarly generalized.

All the prior transition rules leave the heap unchanged. For example

$$H \; ; S \; ; \eta \vdash e_1 \odot e_2 \rhd K \qquad \longrightarrow \qquad H \; ; S \; ; \eta \vdash e_1 \rhd (\_ \odot e_2 \, , K)$$

In the following, we present the new rules for manipulating pointers. The expression null evaluates to the address 0. Allocation returns a fresh address a and maps it to an appropriate default value in the new heap.

$$\begin{array}{lll} H \,;\, S \,;\, \eta \vdash \mathsf{null} \rhd K & \longrightarrow & H \,;\, S \,;\, \eta \vdash 0 \rhd K \\ H \,;\, S \,;\, \eta \vdash \mathsf{alloc}(\tau) \rhd K & \longrightarrow & H[a \mapsto \mathsf{default}(\tau), \mathsf{next} \mapsto a + |\tau|] \,;\, S \,;\, \eta \vdash a \rhd K \\ & & a = H(\mathsf{next}) \end{array}$$

Freshly allocated cells are initialized with a default value for the type τ. In the implementation, this is arranged to always be 0 (in whatever word length required by the size of τ). For booleans this means false, for integers 0 and for pointers null in the source language.

For the implementation of this rule, we need to know the sizes of each type. This is, of course, highly dependent on the processor architecture and conventions. For this course, we compile to x86-64, in which case we have:

$$\begin{array}{lcl} |\mathsf{int}| & = & 4 \\ |\mathsf{bool}| & = & 4 \\ |\tau*| & = & 8 \\ |\tau[\,]| & = & 8 \end{array}$$

Of course, 8 does not correspond to the real sizes of the (unbounded) addresses that we have at the high level. But this is not important as we only need to provide a accurate model of the target and not implement this exact model. In the target code, addresses will have 64 bit.

Dereferencing a pointer just retrieves from the address, assuming it is not 0. If it is 0, we raise the memory exception mem, which for us will be the signal `SIGUSR2` (12) on our architecture. (We use `SIGUSR2` instead of the obvious choice, `SIGSEGV`, for two reasons: it allows us to better distinguish stack overflow, and it allows us to distinguish "accidental" and "on purpose" memory errors.)

$$\begin{array}{llll} H \,;\, S \,;\, \eta \vdash *e \rhd K & \longrightarrow & H \,;\, S \,;\, \eta \vdash e \rhd (*\_ \,,\, K) & \\ H \,;\, S \,;\, \eta \vdash a \rhd (*\_ \,,\, K) & \longrightarrow & H \,;\, S \,;\, \eta \vdash H(a) \rhd K & (a \neq 0) \\ H \,;\, S \,;\, \eta \vdash a \rhd (*\_ \,,\, K) & \longrightarrow & \mathsf{exception}(\mathsf{mem}) & (a = 0) \end{array}$$

In order to implement this correctly at a lower level of abstraction, we need to know the size of the data stored at location a in H. Because of our conventions, this would always be 4 or 8; in C, other sizes would be possible.

This leaves us with a puzzle: how do we *write* to memory? In C0 (and C) this is accomplished via assignments where the left-hand side dictates the destination of the write operation. These are sometimes called *l-values*, where l stands for *left-hand side*.

5 Writing to Heap Destinations

We define *destinations* (or *l-values*)

$$d ::= x \mid *d$$

Adding arrays and structs will add more kinds of destinations. Every kind of destination is also a valid expression, so we can just type destinations as expressions.

$$\frac{\Gamma \vdash d : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{assign}(d, e) : [\tau']}$$

Recall that in typings of statements, τ' is the return type of the function that we are currently typing. The rule for assignment is valid for every τ'.

In the operational semantics we now distinguish variables from other destinations, since variables are on the stack (or in registers), while destinations $*d$ are on the heap. First, a reminder for assignment if the destination is a variable.

$$\begin{array}{lcl} H \; ; S \; ; \eta \vdash \mathsf{assign}(x, e) \blacktriangleright K & \longrightarrow & H \; ; S \; ; \eta \vdash e \rhd (\mathsf{assign}(x, \_) \; , K) \\ H \; ; S \; ; \eta \vdash c \rhd (\mathsf{assign}(x, \_) \; , K) & \longrightarrow & H \; ; S \; ; \eta[x \mapsto c] \rhd \mathsf{nop} \blacktriangleright K \end{array}$$

If the destination is *not* a variable, we proceed from left to right, first determining the address which is the real memory destination, then evaluating the right-hand side.

$$\begin{array}{lcll} H \; ; S \; ; \eta \vdash \mathsf{assign}(*d, e) \blacktriangleright K & \longrightarrow & H \; ; S \; ; \eta \vdash d \rhd (\mathsf{assign}(*\_, e) \; , K) & \\ H \; ; S \; ; \eta \vdash a \rhd (\mathsf{assign}(*\_, e) \; , K) & \longrightarrow & H \; ; S \; ; \eta \vdash e \rhd (\mathsf{assign}(*a, \_) \; , K) & \\ H \; ; S \; ; \eta \vdash c \rhd (\mathsf{assign}(*a, \_) \; , K) & \longrightarrow & H[a \mapsto c] \; ; S \; ; \eta \vdash \mathsf{nop} \blacktriangleright K & (a \neq 0) \\ H \; ; S \; ; \eta \vdash c \rhd (\mathsf{assign}(*a, \_) \; , K) & \longrightarrow & \mathsf{exception(mem)} \quad (a = 0) & \end{array}$$

Detail: Evaluating Assignments

Based on the rules above, what should happen in the following code fragments.

```
int* p = NULL;
*p = 1/0;
```

First we define p to be 0. Then we evaluate the assignment from left to right. This means we first evaluate p to 0. Second we evaluate $1/0$. This will raise an arithmetic exception, which is therefore the outcome of the execution.

```
int** p = NULL;
**p = 1/0;
```

First we define p to be 0. Then we evaluate the assignment from left to right. This means we first evaluate $*p$. Since the value of p is 0 this raises a memory exception, which is therefore the outcome of the execution.

6 Arrays

Arrays are in many ways similar to pointers, but there are no null arrays. We'll discuss default arrays below. For now, though, this is a simplification since the typing rules are more straightforward.

$$\begin{array}{lcl} \tau & ::= & \ldots \mid \tau[\,] \\ e & ::= & \ldots \mid \mathsf{alloc\_array}(\tau, e) \mid e_1[e_2] \\ d & ::= & \ldots \mid d[e] \end{array}$$

$$\frac{\Gamma \vdash e : \mathsf{int}}{\Gamma \vdash \mathsf{alloc\_array}(\tau, e) : \tau[\,]} \qquad \frac{\Gamma \vdash e_1 : \tau[\,] \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1[e_2] : \tau}$$

The dynamic semantics for allocation obtains a fresh segment of memory and initializes all n elements of the array with the default value of type τ.

$$\begin{array}{lcl} H \,;\, S \,;\, \eta \vdash \mathsf{alloc\_array}(\tau, e) \rhd K & \longrightarrow & H \,;\, S \,;\, \eta \vdash e \rhd (\mathsf{alloc\_array}(\tau, \_)\ ,\, K) \\ H \,;\, S \,;\, \eta \vdash n \rhd (\mathsf{alloc\_array}(\tau, \_)\ ,\, K) & \longrightarrow & H' \,;\, S \,;\, \eta \vdash a \rhd K \qquad (n \geq 0) \\ & & a = H(\mathsf{next}) \\ \multicolumn{3}{c}{H' = H[a + 0|\tau| \mapsto \mathsf{default}(\tau), \ldots, a + (n-1)|\tau| \mapsto \mathsf{default}(\tau), \mathsf{next} \mapsto a + n|\tau|]} \\ H \,;\, S \,;\, \eta \vdash n \rhd (\mathsf{alloc\_array}(\tau, \_)\ ,\, K) & \longrightarrow & \mathsf{exception(mem)} \qquad (n < 0) \end{array}$$

Array access evaluates from left to right and then computes the correct memory address for the value.

$$\begin{array}{lcll} H \,;\, S \,;\, \eta \vdash e_1[e_2] \rhd K & \longrightarrow & H \,;\, S \,;\, \eta \vdash e_1 \rhd (\_[e_2]\ ,\, K) & \\ H \,;\, S \,;\, \eta \vdash a \rhd (\_[e_2]\ ,\, K) & \longrightarrow & H \,;\, S \,;\, \eta \vdash e_2 \rhd (a[\_]\ ,\, K) & \\ H \,;\, S \,;\, \eta \vdash i \rhd (a[\_]\ ,\, K) & \longrightarrow & H \,;\, S \,;\, \eta \vdash H(a + i|\tau|) \rhd K & \\ & & & a \neq 0, 0 \leq i < \mathsf{length}(a), a : \tau[\,] \\ H \,;\, S \,;\, \eta \vdash i \rhd (a[\_]\ ,\, K) & \longrightarrow & \mathsf{exception(mem)} & a = 0 \text{ or } i < 0 \text{ or } i \geq \mathsf{length}(a) \end{array}$$

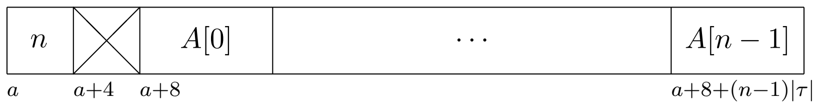
There are two significant complications here: where do we obtain the length of the array stored at address a, and where do we get the type τ?

The second is actually easier: when we compile an array access, we will know the type of e_1. It must be of the form $\tau[\,]$ for some τ. Then we calculate its size *at compile time* and generate code to multiply it by the index i.

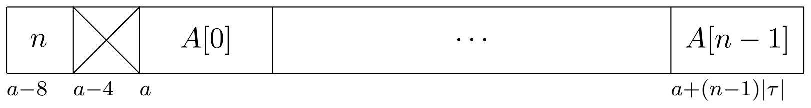
Finding the length of the array is actually harder, since it is not known at compile time. This is because array allocation has the form $\mathsf{alloc\_array}(\tau, e)$ where e is an arbitrary expression that should evaluate to the number of elements in the array to allocate.

Detail: Storing the Array Length

One possibility is to allocate a few additional bytes to store the length of the array. This could be layed out as follows, where a is the address of the array A with elements of type τ.



Alternatively, we could lay it out with the address a pointing to the first array element. This simplifies the address arithmetic, and would also allow passing this pointer directly to C (which would not care about the length information to the left).



The reason we locate the length n at $a - 8$ and not $a - 4$ is so that a itself will be aligned at 0 modulo 8, if the whole memory block as returned from calloc is aligned that way.

Under this second regime, the code pattern for $e_1[e_2]$ with $e_1 : \tau[\,]$ and $|\tau| = k$ could be like this:

```
cogen($e_1, a$)              ($a$ new)
cogen($e_2, i$)              ($i$ new)
$a_1 \leftarrow a - 8$
$t_2 \leftarrow M[a_1]$
if ($i < 0$) goto error
if ($i \geq t_2$) goto error
$a_3 \leftarrow i * \$k$
$a_4 \leftarrow a + a_3$
$t_5 \leftarrow M[a_4]$
```

Here, a, a_1, a_3, a_4 would be 64 bit temps, t_2 would be 32 bits, and t_5 would be k bytes. We have written $\$k$ to indicate that this is an immediate operand (that is, a compile-time constant). Some compound memory operands can be used on x86-64 to avoid some intermediate computation such as a_1 or a_4. Also, we can exploit properties of two's complement arithmetic and combine the two comparisons into a single unsigned comparison of i and t_2.

Of course, there are still limits to interoperability with C: if C passes an array to a C0 program, we somehow need to find out its length and marshal it somewhere else so we can add the length information. Alternatively, we can compile the code in *unsafe* mode where array bounds are not checked, which is just what C does.

Executing assignments with the new destinations is quite similar to reading.

$$
\begin{array}{lcl}
H \; ; S \; ; \eta \vdash \mathsf{assign}(d[e_2], e_3) \blacktriangleright K & \longrightarrow & H \; ; S \; ; \eta \vdash d \rhd (\mathsf{assign}(\_[e_2], e_3) \; , K) \\
H \; ; S \; ; \eta \vdash a \rhd (\mathsf{assign}(\_[e_2], e_3) \; , K) & \longrightarrow & H \; ; S \; ; \eta \vdash e_2 \rhd (\mathsf{assign}(a[\_], e_3) \; , K) \\
H \; ; S \; ; \eta \vdash i \rhd (\mathsf{assign}(a[\_], e_3) \; , K) & \longrightarrow & H \; ; S \; ; \eta \vdash e_3 \rhd (\mathsf{assign}(a + i|\tau|, \_) \; , K) \\
 & & \qquad a \neq 0, 0 \leq i < \mathsf{length}(a), a : \tau[\,] \\
H \; ; S \; ; \eta \vdash i \rhd (\mathsf{assign}(a[\_], e_3) \; , K) & \longrightarrow & \mathsf{exception}(\mathsf{mem}) \\
 & & \qquad a = 0 \text{ or } i < 0 \text{ or } i \geq \mathsf{length}(a) \\
H \; ; S \; ; \eta \vdash c \rhd (\mathsf{assign}(b, \_) \; , K) & \longrightarrow & H[b \mapsto c] \; ; S \; ; \eta \vdash \mathsf{nop} \blacktriangleright K
\end{array}
$$

7 Detail: Default Values of Array Type

Each type has a default value. For integers it is 0, for booleans 0 (which represents false), and for pointers it is 0 (which represents null). The default for arrays is also 0, which represents an array of size 0. We can never legally access any element of this default array, since the condition that the index must be in bounds can never be satisfied. Nevertheless, arrays can be compared for equality and disequality (which is a comparison of their address), so zero-sized arrays are not entirely useless. In particular, $\mathsf{alloc\_array}(0)$ must return a fresh zero-sized array that's different from all other arrays already allocated, and also different from the default array of size 0.

The fact that $a = 0$ is a valid array address creates an issue when we try to access $M[a - 8]$ to obtain its size. We could rely on the operating system to raise a memory exception, although that may not be reliably so. To be sure, we should check whether a is 0 before doing address calculation. Of course, if we are in *unsafe* mode when bounds-checking is turned off (which we will implement in Lab 5), then this is not necessary.

8 Detail: Compound Assignment Operators

Previously, we could expand `x += e` to `x = x + e`. However, with the addition of arrays, this has become problematic. The difficulty is `d1[e2] += e3`. After syntactic expansion we obtain `d1[e2] = d1[e2] + e3` in which both d_1 and e_2 would be evaluated twice. Since evaluation of expressions and destinations now can have an effect, that effect would be unexpectedly repeated. Instead we have to more-or-less repeat the rules for assignment with appropriate checks for out-of-bounds access and arithmetic exceptions. The address-of operator introduced in the next lecture can make this somewhat more elegant.